IEC 62541-3

Edition 3.0 2020-07

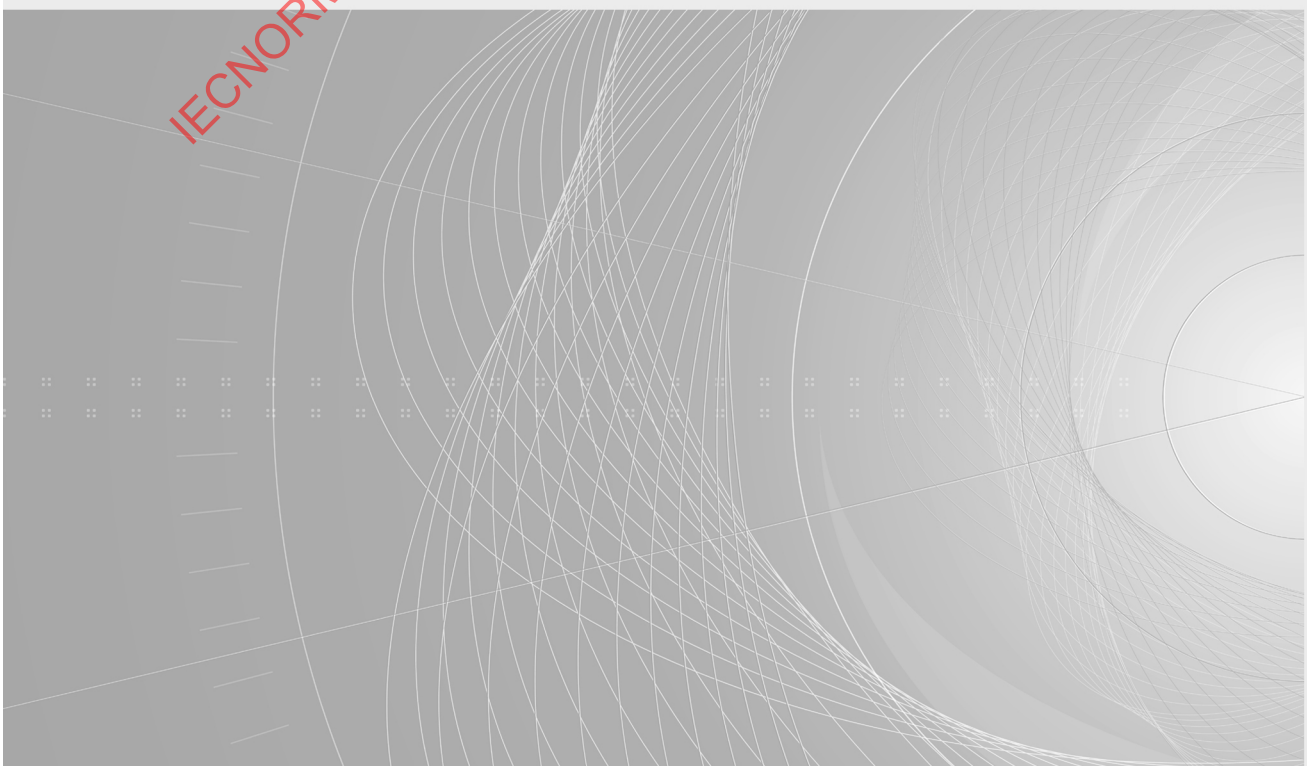
INTERNATIONAL STANDARD

NORME INTERNATIONALE

colour inside

OPC unified architecture – Part 3: Address Space Model

Architecture unifiée OPC – Partie 3: Modèle d'espace d'adressage



IEC 62541-3:2020-07(en-fr)

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

Tel.: +41 22 919 02 11
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 corrigendum or an amendment might have been published.

IEC publications search - webstore.iec.ch/advsearchform
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 It also gives information on projects, replaced and withdrawn publications.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 once a month by email.

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: sales@iec.ch.

Electropedia - www.electropedia.org
The world's leading online dictionary on electrotechnology, containing more than 22 000 terminological entries in English and French, with equivalent terms in 16 additional languages. Also known as the International Electrotechnical Vocabulary (IEV) online.

IEC Glossary - std.iec.ch/glossary
67 000 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.

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.

Recherche de publications IEC - webstore.iec.ch/advsearchform
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 une fois par mois par email.

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

Electropedia - www.electropedia.org
Le premier dictionnaire d'électrotechnologie en ligne au monde, avec plus de 22 000 articles terminologiques en anglais et en français, ainsi que les termes équivalents dans 16 langues additionnelles. Egalement appelé Vocabulaire Electrotechnique International (IEV) en ligne.

Glossaire IEC - std.iec.ch/glossary
67 000 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.

IEC 62541-3

Edition 3.0 2020-07

INTERNATIONAL STANDARD

NORME INTERNATIONALE

colour inside

OPC unified architecture – Part 3: Address Space Model

Architecture unifiée OPC – Partie 3: Modèle d'espace d'adressage

INTERNATIONAL ELECTROTECHNICAL COMMISSION

COMMISSION ELECTROTECHNIQUE INTERNATIONALE

ICS 25.040.40; 35.100.05

ISBN 978-2-8322-8580-0

CONTENTS

INTERNATIONAL ELECTROTECHNICAL COMMISSION

_____

OPC UNIFIED ARCHITECTURE –

Part 3: Address Space Model

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 62541-3 has been prepared by subcommittee 65E: Devices and integration in enterprise systems, of IEC technical committee 65: Industrial-process measurement, control and automation.

This third edition cancels and replaces the second edition published in 2015.

This edition includes the following significant technical changes with respect to the previous edition:

a) Added new improved approach for exposing structure definitions. An Attribute on the DataType Node now simply contains a binary description.
b) Added new flags for Variables to indicate atomicity when reading or writing.
c) Added Roles and Permissions to allow configuration of a role-based authorization.
d) Added new data types: "Union", "Decimal", "OptionSet", "DateString", "TimeString", "DurationString", NormalizedString", "DecimalString", and "AudioDataType".

e) Added definition on how to use the ModellingRules OptionalPlaceHolder and MandatoryPlaceHolder for Methods.

f) Added optional Properties “MaxCharacters” and “MaxByteStringLength” to Variable Nodes.

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

FDIS	Report on voting
65E/715/FDIS	65E/731/RVD

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

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

Throughout this document and the other parts of the IEC 62541 series, certain document conventions are used:

Italics are used to denote a defined term or definition that appears in Clause 3 in one of the parts of the series.

Italics are also used to denote the name of a service input or output parameter or the name of a structure or element of a structure that are usually defined in tables.

The *italicized terms and names* are also, with a few exceptions, written in camel-case (the practice of writing compound words or phrases in which the elements are joined without spaces, with each element's initial letter capitalized within the compound). For example the defined term is *AddressSpace* instead of Address Space. This makes it easier to understand that there is a single definition for *AddressSpace*, not separate definitions for Address and Space.

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

The committee has decided that the contents of this document will remain unchanged until the stability date indicated on the IEC website under "http://webstore.iec.ch" in the data related to the specific document. At this date, the document 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 3: Address Space Model

1 Scope

This part of IEC 62541 defines the OPC Unified Architecture (OPC UA) *AddressSpace* and its *Objects*. This document is the OPC UA meta model on which OPC UA information models are based.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. 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 62541-4, *OPC Unified Architecture – Part 4: Services*

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

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

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

ISO/IEC/IEEE 60559:2011, *Information technology – Microprocessor Systems – Floating-Point arithmetic*

ISO 639 (all parts), *Codes for the representation of names of languages*

ISO 3166 (all parts), *Codes for the representation of names of countries and their subdivisions*

ISO 8601 (all parts), *Date and time – Representations for information interchange*

IETF RFC 5646, Tags for Identifying Languages
http://tools.ietf.org/html/rfc5646

Unicode Standard Annex #15: Unicode Normalization Forms,
http://www.unicode.org/reports/tr15/

W3C XML Schema Definition Language (XSD) Part 2: DataTypes
http://www.w3.org/TR/xmlschema-2/

TAI: International Atomic Time
http://www.bipm.org/en/bipm-services/timescales/tai.html

3 Terms, definitions, abbreviated terms and conventions

3.1 Terms and definitions

For the purposes of this document, the terms and definitions given in IEC TR 62541-1 and the following apply.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

- IEC Electropedia: available at http://www.electropedia.org/
- ISO Online browsing platform: available at http://www.iso.org/obp

3.1.1
DataType
instance of a *DataType Node* that is used together with the *ValueRank Attribute* to define the data type of a *Variable*

3.1.2
DataTypeId
NodeId of a *DataType Node*

3.1.3
DataVariable
Variable that represents the *value* of an *Object*, either directly or indirectly for complex *Variables*, where the *Variables* are always the *TargetNode* of a *HasComponent Reference*

3.1.4
EventType
ObjectType Node that represents the type definition of an *Event*

3.1.5
Hierarchical Reference
Reference that is used to construct hierarchies in the *AddressSpace*

Note 1 to entry: All hierarchical ReferenceTypes are derived from HierarchicalReferences.

3.1.6
InstanceDeclaration
Node that is used by a complex *TypeDefinitionNode* to expose its complex structure

Note 1 to entry: This is an instance used by a type definition.

3.1.7
ModellingRule
metadata of an *InstanceDeclaration* that defines how the *InstanceDeclaration* will be used for instantiation and also defines subtyping rules for an *InstanceDeclaration*

3.1.8
Property
Variable that is the *TargetNode* for a *HasProperty Reference*

Note 1 to entry: *Properties* describe the characteristics of a *Node*.

3.1.9
SourceNode
Node having a *Reference* to another *Node*

EXAMPLE: In the *Reference* “A contains B”, “A” is the *SourceNode*.

3.1.10
TargetNode
Node that is referenced by another *Node*

EXAMPLE: In the *Reference* “A contains B”, “B” is the *TargetNode*.

3.1.11
TypeDefinitionNode
Node that is used to define the type of another *Node*

Note 1 to entry: *ObjectType* and *VariableType Nodes* are *TypeDefinitionNodes*.

3.1.12
VariableType
Node that represents the type definition for a *Variable*

3.2 Abbreviated terms

UA Unified Architecture
UML Unified Modeling Language
URI Uniform Resource Identifier
W3C World Wide Web Consortium
XML eXtensible Markup Language

3.3 Conventions

3.3.1 Conventions for AddressSpace figures

Nodes and their *References* to each other are illustrated using figures. Figure 1 illustrates the conventions used in these figures.

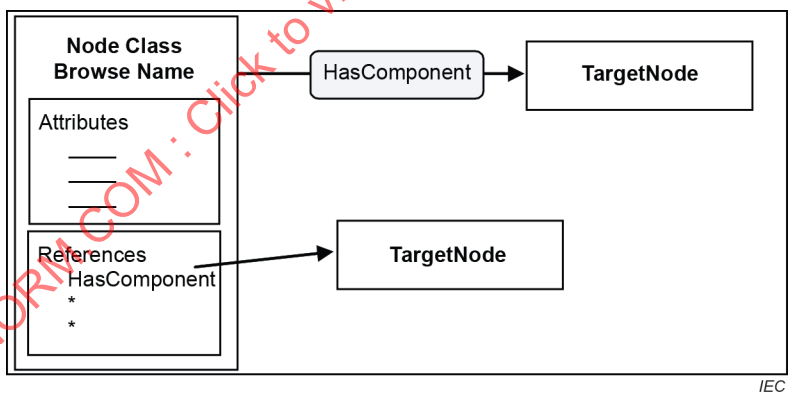


Figure 1 – AddressSpace Node diagrams

In these figures, rectangles represent *Nodes*. *Node* rectangles may be titled with one or two lines of text. When two lines are used, the first text line in the rectangle identifies the *NodeClass* and the second line contains the *BrowseName*. When one line is used, it contains the *BrowseName*.

Node rectangles may contain boxes used to define their *Attributes* and *References*. Specific names in these boxes identify specific *Attributes* and *References*.

Shaded rectangles with rounded corners and with arrows passing through them represent *References*. The arrow that passes through them begins at the *SourceNode* and points to the *TargetNode*. *References* may also be shown by drawing an arrow that starts at the *Reference* name in the “References” box and ends at the *TargetNode*.

3.3.2 Conventions for defining NodeClasses

Clause 5 defines *AddressSpace NodeClasses*. Table 1 describes the format of the tables used to define *NodeClasses*.

Table 1 – NodeClass Table Conventions

Name	Use	Data Type	Description
Attributes			
“Attribute name”	“M” or “O”	Data type of the *Attribute*	Defines the *Attribute*
References			
“Reference name”	“1”, “0..1” or “0..*”	Not used	Describes the use of the *Reference* by the *NodeClass*
Standard Properties			
“Property name”	“M” or “O”	Data type of the *Property*	Defines the *Property*

The Name column contains the name of the *Attribute*, the name of the *ReferenceType* used to create a *Reference* or the name of a *Property* referenced using the *HasProperty Reference*.

The Use column defines whether the *Attribute* or *Property* is mandatory (M) or optional (O). When mandatory the *Attribute* or *Property* shall exist for every *Node* of the *NodeClass*. For *References* it specifies the cardinality. The following values may apply:

- “0..*” identifies that there are no restrictions, that is, the *Reference* does not have to be provided but there is no limitation how often it can be provided;
- “0..1” identifies that the *Reference* is provided at most once;
- “1” identifies that the *Reference* shall be provided exactly once.

The Data Type column contains the name of the *DataType* of the *Attribute* or *Property*. It is not used for *References*.

The Description column contains the description of the *Attribute*, the *Reference* or the *Property*.

Only this document may define *Attributes*. Thus, all *Attributes* of the *NodeClass* are specified in the table and may only be extended by other parts of the IEC 62541 series.

This document also defines *ReferenceTypes*, but *ReferenceTypes* may also be specified by a *Server* or by a *Client* using the *NodeManagement Services* specified in IEC 62541-4. Thus, the *NodeClass* tables contained in this document may contain the base *ReferenceType* called *References* identifying that any *ReferenceType* may be used for the *NodeClass,* including system specific *ReferenceTypes*. The *NodeClass* tables only specify how the *NodeClasses* can be used as *SourceNodes* of *References*, not as *TargetNodes*. If a *NodeClass* table allows a *ReferenceType* for its *NodeClass* to be used as *SourceNode*, this is also true for subtypes of the *ReferenceType*. However, subtypes of the *ReferenceType* may restrict its *SourceNodes*.

This document defines *Properties*, but *Properties* can be defined by other standards organizations or vendors and *Nodes* can have *Properties* that are not standardized. *Properties* defined in this document are defined by their name, which is mapped to the *BrowseName* having the *NamespaceIndex* 0, which represents the *Namespace* for OPC UA.

The Use column (optional or mandatory) does not imply a specific *ModellingRule* for *Properties*. Different *Server* implementations will choose to use *ModellingRules* appropriate for them.

4 AddressSpace concepts

4.1 Overview

Clause 4 defines the concepts of the *AddressSpace*. Clause 5 defines the *NodeClasses* of the *AddressSpace* representing the *AddressSpace* concepts. Clause 6 defines details on the type model for *ObjectTypes* and *VariableTypes*. Standard *ReferenceTypes*, *DataTypes* and *EventTypes* are defined in Clauses 7 to 9.

The informative Annex A describes general considerations on how to use the Address Space Model and the informative Annex B provides a UML Model of the Address Space Model. The normative Annex C defines a graphical notation for OPC UA data.

4.2 Object Model

The primary objective of the OPC UA *AddressSpace* is to provide a standard way for *Servers* to represent *Objects* to *Clients*. The OPC UA Object Model has been designed to meet this objective. It defines *Objects* in terms of *Variables* and *Methods*. It also allows relationships to other *Objects* to be expressed. Figure 2 illustrates the model.

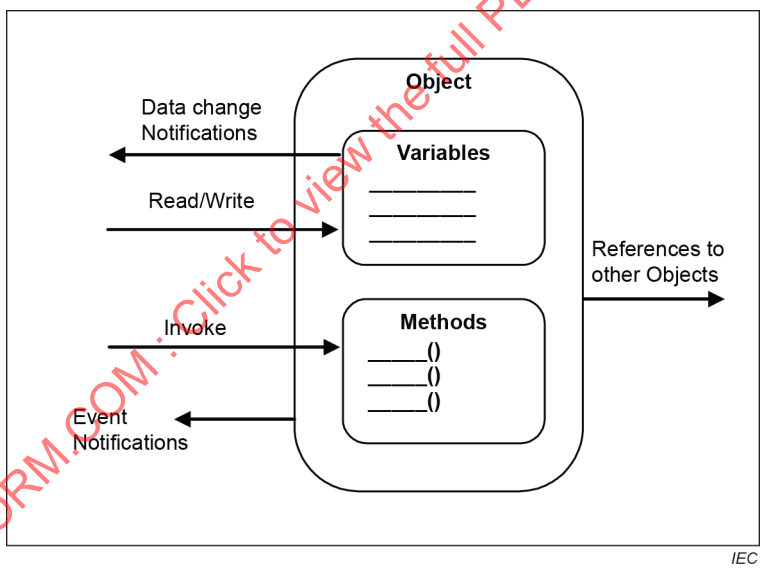


Figure 2 – OPC UA Object Model

The elements of this model are represented in the *AddressSpace* as *Nodes*. Each *Node* is assigned to a *NodeClass* and each *NodeClass* represents a different element of the Object Model. Clause 5 defines the *NodeClasses* used to represent this model.

4.3 Node Model

4.3.1 General

The set of *Objects* and related information that the OPC UA *Server* makes available to *Clients* is referred to as its *AddressSpace*. The model for *Objects* is defined by the OPC UA Object Model (see 4.2).

Objects and their components are represented in the *AddressSpace* as a set of *Nodes* described by *Attributes* and interconnected by *References*. Figure 3 illustrates the model of a *Node* and the remainder of 4.3 discusses the details of the Node Model.

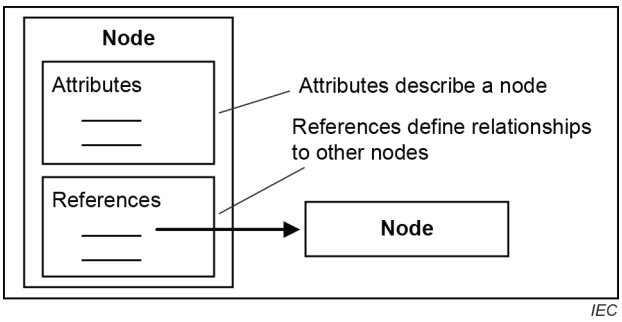


Figure 3 – AddressSpace Node Model

4.3.2 NodeClasses

NodeClasses are defined in terms of the *Attributes* and *References* that shall be instantiated (given values) when a *Node* is defined in the *AddressSpace*. *Attributes* are discussed in 4.3.3 and *References* in 4.3.4.

Clause 5 defines the *NodeClasses* for the OPC UA *AddressSpace*. These *NodeClasses* are referred to collectively as the metadata for the *AddressSpace*. Each *Node* in the *AddressSpace* is an instance of one of these *NodeClasses*. No other *NodeClasses* shall be used to define *Nodes*, and as a result, *Clients* and *Servers* are not allowed to define *NodeClasses* or extend the definitions of these *NodeClasses*.

4.3.3 Attributes

Attributes are data elements that describe *Nodes*. *Clients* can access *Attribute* values using Read, Write, Query, and Subscription/MonitoredItem *Services*. These *Services* are defined in IEC 62541-4.

Attributes are elementary components of *NodeClasses*. *Attribute* definitions are included as part of the *NodeClass* definitions in Clause 5 and, therefore, are not included in the *AddressSpace*.

Each *Attribute* definition consists of an attribute id (for attribute ids of *Attributes*, see IEC 62541-6), a name, a description, a data type and a mandatory/optional indicator. The set of *Attributes* defined for each *NodeClass* shall not be extended by *Clients* or *Servers*.

When a *Node* is instantiated in the *AddressSpace*, the values of the *NodeClass Attributes* are provided. The mandatory/optional indicator for the *Attribute* indicates whether the *Attribute* shall be instantiated.

4.3.4 References

References are used to relate *Nodes* to each other. They can be accessed using the browsing and querying *Services* defined in IEC 62541-4.

Like *Attributes*, they are defined as fundamental components of *Nodes*. Unlike *Attributes*, *References* are defined as instances of *ReferenceType Nodes*. *ReferenceType Nodes* are visible in the *AddressSpace* and are defined using the *ReferenceType NodeClass* (see 5.3).

The *Node* that contains the *Reference* is referred to as the *SourceNode* and the *Node* that is referenced is referred to as the *TargetNode*. The combination of the *SourceNode*, the

ReferenceType and the *TargetNode* is used in OPC UA *Services* to uniquely identify *References*. Thus, each *Node* can reference another *Node* with the same *ReferenceType* only once. Any subtypes of concrete *ReferenceTypes* are considered to be equal to the base concrete *ReferenceTypes* when identifying *References* (see 5.3 for subtypes of *ReferenceTypes*). Figure 4 illustrates this model of a *Reference*.

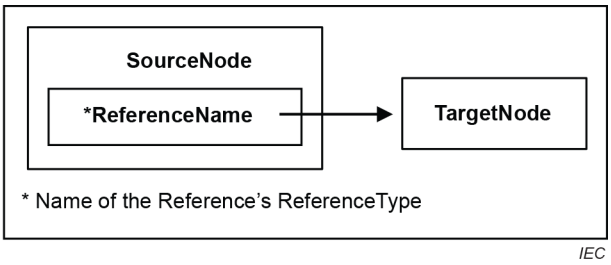


Figure 4 – Reference Model

The *TargetNode* of a *Reference* may be in the same *AddressSpace* or in the *AddressSpace* of another OPC UA *Server*. *TargetNodes* located in other *Servers* are identified in OPC UA *Services* using a combination of the remote *Server* name and the identifier assigned to the *Node* by the remote *Server*.

OPC UA does not require that *the TargetNode* exists, thus *References* may point to a *Node* that does not exist.

4.4 Variables

4.4.1 General

Variables are used to represent *values*. Two types of *Variables* are defined, *Properties* and *DataVariables*. They differ in the kind of data that they represent and whether they can contain other *Variables*.

4.4.2 Properties

Properties are *Server*-defined characteristics of *Objects*, *DataVariables* and other *Nodes*. *Properties* differ from *Attributes* in that they characterize *what* the *Node* represents, such as a device or a purchase order. *Attributes* define additional metadata that is instantiated for all *Nodes* from a *NodeClass*. *Attributes* are common to all *Nodes* of a *NodeClass* and only defined by this document whereas *Properties* can be *Server*-defined.

For example, an *Attribute* defines the *DataType* of *Variables* whereas a *Property* can be used to specify the engineering unit of some *Variables*.

To prevent recursion, *Properties* are not allowed to have *Properties* defined for them. To easily identify *Properties*, the *BrowseName* of a *Property* shall be unique in the context of the *Node* containing the *Properties* (see 5.6.3 for details).

A *Node* and its *Properties* shall always reside in the same *Server*.

4.4.3 DataVariables

DataVariables represent the content of an *Object*. For example, a file *Object* may be defined that contains a stream of bytes. The stream of bytes may be defined as a *DataVariable* that is an array of bytes. *Properties* may be used to expose the creation time and owner of the file *Object*.

For example, if a *DataVariable* is defined by a data structure that contains two fields, “startTime” and “endTime” then it might have a *Property* specific to that data structure, such as “earliestStartTime”.

As another example, function blocks in control systems might be represented as *Objects*. The parameters of the function block, such as its setpoints, may be represented as *DataVariables*. The function block *Object* might also have *Properties* that describe its execution time and its type.

DataVariables may have additional *DataVariables*, but only if they are complex. In this case, their *DataVariables* shall always be elements of their complex definitions. Following the example introduced by the description of *Properties* in 4.4.2, the *Server* could expose “startTime” and “endTime” as separate components of the data structure.

As another example, a complex *DataVariable* may define an aggregate of temperature values generated by three separate temperature transmitters that are also visible in the *AddressSpace*. In this case, this complex *DataVariable* could define *HasComponent References* from it to the individual temperature values that it is composed of.

4.5 TypeDefinitionNodes

4.5.1 General

OPC UA *Servers* shall provide type definitions for *Objects* and *Variables*. The *HasTypeDefinition Reference* shall be used to link an instance with its type definition represented by a *TypeDefinitionNode*. Type definitions are required; however, IEC 62541-5 defines a *BaseObjectType*, a *PropertyType,* and a *BaseDataVariableType* so a *Server* can use such a base type if no more specialized type information is available. *Objects* and *Variables* inherit the *Attributes* specified by their *TypeDefinitionNode* (see 6.4 for details).

In some cases, the *NodeId* used by the *HasTypeDefinition Reference* will be well-known to *Clients* and *Servers*. Organizations may define *TypeDefinitionNodes* that are well-known in the industry. Well-known *NodeIds* of *TypeDefinitionNodes* provide for commonality across OPC UA *Servers* and allow *Clients* to interpret the *TypeDefinitionNode* without having to read it from the *Server*. Therefore, *Servers* may use well-known *NodeIds* without representing the corresponding *TypeDefinitionNodes* in their *AddressSpace*. However, the *TypeDefinition-Nodes* shall be provided for generic *Clients*. These *TypeDefinitionNodes* may exist in another *Server*.

The following example, illustrated in Figure 5, describes the use of the *HasTypeDefinition Reference*. In this example, a setpoint parameter “SP” is represented as a *DataVariable* in the *AddressSpace*. This *DataVariable* is part of an *Object* not shown in the figure.

To provide for a common setpoint definition that can be used by other *Objects*, a specialized *VariableType* is used. Each setpoint *DataVariable* that uses this common definition will have a *HasTypeDefinition Reference* that identifies the common “SetPoint” *VariableType*.

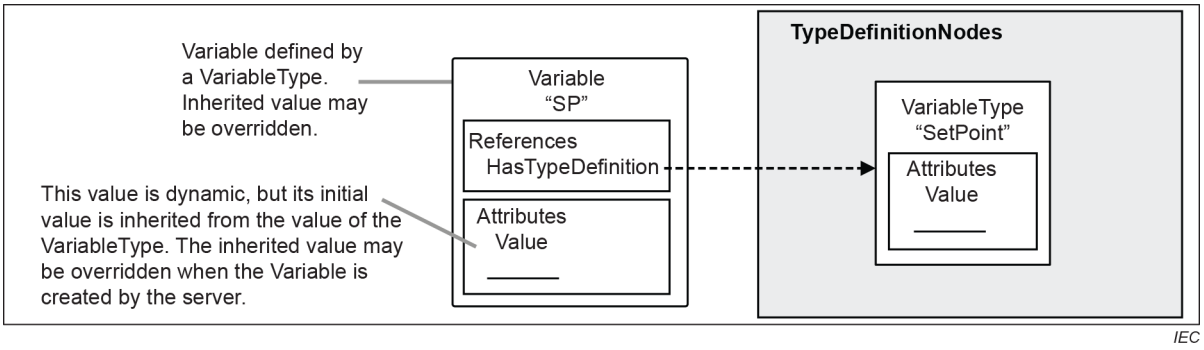


Figure 5 – Example of a Variable defined by a VariableType

4.5.2 Complex TypeDefinitionNodes and their InstanceDeclarations

TypeDefinitionNodes can be complex. A complex *TypeDefinitionNode* also defines *References* to other *Nodes* as part of the type definition. The *ModellingRules* defined in 6.4.4 specify how those *Nodes* are handled when creating an instance of the type definition.

A *TypeDefinitionNode* references instances instead of other *TypeDefinitionNodes* to allow unique names for several instances of the same type, to define default values and to add *References* for those instances that are specific to this complex *TypeDefinitionNode* and not to the *TypeDefinitionNode* of the instance. For example, in Figure 6 the *ObjectType* "AI_BLK_TYPE", representing a function block, has a *HasComponent Reference* to a *Variable* "SP" of the *VariableType* "SetPoint". "AI_BLK_TYPE" could have an additional setpoint *Variable* of the same type using a different name. It could add a *Property* to the *Variable* that was not defined by its *TypeDefinitionNode* "SetPoint". And it could define a default value for "SP", that is, each instance of "AI_BLK_TYPE" would have a *Variable* "SP" initially set to this value.

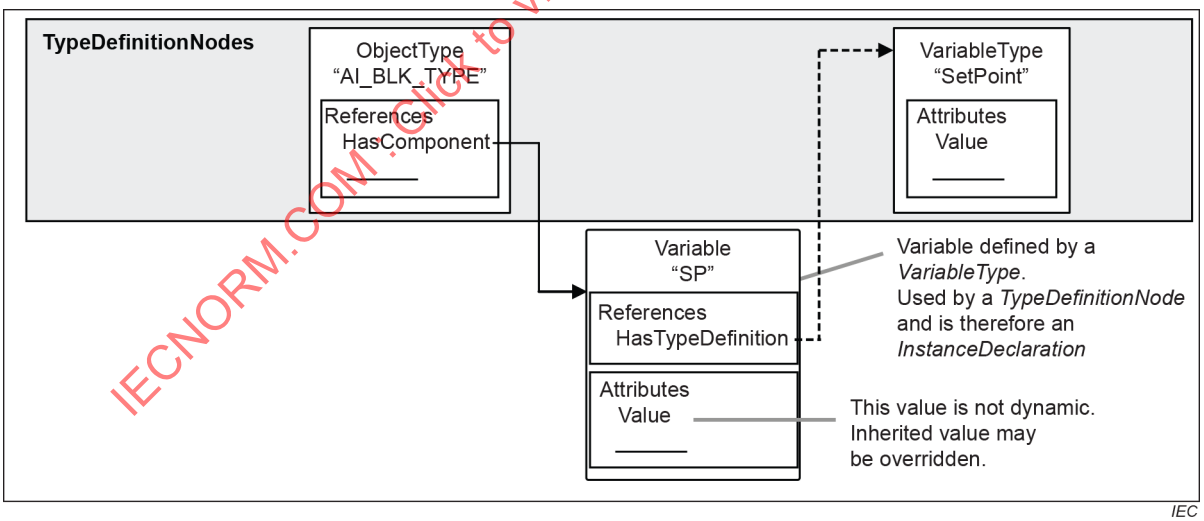


Figure 6 – Example of a Complex TypeDefinition

This approach is commonly used in object-oriented programming languages in which the variables of a class are defined as instances of other classes. When the class is instantiated, each variable is also instantiated, but with the default values (constructor values) defined for the containing class. That is, typically, the constructor for the component class runs first, followed by the constructor for the containing class. The constructor for the containing class may override component values set by the component class.

To distinguish instances used for the type definitions from instances that represent real data, those instances are called *InstanceDeclarations*. However, this term is used to simplify this

specification, if an instance is an *InstanceDeclaration* or not is only visible in the *AddressSpace* by following its *References*. Some instances may be shared and therefore referenced by *TypeDefinitionNodes*, *InstanceDeclarations* and instances. This is similar to class variables in object-oriented programming languages.

4.5.3 Subtyping

This document allows subtyping of type definitions. The subtyping rules are defined in Clause 6. Subtyping of *ObjectTypes* and *VariableTypes* allows:

- *Clients* that only know the supertype to handle an instance of the subtype as if it were an instance of the supertype;
- instances of the supertype to be replaced by instances of the subtype;
- specialized types that inherit common characteristics of the base type.

In other words, subtypes reflect the structure defined by their supertype but may add additional characteristics. For example, a vendor may wish to extend a general "TemperatureSensor" *VariableType* by adding a *Property* providing the next maintenance interval. The vendor would do this by creating a new *VariableType* which is a *TargetNode* for a *HasSubtype* reference from the original *VariableType* and adding the new *Property* to it.

4.5.4 Instantiation of complex TypeDefinitionNodes

The instantiation of complex *TypeDefinitionNodes* depends on the *ModellingRules* defined in 6.4.4. However, the intention is that instances of a type definition will reflect the structure defined by the *TypeDefinitionNode*. Figure 7 shows an instance of the *TypeDefinitionNode* "AI_BLK_TYPE", where the *ModellingRule Mandatory*, defined in 6.4.4.5.2, was applied for its containing *Variable*. Thus, an instance of "AI_BLK_TYPE", called "AI_BLK_1", has a *HasTypeDefinition Reference* to "AI_BLK_TYPE". It also contains a *Variable* "SP" having the same *BrowseName* as the *Variable* "SP" used by the *TypeDefinitionNode* and thereby reflects the structure defined by the *TypeDefinitionNode*.

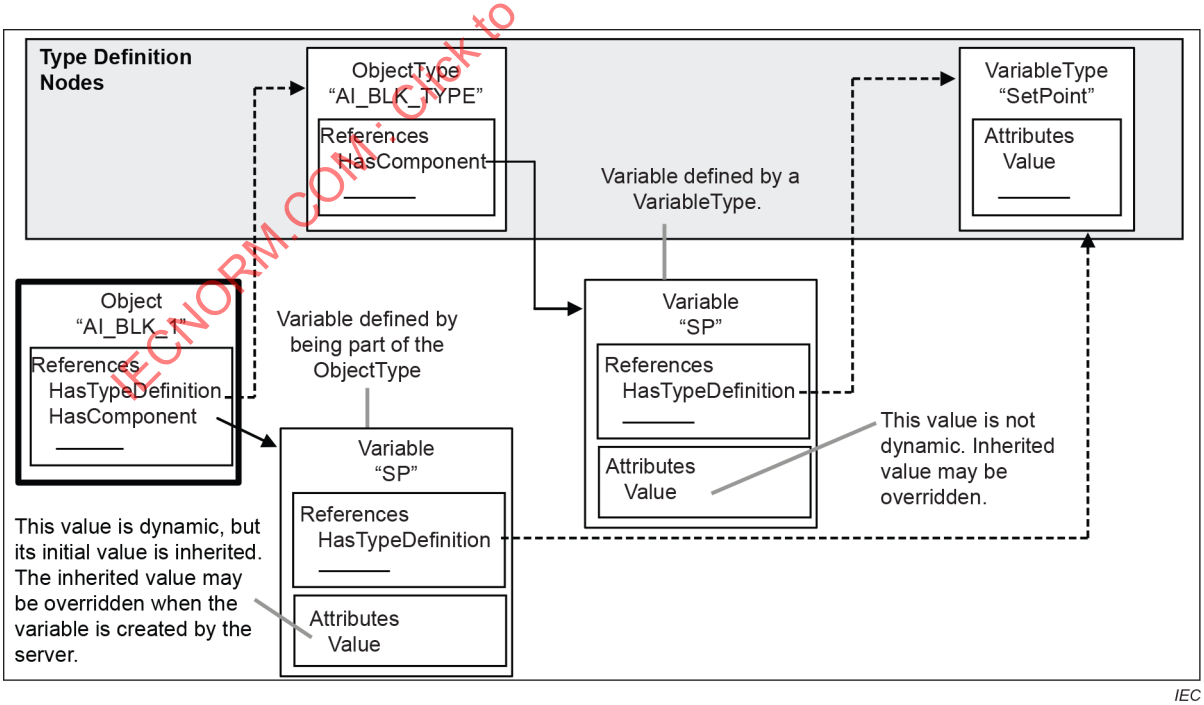


Figure 7 – Object and its Components defined by an ObjectType

A *Client* knowing the *ObjectType* "AI_BLK_TYPE" can use this knowledge to directly browse to the containing *Nodes* for each instance of this type. This allows programming against the

TypeDefinitionNode. For example, a graphical element may be programmed in the *Client* that handles all instances of “AI_BLK_TYPE” in the same way by showing the value of “SP”.

There are several constraints related to programming against the *TypeDefinitionNode*. A *TypeDefinitionNode* or an *InstanceDeclaration* shall never reference two *Nodes* having the same *BrowseName* using forward *Hierarchical References*. Instances based on *InstanceDeclarations* shall always keep the same *BrowseName* as the *InstanceDeclaration* they are derived from. A special *Service* defined in IEC 62541-4 called TranslateBrowsePathsToNodeIds may be used to identify the instances based on the *InstanceDeclarations*. Using the simple Browse *Service* might not be sufficient since the uniqueness of the *BrowseName* is only required for *TypeDefinitionNodes* and *InstanceDeclarations*, not for other instances. Thus, “AI_BLK_1” may have another *Variable* with the *BrowseName* “SP”, although this one would not be derived from an *InstanceDeclaration* of the *TypeDefinitionNode*.

Instances derived from an *InstanceDeclaration* shall be of the same *TypeDefinitionNode* or a subtype of this *TypeDefinitionNode*.

A *TypeDefinitionNode* and its *InstanceDeclarations* shall always reside in the same *Server*. However, instances may point with their *HasTypeDefinition Reference* to a *TypeDefinitionNode* in a different *Server*.

4.6 Event Model

4.6.1 General

The Event Model defines a general purpose eventing system that can be used in many diverse vertical markets.

Events represent specific transient occurrences. System configuration changes and system errors are examples of *Events*. *Event Notifications* report the occurrence of an *Event*. *Events* defined in this document are not directly visible in the OPC UA *AddressSpace*. *Objects* and *Views* can be used to subscribe to *Events*. The *EventNotifier Attribute* of those *Nodes* identifies if the *Node* allows subscribing to *Events*. *Clients* subscribe to such *Nodes* to receive *Notifications* of *Event* occurrences.

Event Subscriptions use the Subscription/MonitoredItem *Services* defined in IEC 62541-4 to subscribe to the *Event Notifications* of a *Node*.

Any OPC UA *Server* that supports eventing shall expose at least one *Node* as *EventNotifier*. The *Server Object* defined in IEC 62541-5 is used for this purpose. *Events* generated by the *Server* are available via this *Server Object*. A *Server* is not expected to produce *Events* if the connection to the event source is down for some reason (i.e. the system is offline).

Events may also be exposed through other *Nodes* anywhere in the *AddressSpace*. These *Nodes* (identified via the *EventNotifier Attribute*) provide some subset of the *Events* generated by the *Server*. The position in the *AddressSpace* dictates what this subset will be. For example, a process area *Object* representing a functional area of the process would provide *Events* originating from that area of the process only. It should be noted that this is only an example and it is fully up to the *Server* to determine what *Events* should be provided by which *Node*.

4.6.2 EventTypes

Each *Event* is of a specific *EventType*. A *Server* may support many types. This document defines the *BaseEventType* that all other *EventTypes* derive from. It is expected that other companion specifications will define additional *EventTypes* deriving from the base types defined in this document.

The *EventTypes* supported by a *Server* are exposed in the *AddressSpace* of a *Server*. *EventTypes* are represented as *ObjectTypes* in the *AddressSpace* and do not have a special *NodeClass* associated to them. IEC 62541-5 defines how a *Server* exposes the *EventTypes* in detail.

EventTypes defined in this document are specified as abstract and therefore never instantiated in the *AddressSpace*. Event occurrences of those *EventTypes* are only exposed via a *Subscription*. *EventTypes* exist in the *AddressSpace* to allow *Clients* to discover the *EventType*. This information is used by a *Client* when establishing and working with *Event Subscriptions*. *EventTypes* defined by other parts of IEC 62541 or companion specifications as well as *Server* specific *EventTypes* may be defined as not abstract; therefore, instances of those *EventTypes* may be visible in the *AddressSpace* although *Events* of those *EventTypes* are also accessible via the *Event Notification* mechanisms.

Standard *EventTypes* are described in Clause 9. Their representation in the *AddressSpace* is specified in IEC 62541-5.

4.6.3 Event Categorization

Events can be categorized by creating new *EventTypes* which are subtypes of existing *EventTypes* but do not extend an existing type. They are used only to identify an event as being of the new *EventType*. For example, the *EventType* DeviceFailureEventType could be subtyped into TransmitterFailureEventType and ComputerFailureEventType. These new subtypes would not add new *Properties* or change the semantic inherited from the DeviceFailureEventType other than purely for categorization of the *Events*.

Event sources can also be organized into groups by using the *Event ReferenceTypes* described in 7.17 and 7.18. For example, a *Server* may define *Objects* in the *AddressSpace* representing *Events* related to physical devices, or *Event* areas of a plant or functionality contained in the *Server*. *Event References* would be used to indicate which *Event* sources represent physical devices and which ones represent some *Server*-based functionality. In addition, *References* can be used to group the physical devices or *Server*-based functionality into hierarchical *Event* areas. In some cases, an *Event* source may be categorized as being both a device and a *Server* function. In this case, two relationships would be established. Refer to the description of the *Event ReferenceTypes* for additional examples.

Clients can select a category or categories of *Events* by defining content filters that include terms specifying the *EventType* of the *Event* or a grouping of *Event* sources. The two mechanisms allow for a single *Event* to be categorized in multiple manners. A *Client* could obtain all *Events* related to a physical device or all failures of a particular device.

4.7 Methods

Methods are “lightweight” functions, whose scope is bounded by an owning (see Note) *Object*, similar to the methods of a class in object-oriented programming or an owning *ObjectType*, similar to static methods of a class. *Methods* are invoked by a *Client*, proceed to completion on the *Server* and return the result to the *Client*. The lifetime of the *Method’s* invocation instance begins when the *Client* calls the *Method* and ends when the result is returned.

NOTE The owning *Object* or *ObjectType* is specified in the service call when invoking the *Method*.

While *Methods* may affect the state of the owning *Object*, they have no explicit state of their own. In this sense, they are stateless. *Methods* can have a varying number of input arguments and return resultant arguments. Each *Method* is described by a *Node* of the *Method NodeClass*. This *Node* contains the metadata that identifies the *Method’s* arguments and describes its behaviour.

Methods are invoked by using the Call *Service* defined in IEC 62541-4.

Clients discover the *Methods* supported by a *Server* by browsing for the owning *Objects References* that identify their supported *Methods*.

4.8 Roles

4.8.1 Overview

A *Role* is a function assumed by a *Client* when it accesses a *Server*. *Roles* are used to separate authentication (determining who a *Client* is) from authorization (determining what the *Client* is allowed to do). By separating these tasks *Servers* can allow centralized services to manage user identities and credentials while the *Server* only manages the *Permissions* on its *Nodes* assigned to *Roles*.

The set of *Roles* supported by a *Server* are published as components of the *Roles Object* defined in IEC 62541-5. *Servers* should define a base set of *Roles* and allow configuration *Clients* to add system specific *Roles*.

When a *Session* is created, the *Server* shall determine what *Roles* are granted to that *Session*. This document defines standard mapping rules which *Servers* may support. *Servers* may also use vendor specific mapping rules in addition to or instead of the standard rules.

The standard mapping rules allow *Roles* to be granted based on:

- user identity;
- application identity;
- endpoint.

User identity mappings can be based on user names, user certificates or user groups. Well known groups include ‘AuthenticatedUser’ (any user with valid credentials) and ‘Anonymous’ (no user credentials provided).

Application identity mappings are based on the *ApplicationUri* specified in the *Client Certificate*. Application identity can only be enforced if the *Client* proves possession of a trusted *Certificate* by using it to create a *Secure Channel* or by providing a signature in *ActivateSession* (see IEC 62541-4).

Endpoint identity mappings are based on the URL used to connect to the *Server*. Endpoint identity can be used to restrict access to *Clients* running on particular networks.

IEC 62541-5 defines the *Objects*, *Methods* and *DataTypes* used to represent and manage these mapping rules in the *AddressSpace*.

4.8.2 Well-known Roles

All *Servers* should support the well-known *Roles* which are defined in Table 2. The *NodeIds* for the well-known *Roles* are defined in IEC 62541-6.

Table 2 – Well-known Roles

BrowseName	Suggested Permissions
Anonymous	The *Role* has very limited access for use when a *Session* has anonymous credentials.
AuthenticatedUser	The *Role* has limited access for use when a *Session* has valid non-anonymous credentials but has not been explicitly granted access to a *Role*.
Observer	The *Role* is allowed to browse, read live data, read historical data/events or subscribe to data/events.
Operator	The *Role* is allowed to browse, read live data, read historical data/events or subscribe to data/events. In addition, the *Session* is allowed to write some live data and call some *Methods*.
Engineer	The *Role* is allowed to browse, read/write configuration data, read historical data/events, call *Methods* or subscribe to data/events.
Supervisor	The *Role* is allowed to browse, read live data, read historical data/events, call *Methods* or subscribe to data/events.
ConfigureAdmin	The *Role* is allowed to change the non-security related configuration settings.
SecurityAdmin	The *Role* is allowed to change security related settings.

4.8.3 Evaluating Permissions with Roles

When a *Client* attempts to access a *Node,* the *Server* goes through the list of *Roles* granted to the *Session* and logically ORs the *Permissions* for the *Role* on the *Node. If there are no Node specific Permissions then* the default *Permissions* for the *Role* in the *DefaultRolePermissions Property* of the *NamespaceMetadata* for the namespace the *Node* belongs to are used (see IEC 62541-5). The resulting mask is the effective *Permissions*. If the bits corresponding to current operation are set, then the operation can proceed. If they are not set the *Server* returns *Bad_UserAccessDenied*.

Roles appear under the *Roles Object* in the *Server AddressSpace*. Each *Role* has mapping rules defined which appear as *Properties* of the *Role Object* (see IEC 62541-5). The examples shown in Table 3 illustrate how the standard mapping rules can be used to determine which *Roles* a *Session* has access to and, consequently, the *Permissions* that are granted to the *Session*.

Table 3 – Example Roles

Role	Mapping Rules	Description
Anonymous	Identities = Anonymous Applications = Endpoints =	An identity mapping rule that specifies the *Role* applies to anonymous users.
AuthenticatedUser	Identities = AuthenticatedUser Applications = Endpoints =	An identity mapping rule that specifies the *Role* applies to authenticated users.
Operator1	Identities = User with name 'Joe' Applications = urn:OperatorStation1 Endpoints =	An identity mapping rule that specifies specific users that have access to the *Role* with an application rule that restricts access to a single Client application.
Operator2	Identities = Users with name 'Joe' or 'Ann' Applications = urn:OperatorStation2 Endpoints =	An identity mapping rule that specifies specific users that have access to the *Role* with an application rule that restricts access to a single Client application.
Supervisor	Identities = User with name 'Root' Applications = Endpoints =	An identity mapping rule that specifies specific users that have access to the *Role*
Administrator	Identities = User with name 'Root' Applications = Endpoints = opc.tcp://127.0.0.1:48000	An identity mapping rule that specifies specific users that have access to the *Role* when they connect via a specific Endpoint.

The examples also make use of the *Nodes* defined in Table 4. The table specifies the value of the *RolePermissions Attribute* for each *Node*.

Table 4 – Example Nodes

Node	Role Permissions
Unit1.Measurement	AuthenticatedUser = Browse Operator1 = Browse, Read
Unit2.Measurement	AuthenticatedUser = Browse Operator2 = Browse, Read
SetPoint	AuthenticatedUser = Browse Operator1 and Operator2 = Browse, Read, Write Supervisor = Browse, Read
DisableDevice	AuthenticatedUser = Browse Operator1 and Operator2 = Browse, Read Administrator = Browse, Read, Write

When a *Client* creates a *Session* the *Roles* assigned to the *Session* depend on the rules defined for each *Role*. Table 5 lists the assigned *Roles* for different *Sessions* created with different *Users*, *Client* applications and endpoints.

Table 5 – Example Role assignment

User Provided by Client	Roles Assigned to Session
Anonymous	Anonymous
Sam	AuthenticatedUser
Joe using OperatorStation1 application.	AuthenticatedUser, Operator1
Joe using OperatorStation2 application.	AuthenticatedUser, Operator2
Joe using generic application.	AuthenticatedUser
Root using OperatorStation1 application.	AuthenticatedUser, Supervisor
Root using generic application and 127.0.0.1 endpoint.	AuthenticatedUser, Supervisor, Administrator
Root using generic application and another endpoint.	AuthenticatedUser, Supervisor

When a *Client* application accesses a *Node* the *RolePermissions* for the *Node* are compared to the *Roles* assigned to the *Session*. Any *Permission* available to at least one *Role* is granted to the *Client*. Table 6 provides a number of scenarios and examples and the resulting decision on access.

Table 6 – Examples of evaluating access

Use Case	Role Permissions
Anonymous user on localhost browses Unit1.Measurement *Node*.	Access denied because no rule defined for Anonymous users.
User ‘Sam’ using OperatorStation1 application browses Unit1.Measurement *Node*.	Allowed because AuthenticatedUser is granted Browse Permission.
User ‘Sam’ using OperatorStation2 application reads *Value* of Unit1.Measurement *Node*.	Access denied because AuthenticatedUser is not granted Read *Permission*.
User ‘Joe’ using OperatorStation1 application reads *Value* of Unit1.Measurement *Node*.	Allowed because Operator1 is granted Read *Permission*.
User ‘Joe’ using OperatorStation2 application reads *Value* of Unit1.Measurement *Node*.	Access denied because AuthenticatedUser and Operator2 are not granted Read *Permission*.
User ‘Joe’ using generic OPC UA application reads *Value* of Measurement *Node*.	Access denied because AuthenticatedUser is not granted Read *Permission*.
User ‘Joe’ using OperatorStation1 application writes *Value* of SetPoint *Node*.	Allowed because Operator1 is granted Write *Permission*.
User ‘Root’ using OperatorStation1 application writes the *Value* of SetPoint *Node*.	Denied because AuthenticatedUser and Supervisor are not granted Write *Permission*.
User ‘Joe’ using OperatorStation1 application writes *Value* of DisableDevice *Node*.	Access denied because AuthenticatedUser and Operator1 are not granted Write *Permission*.
User ‘Root’ using OperatorStation1 application writes the *Value* of DisableDevice *Node*.	Access denied because AuthenticatedUser and Supervisor are not granted Write *Permission*.
User ‘Root’ using endpoint 127.0.0.1 to write *Value* of DisableDevice *Node*.	Allowed because Administrator is granted Write *Permission*.

5 Standard NodeClasses

5.1 Overview

Clause 5 defines the *NodeClasses* used to define *Nodes* in the OPC UA *AddressSpace*. *NodeClasses* are derived from a common *Base NodeClass*. This *NodeClass* is defined first, followed by those used to organize the *AddressSpace* and then by the *NodeClasses* used to represent *Objects*.

The *NodeClasses* defined to represent *Objects* fall into three categories: those used to define instances, those used to define types for those instances and those used to define data types. Subclause 6.3 describes the rules for subtyping and 6.4 the rules for instantiation of the type definitions.

5.2 Base NodeClass

5.2.1 General

The OPC UA Address Space Model defines a *Base NodeClass* from which all other *NodeClasses* are derived. The derived *NodeClasses* represent the various components of the OPC UA Object Model (see 4.2). The *Attributes* of the *Base NodeClass* are specified in Table 7. There are no *References* specified for the *Base NodeClass*.

Table 7 – Base NodeClass

Name	Use	Data Type	Description
Attributes			
NodeId	M	NodeId	See 5.2.2
NodeClass	M	NodeClass	See 5.2.3
BrowseName	M	QualifiedName	See 5.2.4
DisplayName	M	LocalizedText	See 5.2.5
Description	O	LocalizedText	See 5.2.6
WriteMask	O	AttributeWriteMask	See 5.2.7
UserWriteMask	O	AttributeWriteMask	See 5.2.8
RolePermissions	O	RolePermissionType[]	See 5.2.9
UserRolePermissions	O	RolePermissionType[]	See 5.2.10
AccessRestrictions	O	AccessRestrictionsType	See 5.2.11
References			No *References* specified for this *NodeClass*

5.2.2 NodeId

Nodes are unambiguously identified using a constructed identifier called the *NodeId*. Some *Servers* may accept alternative *NodeIds* in addition to the canonical *NodeId* represented in this *Attribute*. A *Server* shall persist the *NodeId* of a *Node*, that is, it shall not generate new *NodeIds* when rebooting. The structure of the *NodeId* is defined in 8.2.

5.2.3 NodeClass

The *NodeClass Attribute* identifies the *NodeClass* of a *Node*. Its data type is defined in 8.30.

5.2.4 BrowseName

Nodes have a *BrowseName Attribute* that is used as a non-localized human-readable name when browsing the *AddressSpace* to create paths out of *BrowseNames*. The TranslateBrowsePathsToNodeIds *Service* defined in IEC 62541-4 can be used to follow a path constructed of *BrowseNames*.

A *BrowseName* should never be used to display the name of a *Node*. The *DisplayName* should be used instead for this purpose.

Unlike *NodeIds*, the *BrowseName* cannot be used to unambiguously identify a *Node*. Different *Nodes* may have the same *BrowseName*.

Subclause 8.3 defines the structure of the *BrowseName*. It contains a namespace and a string. The namespace is provided to make the *BrowseName* unique in some cases in the context of a *Node* (e.g. *Properties* of a *Node*) although not unique in the context of the *Server*. If different organizations define *BrowseNames* for *Properties*, the namespace of the *BrowseName* provided by the organization makes the *BrowseName* unique, although different organizations may use the same string having a slightly different meaning.

Servers may often choose to use the same namespace for the *NodeId* and the *BrowseName*. However, if they want to provide a standard *Property*, its *BrowseName* shall have the namespace of the standards body although the namespace of the *NodeId* reflects something else, for example the local *Server*.

It is recommended that standards bodies defining standard type definitions use their namespace for the *NodeId* of the *TypeDefinitionNode* as well as for the *BrowseName* of the *TypeDefinitionNode.*

The string-part of the *BrowseName* is case sensitive. That is, *Clients* shall consider them case sensitive. *Servers* are allowed to handle *BrowseNames* passed in *Service* requests as case insensitive. Examples are the *TranslateBrowsePathsToNodeIds Service* or *Event* filter.

5.2.5 DisplayName

The *DisplayName Attribute* contains the localized name of the *Node*. *Clients* should use this *Attribute* if they want to display the name of the *Node* to the user. They should not use the *BrowseName* for this purpose. The *Server* may maintain one or more localized representations for each *DisplayName*. *Clients* negotiate the locale to be returned when they open a session with the *Server*. Refer to IEC 62541-4 for a description of session establishment and locales. Subclause 8.5 defines the structure of the *DisplayName*. The string part of the *DisplayName* is restricted to 512 characters.

5.2.6 Description

The optional *Description Attribute* shall explain the meaning of the *Node* in a localized text using the same mechanisms for localization as described for the *DisplayName* in 5.2.5.

5.2.7 WriteMask

The optional *WriteMask Attribute* exposes the possibilities of a *Client* to write the *Attributes* of the *Node*. The *WriteMask Attribute* does not take any user access rights into account, that is, although an *Attribute* is writable this may be restricted to a certain user/user group.

If the OPC UA *Server* does not have the ability to get the *WriteMask* information for a specific *Attribute* from the underlying system, it should state that it is writable. If a write operation is called on the *Attribute*, the *Server* should transfer this request and return the corresponding *StatusCode* if such a request is rejected. *StatusCodes* are defined in IEC 62541-4.

The *AttributeWriteMask DataType* is defined in 8.60.

5.2.8 UserWriteMask

The optional *UserWriteMask Attribute* exposes the possibilities of a *Client* to write the *Attributes* of the *Node* taking user access rights into account. It uses the *AttributeWriteMask DataType* which is defined in 8.60.

The *UserWriteMask Attribute* can only further restrict the *WriteMask Attribute,* when it is set to not writable in the general case that applies for every user.

Clients cannot assume an *Attribute* can be written based on the *UserWriteMask Attribute*. It is possible that the *Server* may return an access denied error due to some *Server* specific change which was not reflected in the state of this *Attribute* at the time the *Client* accessed it.

5.2.9 RolePermissions

The optional *RolePermissions Attribute* specifies the *Permissions* that apply to a *Node* for all *Roles* which have access to the *Node*. The value of the *Attribute* is an array of *RolePermissionType Structures* (see Table 8).

Table 8 – RolePermissionType

Name	Type	Description
RolePermissionType	Structure	Specifies the *Permissions* for a *Role*
roleId	NodeId	The *NodeId* of the *Role Object*.
permissions	PermissionType	A mask specifying which *Permissions* are available to the *Role*.

Servers may allow administrators to write to the *RolePermissions Attribute*.

If not specified, the value of *DefaultRolePermissions Property* from the *NamespaceMetadata Object* associated with the *Node* shall be used instead. If the *NamespaceMetadata Object* does not define the *Property* or does not exist, then the *Server* should not publish any information about how it manages *Permissions*.

If a *Server* supports *Permissions* for a particular *Namespace* it shall add the *DefaultRolePermissions Property* to the *NamespaceMetadata Object* for that *Namespace* (see Figure 8). If a particular *Node* in the *Namespace* needs to override the default values, the *Server* adds the *RolePermissions Attribute* to the *Node.* The *DefaultRolePermissions Property* and *RolePermissions Attribute* shall only be readable by administrators. If a *Server* allows the *Permissions* to be changed these values shall be writable. If the *Server* allows the *Permissions* to be overridden for a particular *Node* but does not currently have any *Node Permissions* configured, then the value of the *Attribute* shall be an empty array. If the administrator wishes to remove overridden *Permissions*, an empty array shall be written to this *Attribute*. *Servers* shall prevent *Permissions* from being changed in such a way as to render the *Server* inoperable.

If a *Server* publishes information about the *Roles* for a *Namespace* assigned to the current *Session,* it shall add the *DefaultUserRolePermissions Property* to the *NamespaceMetadata Object* for that *Namespace.* The value of this *Property* shall be a readonly list of *Permissions* for each *Role* assigned to the current *Session*. If a particular *Node* in the *Namespace* overrides the default *RolePermissions* the *Server* shall also override the *DefaultUserRolePermissions* by adding the *UserRolePermissions Attribute* to the *Node.* If the *Server* allows the *Permissions* to be overridden for a particular *Node* but does not currently have any *Node Permissions* configured, then the Server shall return the value of the *DefaultUserRolePermissions Property* for the *Node Namespace.*

If a Server implements a vendor specific *Role Permission* model for a *Namespace*, it shall not add the *DefaultRolePermissions* or *DefaultUserRolePermissions Properties* to the *Namespace Metadata Object*.

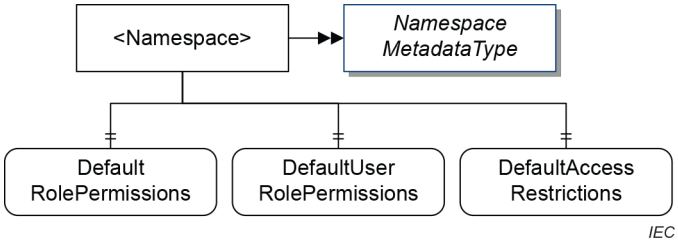


Figure 8 – Permissions in the Address Space

5.2.10 UserRolePermissions

The optional *UserRolePermissions Attribute* specifies the *Permissions* that apply to a *Node* for all *Roles* granted to current *Session*. The value of the *Attribute* is an array of *RolePermissionType Structures* (see Table 8).

Clients may determine their effective *Permissions* by performing a logical OR of *Permissions* for each *Role* in the array.

The value of this *Attribute* is derived from the rules used by the *Server* to map *Sessions* to *Roles*. This mapping may be vendor specific or it may use the standard *Role* model defined in 4.8.

This *Attribute* shall not be writable.

If not specified, the value of *DefaultUserRolePermissions Property* from the *Namespace Metadata Object* associated with the *Node* is used instead. If the *NamespaceMetadata Object* does not define the *Property* or does not exist, then the *Server* does not publish any information about *Roles* mapped to the current *Session*.

5.2.11 AccessRestrictions

The optional *AccessRestrictions Attribute* specifies the *AccessRestrictions* that apply to a *Node*. Its data type is defined in 8.56. If a *Server* supports *AccessRestrictions* for a particular *Namespace* it adds the *DefaultAccessRestrictions Property* to the *NamespaceMetadata Object* for that *Namespace* (see Figure 8). If a particular *Node* in the *Namespace* needs to override the default value the *Server* adds the *AccessRestrictions Attribute* to the *Node*.

If a *Server* implements a vendor specific access restriction model for a *Namespace*, it does not add the *DefaultAccessRestrictions Property* to the *NamespaceMetadata Object*.

5.3 ReferenceType NodeClass

5.3.1 General

References are defined as instances of *ReferenceType Nodes*. *ReferenceType Nodes* are visible in the *AddressSpace* and are defined using the *ReferenceType NodeClass* as specified in Table 9. In contrast, a *Reference* is an inherent part of a *Node* and no *NodeClass* is used to represent *References*.

This document defines a set of *ReferenceTypes* provided as an inherent part of the OPC UA Address Space Model. These *ReferenceTypes* are defined in Clause 7 and their representation in the *AddressSpace* is defined in IEC 62541-5. *Servers* may also define *ReferenceTypes*. In addition, IEC 62541-4 defines *NodeManagement Services* that allow *Clients* to add *ReferenceTypes* to the *AddressSpace*.

Table 9 – ReferenceType NodeClass

Name	Use	Data Type	Description
Attributes			
Base NodeClass Attributes	M	--	Inherited from the *Base NodeClass*. See 5.2.
IsAbstract	M	Boolean	A boolean *Attribute* with the following values: TRUE it is an abstract *ReferenceType*, i.e. no *Reference* of this type shall exist, only of its subtypes. FALSE it is not an abstract *ReferenceType*, i.e. *References* of this type can exist.
Symmetric	M	Boolean	A boolean *Attribute* with the following values: TRUE the meaning of the *ReferenceType* is the same as seen from both the *SourceNode* and the *TargetNode*. FALSE the meaning of the *ReferenceType* as seen from the *TargetNode* is the inverse of that as seen from the *SourceNode*.
InverseName	O	LocalizedText	The inverse name of the *Reference*, which is the meaning of the *ReferenceType* as seen from the *TargetNode*.
References			
HasProperty	0..*		Used to identify the Properties (see 5.3.3.2).
HasSubtype	0..*		Used to identify subtypes (see 5.3.3.3).
Standard Properties			
NodeVersion	O	String	The *NodeVersion Property* is used to indicate the version of a *Node*. The *NodeVersion Property* is updated each time a *Reference* is added to or deleted from the *Node* the *Property* belongs to. *Attribute* value changes do not cause the *NodeVersion* to change. *Clients* may read the *NodeVersion Property* or subscribe to it to determine when the structure of a *Node* has changed.

5.3.2 Attributes

The *ReferenceType NodeClass* inherits the base *Attributes* from the *Base NodeClass* defined in 5.2. The inherited *BrowseName Attribute* is used to specify the meaning of the *ReferenceType* as seen from the *SourceNode*. For example, the *ReferenceType* with the *BrowseName* "Contains" is used in *References* that specify that the *SourceNode* contains the *TargetNode*. The inherited *DisplayName Attribute* contains a translation of the *BrowseName*.

The *BrowseName* of a *ReferenceType* shall be unique in a *Server*. It is not allowed that two different *ReferenceTypes* have the same *BrowseName*.

The *IsAbstract Attribute* indicates if the *ReferenceType* is abstract. Abstract *ReferenceTypes* cannot be instantiated and are used only for organizational reasons, for example to specify some general semantics or constraints that its subtypes inherit.

The *Symmetric Attribute* is used to indicate whether or not the meaning of the *ReferenceType* is the same for both the *SourceNode* and *TargetNode*.

If a *ReferenceType* is symmetric, the *InverseName Attribute* shall be omitted. Examples of symmetric *ReferenceTypes* are "Connects To" and "Communicates With". Both imply the same semantic coming from the *SourceNode* or the *TargetNode*. Therefore both directions are considered to be forward *References*.

If the *ReferenceType* is non-symmetric and not abstract, the *InverseName Attribute* shall be set. The *InverseName Attribute* specifies the meaning of the *ReferenceType* as seen from the *TargetNode*. Examples of non-symmetric *ReferenceTypes* include “Contains” and “Contained In”, and “Receives From” and “Sends To”.

References that use the *InverseName*, such as “Contained In” *References*, are referred to as inverse *References*.

Figure 9 provides examples of symmetric and non-symmetric *References* and the use of the *BrowseName* and the *InverseName*.

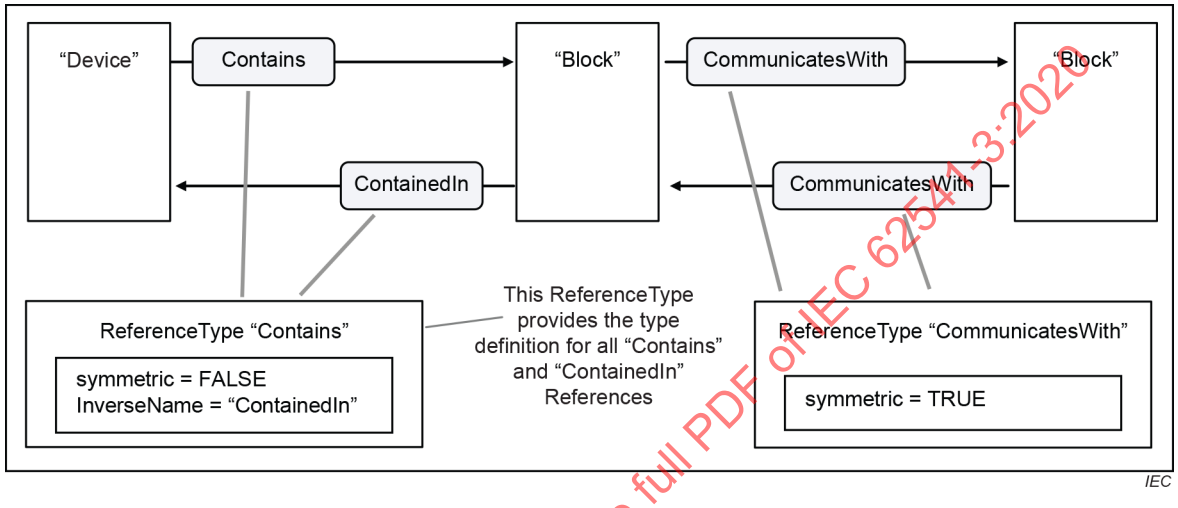


Figure 9 – Symmetric and Non-Symmetric References

It might not always be possible for *Servers* to instantiate both forward and inverse *References* for non-symmetric *ReferenceTypes* as shown in Figure 9. When they do, the *References* are referred to as *bidirectional*. Although not required, it is recommended that all *Hierarchical References* be instantiated as bidirectional to ensure browse connectivity. A bidirectional *Reference* is modelled as two separate *References*.

As an example of a *unidirectional Reference*, it is often the case that a signal sink knows its signal source, but this signal source does not know its signal sink. The signal sink would have a “Sourced By” *Reference* to the signal source, without the signal source having the corresponding “Sourced To” inverse *References* to its signal sinks.

The *DisplayName* and the *InverseName* are the only standardized places to indicate the semantic of a *ReferenceType*. There may be more complex semantics associated with a *ReferenceType* than can be expressed in those *Attributes* (e.g. the semantic of *HasSubtype*). This document does not specify how this semantic should be exposed. However, the *Description Attribute* can be used for this purpose. This document provides a semantic for the *ReferenceTypes* specified in Clause 7.

A *ReferenceType* can have constraints restricting its use. For example, it can specify that starting from *Node* A and only following *References* of this *ReferenceType* or one of its subtypes, it shall never be able to return to A, that is, a “No Loop” constraint.

This document does not specify how those constraints could or should be made available in the *AddressSpace*. Nevertheless, for the standard *ReferenceTypes*, some constraints are specified in Clause 7. This document does not restrict the kind of constraints valid for a *ReferenceType*. It can, for example, also affect an *ObjectType*. The restriction that a *ReferenceType* can only be used by relating *Nodes* of some *NodeClasses* with a defined cardinality is a special constraint of a *ReferenceType*.

5.3.3 References

5.3.3.1 General

HasSubtype References and *HasProperty References* are the only *ReferenceTypes* that may be used with *ReferenceType Nodes* as *SourceNode*. *ReferenceType Nodes* shall not be the *SourceNode* of other types of *References*.

5.3.3.2 HasProperty References

HasProperty References are used to identify the *Properties* of a *ReferenceType* and shall only refer to *Nodes* of the *Variable NodeClass*.

The *Property NodeVersion* is used to indicate the version of the *ReferenceType*.

There are no additional *Properties* defined for *ReferenceTypes* in this document. Additional parts of IEC 62541 may define additional *Properties* for *ReferenceTypes*.

5.3.3.3 HasSubtype References

HasSubtype References are used to define subtypes of *ReferenceTypes*. It is not required to provide the *HasSubtype Reference* for the supertype, but the subtype shall provide the inverse *Reference* to its supertype. The following rules for subtyping apply.

a) The semantic of a *ReferenceType* (e.g. “spans a hierarchy”) is inherited to its subtypes and can be refined there (e.g. “spans a special hierarchy”). The *DisplayName*, and also the *InverseName* for non-symmetric *ReferenceTypes*, reflect the specialization.

b) If a *ReferenceType* specifies some constraints (e.g. “allow no loops”) this is inherited and can only be refined (e.g. inheriting “no loops” could be refined as “shall be a tree – only one parent”) but not lowered (e.g. “allow loops”).

c) The constraints concerning which *NodeClasses* can be referenced are also inherited and can only be further restricted. That is, if a *ReferenceType* “A” is not allowed to relate an *Object* with an *ObjectType*, this is also true for its subtypes.

d) A *ReferenceType* shall have exactly one supertype, except for the *References ReferenceType* defined in 7.2 as the root type of the *ReferenceType* hierarchy. The *ReferenceType* hierarchy does not support multiple inheritances.

5.4 View NodeClass

Underlying systems are often large and *Clients* often have an interest in only a specific subset of the data. They do not need, or want, to be burdened with viewing *Nodes* in the *AddressSpace* for which they have no interest.

To address this problem, this document defines the concept of a *View*. Each *View* defines a subset of the *Nodes* in the *AddressSpace*. The entire *AddressSpace* is the default *View*. Each *Node* in a *View* may contain only a subset of its *References*, as defined by the creator of the *View*. The *View Node* acts as the root for the *Nodes* in the *View*. *Views* are defined using the *View NodeClass*, which is specified in Table 10.

All *Nodes* contained in a *View* shall be accessible starting from the *View Node* when browsing in the context of the *View*. It is not expected that all containing *Nodes* can be browsed directly from the *View Node* but rather browsed from other *Nodes* contained in the *View*.

A *View Node* may not only be used as additional entry point into the *AddressSpace* but as a construct to organize the *AddressSpace* and thus as the only entry point into a subset of the *AddressSpace*. Therefore *Clients* shall not ignore *View Nodes* when exposing the *AddressSpace*. Simple *Clients* that do not deal with *Views* for filtering purposes can, for example, handle a *View Node* like an *Object* of type *FolderType* (see 5.5.3).

Table 10 – View NodeClass

<table>
<tr><th>Name</th><th>Use</th><th>Data Type</th><th>Description</th></tr>
<tr><td>Attributes</td><td></td><td></td><td></td></tr>
<tr><td>Base NodeClass Attributes</td><td>M</td><td>--</td><td>Inherited from the Base NodeClass. See 5.2.</td></tr>
<tr><td>ContainsNoLoops</td><td>M</td><td>Boolean</td><td>If set to “true” this Attribute indicates that by following the References in the context of the View there are no loops, i.e. starting from a Node “A” contained in the View and following the forward References in the context of the View Node “A” will not be reached again. It does not specify that there is only one path starting from the View Node to reach a Node contained in the View.

If set to “false” this Attribute indicates that following References in the context of the View may lead to loops.</td></tr>
<tr><td>EventNotifier</td><td>M</td><td>Byte</td><td>The EventNotifier Attribute is used to indicate if the Node can be used to subscribe to Events or to read / write historic Events.

The EventNotifier is an 8-bit unsigned integer with the structure defined in the following table.
<table>
<tr><th>Field</th><th>Bit</th><th>Description</th></tr>
<tr><td>SubscribeTo Events</td><td>0</td><td>Indicates if it can be used to subscribe to Events
(0 means cannot be used to subscribe to Events, 1 means can be used to subscribe to Events)</td></tr>
<tr><td>Reserved</td><td>1</td><td>Reserved for future use. Shall always be zero.</td></tr>
<tr><td>HistoryRead</td><td>2</td><td>Indicates if the history of the Events is readable
(0 means not readable, 1 means readable)</td></tr>
<tr><td>HistoryWrite</td><td>3</td><td>Indicates if the history of the Events is writable
(0 means not writable, 1 means writable)</td></tr>
<tr><td>Reserved</td><td>4:7</td><td>Reserved for future use. Shall always be zero</td></tr>
</table>
The second two bits also indicate if the history of the Events is available via the OPC UA Server.</td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td>References</td><td></td><td></td><td></td></tr>
<tr><td>HierarchicalReferences</td><td>0..*</td><td></td><td>Top level Nodes in a View are referenced by Hierarchical References (see 7.3).</td></tr>
<tr><td>HasProperty</td><td>0..*</td><td></td><td>HasProperty References identify the Properties of the View.</td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td>Standard Properties</td><td></td><td></td><td></td></tr>
<tr><td>NodeVersion</td><td>O</td><td>String</td><td>The NodeVersion Property is used to indicate the version of a Node.

The NodeVersion Property is updated each time a Reference is added to or deleted from the Node the Property belongs to. Attribute value changes do not cause the NodeVersion to change. Clients may read the NodeVersion Property or subscribe to it to determine when the structure of a Node has changed.</td></tr>
<tr><td>ViewVersion</td><td>O</td><td>UInt32</td><td>The version number for the View. When Nodes are added to or removed from a View, the value of the ViewVersion Property is updated. Clients may detect changes to the composition of a View using this Property. The value of the ViewVersion shall always be greater than 0.</td></tr>
</table>

The *View NodeClass* inherits the base *Attributes* from the *Base NodeClass* defined in 5.2. It also defines two additional *Attributes*.

The mandatory *ContainsNoLoops Attribute* is set to false if the *Server* is not able to identify if the *View* contains loops or not.

The mandatory *EventNotifier Attribute* identifies if the *View* can be used to subscribe to *Events* that either occur in the content of the *View* or as *ModelChangeEvents* (see 9.32) of the content of the *View* or to read / write the history of the *Events*. A *View* that supports *Events* shall provide all *Events* that occur in any *Object* used as *EventNotifier* that is part of the content of the *View*. In addition, it shall provide all *ModelChangeEvents* that occur in the context of the *View*.

To avoid recursion, i.e. getting all *Events* of the *Server*, the *Server Object* defined in IEC 62541-5 shall never be part of any *View* since it provides all *Events* of the *Server*.

Views are defined by the *Server*. The browsing and querying *Services* defined in IEC 62541-4 expect the *NodeId* of a *View Node* to provide these *Services* in the context of the *View*.

HasProperty References are used to identify the *Properties* of a *View*. The *Property NodeVersion* is used to indicate the version of the *View Node*. The *ViewVersion Property* indicates the version of the content of the *View*. In contrast to the *NodeVersion*, the *ViewVersion Property* is updated even if *Nodes* not directly referenced by the *View Node* are added to or deleted from the *View*. This *Property* is optional because it might not be possible for *Servers* to detect changes in the *View* contents. *Servers* may also generate a *ModelChangeEvent,* described in 9.32, if *Nodes* are added to or deleted from the *View*. There are no additional *Properties* defined for *Views* in this document. Additional parts of IEC 62541 may define additional *Properties* for *Views*.

Views can be the *SourceNode* of any *Hierarchical Reference*. They shall not be the *SourceNode* of any non-hierarchical *Reference*.

5.5 Objects

5.5.1 Object NodeClass

Objects are used to represent systems, system components, real-world objects and software objects. *Objects* are defined using the *Object NodeClass*, specified in Table 11.

Table 11 – Object NodeClass

Name	Use	Data Type	Description
Attributes			
Base NodeClass Attributes	M	--	Inherited from the *Base NodeClass*. See 5.2.
EventNotifier	M	EventNotifierType	The *EventNotifier Attribute* is used to indicate if the *Node* can be used to subscribe to *Events* or the read / write historic *Events*. The *EventNotifierType* is defined in 8.59.
References			
HasComponent	0..*		*HasComponent References* identify the *DataVariables*, the *Methods* and *Objects* contained in the *Object*.
HasProperty	0..*		*HasProperty References* identify the *Properties* of the *Object*.
HasModellingRule	0..1		*Objects* can point to at most one *ModellingRule Object* using a *HasModellingRule Reference* (see 6.4.4 for details on *ModellingRules*).
HasTypeDefinition	1		The *HasTypeDefinition Reference* points to the type definition of the *Object*. Each *Object* shall have exactly one type definition and therefore be the *SourceNode* of exactly one *HasTypeDefinition Reference* pointing to an *ObjectType*. See 4.5 for a description of type definitions.
HasEventSource	0..*		The *HasEventSource Reference* points to event sources of the *Object*. *References* of this type can only be used for *Objects* having their "SubscribeToEvents" bit set in the *EventNotifier Attribute*. See 7.17 for details.
HasNotifier	0..*		The *HasNotifier Reference* points to notifiers of the *Object*. *References* of this type can only be used for *Objects* having their "SubscribeToEvents" bit set in the *EventNotifier Attribute*. See 7.18 for details.
Organizes	0..*		This *Reference* should be used only for *Objects* of the *ObjectType FolderType* (see 5.5.3).
<other References>	0..*		*Objects* may contain other *References*.
Standard Properties			
NodeVersion	O	String	The *NodeVersion Property* is used to indicate the version of a *Node*. The *NodeVersion Property* is updated each time a *Reference* is added to or deleted from the *Node* the *Property* belongs to. *Attribute* value changes do not cause the *NodeVersion* to change. *Clients* may read the *NodeVersion Property* or subscribe to it to determine when the structure of a *Node* has changed.
Icon	O	Image	The *Icon Property* provides an image that can be used by *Clients* when displaying the *Node*. It is expected that the *Icon Property* contains a relatively small image.
NamingRule	O	NamingRuleType	The *NamingRule* Property defines the *NamingRule* of a *ModellingRule* (see 6.4.4.2 for details). This *Property* shall only be used for *Objects* of the type *ModellingRuleType* defined in 6.4.4.

The *Object NodeClass* inherits the base *Attributes* from the *Base NodeClass* defined in 5.2.

The mandatory *EventNotifier Attribute* identifies whether the *Object* can be used to subscribe to *Events* or to read and write the history of the *Events*.

The *Object NodeClass* uses the *HasComponent Reference* to define the *DataVariables*, *Objects* and *Methods* of an *Object*.

It uses the *HasProperty Reference* to define the *Properties* of an *Object*. The *Property NodeVersion* is used to indicate the version of the *Object*. The *Property Icon* provides an icon of the *Object*. The *Property NamingRule* defines the *NamingRule* of a *ModellingRule* and shall only be applied to *Objects* of type *ModellingRuleType*. There are no additional *Properties* defined for *Objects* in this document. Additional parts of IEC 62541 may define additional *Properties* for *Objects*.

To specify its *ModellingRule*, an *Object* can use at most one *HasModellingRule Reference* pointing to a *ModellingRule Object*. *ModellingRules* are defined in 6.4.4.

HasNotifier and *HasEventSource References* are used to provide information about eventing and can only be applied to *Objects* used as event notifiers. Details are defined in 7.17 and 7.18.

The *HasTypeDefinition Reference* points to the *ObjectType* used as type definition of the *Object*.

Objects may use any additional *References* to define relationships to other *Nodes*. No restrictions are placed on the types of *References* used or on the *NodeClasses* of the *Nodes* that may be referenced. However, restrictions may be defined by the *ReferenceType* excluding its use for *Objects*. Standard *ReferenceTypes* are described in Clause 7.

If the *Object* is used as an *InstanceDeclaration* (see 4.5) then all *Nodes* referenced with forward *Hierarchical References* direction shall have unique *BrowseNames* in the context of this *Object*.

If the *Object* is created based on an *InstanceDeclaration* then it shall have the same *BrowseName* as its *InstanceDeclaration*.

5.5.2 ObjectType NodeClass

ObjectTypes provide definitions for *Objects*. *ObjectTypes* are defined using the *ObjectType NodeClass*, which is specified in Table 12.

Table 12 – ObjectType NodeClass

Name	Use	Data Type	Description
Attributes			
Base NodeClass Attributes	M	--	Inherited from the *Base NodeClass*. See 5.2.
IsAbstract	M	Boolean	A boolean *Attribute* with the following values: TRUE it is an abstract *ObjectType*, i.e. no *Objects* of this type shall exist, only *Objects* of its subtypes. FALSE it is not an abstract *ObjectType*, i.e. *Objects* of this type can exist.
References			
HasComponent	0..*		*HasComponent References* identify the *DataVariables*, the *Methods*, and *Objects* contained in the *ObjectType*. If and how the referenced *Nodes* are instantiated when an *Object* of this type is instantiated, is specified in 6.4.
HasProperty	0..*		*HasProperty References* identify the *Properties* of the *ObjectType*. If and how the *Properties* are instantiated when an *Object* of this type is instantiated, is specified in 6.4.
HasSubtype	0..*		*HasSubtype References* identify *ObjectTypes* that are subtypes of this type. The inverse *SubtypeOf Reference* identifies the parent type of this type.
GeneratesEvent	0..*		*GeneratesEvent References* identify the type of *Events* instances of this type may generate.
<other References>	0..*		*ObjectTypes* may contain other *References* that can be instantiated by *Objects* defined by this *ObjectType*.
Standard Properties			
NodeVersion	O	String	The *NodeVersion Property* is used to indicate the version of a *Node*. The *NodeVersion Property* is updated each time a *Reference* is added to or deleted from the *Node* the *Property* belongs to. *Attribute* value changes do not cause the *NodeVersion* to change. *Clients* may read the *NodeVersion Property* or subscribe to it to determine when the structure of a *Node* has changed.
Icon	O	Image	The *Icon Property* provides an image that can be used by *Clients* when displaying the *Node*. It is expected that the *Icon Property* contains a relatively small image.

The *ObjectType NodeClass* inherits the base *Attributes* from the *Base NodeClass* defined in 5.2. The additional *IsAbstract Attribute* indicates if the *ObjectType* is abstract or not.

The *ObjectType NodeClass* uses the *HasComponent References* to define the *DataVariables*, *Objects*, and *Methods* for it.

The *HasProperty Reference* is used to identify the *Properties*. The *Property NodeVersion* is used to indicate the version of the *ObjectType*. The *Property Icon* provides an icon of the *ObjectType*. There are no additional *Properties* defined for *ObjectTypes* in this document. Additional parts of IEC 62541 may define additional *Properties* for *ObjectTypes*.

HasSubtype References are used to subtype *ObjectTypes*. *ObjectType* subtypes inherit the general semantics from the parent type. The general rules for subtyping apply as defined in Clause 6. It is not required to provide the *HasSubtype Reference* for the supertype, but it is required that the subtype provides the inverse *Reference* to its supertype.

GeneratesEvent References identify the type of *Events* that instances of the *ObjectType* may generate. These *Objects* may be the source of an *Event* of the specified type or one of its subtypes. *Servers* should make *GeneratesEvent References* bidirectional *References*. However, it is allowed to be unidirectional when the *Server* is not able to expose the inverse direction pointing from the *EventType* to each *ObjectType* supporting the *EventType.* Note that the *EventNotifier Attribute* of an *Object* and the *GeneratesEvent References* of its *ObjectType* are completely unrelated. *Objects* that can generate *Events* might not be used as *Objects* to which *Clients* subscribe to get the corresponding *Event* notifications.

GeneratesEvent References are optional, i.e. *Objects* may generate *Events* of an *EventType* that is not exposed by its *ObjectType*.

ObjectTypes may use any additional *References* to define relationships to other *Nodes*. No restrictions are placed on the types of *References* used or on the *NodeClasses* of the *Nodes* that may be referenced. However, restrictions may be defined by the *ReferenceType* excluding its use for *ObjectTypes*. Standard *ReferenceTypes* are described in Clause 7.

All *Nodes* referenced with forward *Hierarchical References* shall have unique *BrowseNames* in the context of an *ObjectType* (see 4.5).

5.5.3 Standard ObjectType FolderType

The *ObjectType FolderType* is formally defined in IEC 62541-5. Its purpose is to provide *Objects* that have no other semantic than organizing of the *AddressSpace*. A special *ReferenceType* is introduced for those *Folder Objects*, the *Organizes ReferenceType*. The *SourceNode* of such a *Reference* should always be a *View* or an *Object* of the *ObjectType FolderType*; the *TargetNode* can be of any *NodeClass*. *Organizes References* can be used in any combination with *HasChild References* (*HasComponent*, *HasProperty*, etc.; see 7.5) and do not prevent loops. Thus, they can be used to span multiple hierarchies.

5.5.4 Client-side creation of Objects of an ObjectType

Objects are always based on an *ObjectType*, i.e. they have a *HasTypeDefinition Reference* pointing to its *ObjectType*.

Clients can create *Objects* using the *AddNodes Service* defined in IEC 62541-4. The *Service* requires specifying the *TypeDefinitionNode* of the *Object*. An *Object* created by the *AddNodes Service* contains all components defined by its *ObjectType* dependent on the *ModellingRules* specified for the components. However, the *Server* may add additional components and *References* to the *Object* and its components that are not defined by the *ObjectType*. This behaviour is *Server* dependent. The *ObjectType* only specifies the minimum set of components that shall exist for each *Object* of an *ObjectType*.

In addition to the *AddNodes Service ObjectTypes* may have a special *Method* with the *BrowseName* "*Create*". This *Method* is used to create an *Object* of this *ObjectType*. This *Method* may be useful for the creation of *Objects* where the semantic of the creation should differ from the default behaviour expected in the context of the *AddNodes Service*. For example, the values should directly differ from the default values or additional *Objects* should be added, etc. The input and output arguments of this *Method* depend on the *ObjectType*; the only commonality is the *BrowseName* identifying that this *Method* will create an *Object* based on the *ObjectType*. *Servers* should not provide a *Method* on an *ObjectType* with the *BrowseName* "*Create*" for any other purpose than creating *Objects* of the *ObjectType*.

5.6 Variables

5.6.1 General

Two types of *Variables* are defined, *Properties* and *DataVariables*. Although they differ in the way they are used as described in 4.4 and have different constraints described in the remainder of 5.6, they use the same *NodeClass* described in 5.6.2. The constraints of

Properties based on this *NodeClass* are defined in 5.6.3, the constraints of *DataVariables* in 5.6.4.

5.6.2 Variable NodeClass

Variables are used to represent values which may be simple or complex. *Variables* are defined by *VariableTypes*, as specified in 5.6.5.

Variables are always defined as *Properties* or *DataVariables* of other *Nodes* in the *AddressSpace*. They are never defined by themselves. A *Variable* is always part of at least one other *Node*, but may be related to any number of other *Nodes*. *Variables* are defined using the *Variable NodeClass*, specified in Table 13.

Table 13 – Variable NodeClass

Name	Use	Data Type	Description
Attributes			
Base NodeClass Attributes	M	--	Inherited from the *Base NodeClass*. See 5.2.
Value	M	Defined by the *DataType Attribute*	The most recent value of the *Variable* that the *Server* has. Its data type is defined by the *DataType Attribute*. It is the only *Attribute* that does not have a data type associated with it. This allows all *Variables* to have a value defined by the same *Value Attribute*.
DataType	M	NodeId	*NodeId* of the *DataType* definition for the *Value Attribute*. Standard *DataTypes* are defined in Clause 8.
ValueRank	M	Int32	This *Attribute* indicates whether the *Value Attribute* of the *Variable* is an array and how many dimensions the array has. It may have the following values: $n > 1$: the Value is an array with the specified number of dimensions. OneDimension (1): The value is an array with one dimension. OneOrMoreDimensions (0): The value is an array with one or more dimensions. Scalar (−1): The value is not an array. Any (−2): The value can be a scalar or an array with any number of dimensions. ScalarOrOneDimension (−3): The value can be a scalar or a one dimensional array. All DataTypes are considered to be scalar, even if they have array-like semantics like ByteString and String.
ArrayDimensions	O	UInt32[]	This *Attribute* specifies the maximum supported length of each dimension. If the maximum is unknown the value shall be 0. The number of elements shall be equal to the value of the *ValueRank Attribute*. This *Attribute* shall be null if *ValueRank* ≤ 0. For example, if a *Variable* is defined by the following C array: Int32 myArray[346]; then this *Variable's DataType* would point to an Int32 and the *Variable's ValueRank* has the value 1 and the *ArrayDimensions* is an array with one entry having the value 346. The maximum number of elements of an array transferred on the wire is 2147483647 (max Int32).

Name	Use	Data Type	Description
AccessLevel	M	AccessLevelType	The *AccessLevel Attribute* is used to indicate how the *Value* of a *Variable* can be accessed (read/write) and if it contains current and/or historic data. The *AccessLevel* does not take any user access rights into account, i.e. although the *Variable* is writable this may be restricted to a certain user / user group. The AccessLevelType is defined in 8.57.
UserAccessLevel	M	AccessLevelType	The *UserAccessLevel Attribute* is used to indicate how the *Value* of a *Variable* can be accessed (read/write) and if it contains current or historic data taking user access rights into account. The AccessLevelType is defined in 8.57.
MinimumSamplingInterval	O	Duration	The *MinimumSamplingInterval Attribute* indicates how "current" the *Value* of the *Variable* will be kept. It specifies (in milliseconds) how fast the *Server* can reasonably sample the value for changes (see IEC 62541-4 for a detailed description of sampling interval). A *MinimumSamplingInterval* of 0 indicates that the *Server* is to monitor the item continuously. A *MinimumSamplingInterval* of -1 means indeterminate.
Historizing	M	Boolean	The *Historizing Attribute* indicates whether the *Server* is actively collecting data for the history of the *Variable*. This differs from the *AccessLevel Attribute* which identifies if the *Variable* has any historical data. A value of TRUE indicates that the *Server* is actively collecting data. A value of FALSE indicates the *Server* is not actively collecting data. Default value is FALSE.
AccessLevelEx	O	AccessLevelExType	The *AccessLevelEx Attribute* is used to indicate how the *Value* of a *Variable* can be accessed (read/write), if it contains current and/or historic data and its atomicity. The *AccessLevelEx* does not take any user access rights into account, i.e. although the *Variable* is writable this may be restricted to a certain user / user group. The *AccessLevelEx* is an extended version of the AccessLevel attribute and as such contains the 8 bits of the AccessLevel attribute as the first 8 bits. The *AccessLevelEx* is a 32-bit unsigned integer with the structure defined in 8.58. If this Attribute is not provided the information provided by these additional Fields is unknown.
References			
HasModellingRule	0..1		*Variables* can point to at most one *ModellingRule Object* using a *HasModellingRule Reference* (see 6.4.4 for details on *ModellingRules*).
HasProperty	0..*		*HasProperty References* are used to identify the *Properties* of a *DataVariable*. *Properties* are not allowed to be the *SourceNode* of *HasProperty References*.
HasComponent	0..*		*HasComponent References* are used by complex *DataVariables* to identify their composed *DataVariables*. *Properties* are not allowed to use this *Reference*.
HasTypeDefinition	1		The *HasTypeDefinition Reference* points to the type definition of the *Variable*. Each *Variable* shall have exactly one type definition and therefore be the *SourceNode* of exactly one *HasTypeDefinition Reference* pointing to a *VariableType*. See 4.5 for a description of type definitions.
<other References>	0..*		*Data Variables* may be the *SourceNode* of any other *References*. *Properties* may only be the *SourceNode* of any non-hierarchical *Reference*.

Name	Use	Data Type	Description
Standard Properties			
NodeVersion	O	String	The *NodeVersion Property* is used to indicate the version of a *DataVariable*. It does not apply to *Properties*. The *NodeVersion Property* is updated each time a *Reference* is added to or deleted from the *Node* the *Property* belongs to. *Attribute* value changes except for the *DataType Attribute* do not cause the *NodeVersion* to change. *Clients* may read the *NodeVersion Property* or subscribe to it to determine when the structure of a *Node* has changed. Although the relationship of a *Variable* to its *DataType* is not modelled using *References*, changes to the *DataType Attribute* of a *Variable* lead to an update of the *NodeVersion Property*.
LocalTime	O	TimeZone DataType	The *LocalTime Property* is only used for *DataVariables*. It does not apply to *Properties*. This *Property* is a structure containing the Offset and the DaylightSavingInOffset flag. The Offset specifies the time difference (in minutes) between the SourceTimestamp (UTC) associated with the value and the time at the location in which the value was obtained. The SourceTimestamp is defined in IEC 62541-4. If DaylightSavingInOffset is TRUE, then Standard/Daylight savings time (DST) at the originating location is in effect and Offset includes the DST correction. If FALSE then the Offset does not include DST correction and DST may or may not have been in effect.
AllowNulls	O	Boolean	The *AllowNulls Property* is only used for *DataVariables*. It does not apply to *Properties*. This *Property* specifies if a null value is allowed for the *Value Attribute* of the *DataVariable*. If it is set to true, the *Server* may return null values and accept writing of null values. If it is set to false, the *Server* shall never return a null value and shall reject any request writing a null value. If this *Property* is not provided, it is *Server*-specific if null values are allowed or not.
ValueAsText	O	Localized Text	It is used for *DataVariables* with a finite set of *LocalizedTexts* associated with its value. For example any *DataVariables* having an *Enumeration DataType*. This optional *Property* provides the localized text representation of the value. It can be used by *Clients* only interested in displaying the text to subscribe to the *Property* instead of the value attribute.
MaxStringLength	O	UInt32	Only used for *DataVariables* having a *String DataType*. This optional *Property* indicates the maximum number of bytes supported by the *DataVariable*.
MaxCharacters	O	UInt32	Only used for *DataVariables* having a *String DataType*. This optional *Property* indicates the maximum number of Unicode characters supported by the *DataVariable*.
MaxByteStringLength	O	UInt32	Only used for *DataVariables* having a *ByteString DataType*. This optional *Property* indicates the maximum number of bytes supported by the *DataVariable*.

Name	Use	Data Type	Description
MaxArrayLength	O	UInt32	Only used for *DataVariables* having its *ValueRank Attribute* not set to scalar. This optional *Property* indicates the maximum length of an array supported by the *DataVariable*. In a multidimensional array it indicates the overall length. For example, a three-dimensional array of 2 x 3 x 10 has the array length of 60. NOTE In order to expose the length of an array of bytes do not use the *DataType ByteString* but an array of the *DataType Byte*. In that case the *MaxArrayLength* applies.
EngineeringUnits	O	EU Information	Only used for *DataVariables* having a *Number DataType*. This optional *Property* indicates the engineering units for the value of the *DataVariable* (e.g. hertz or seconds). Details about the *Property* and what engineering units should be used are defined in IEC 62541-8. The *DataType EUInformation* is also defined in IEC 62541-8.

The *Variable NodeClass* inherits the base *Attributes* from the *Base NodeClass* defined in 5.2.

The *Variable NodeClass* also defines a set of *Attributes* that describe the *Variable's* Runtime value. The *Value Attribute* represents the *Variable* value. The *DataType*, *ValueRank* and *ArrayDimensions Attributes* provide the capability to describe simple and complex values.

The *AccessLevel Attribute* indicates the accessibility of the *Value* of a *Variable* not taking user access rights into account. If the OPC UA *Server* does not have the ability to get the *AccessLevel* information from the underlying system then it should state that it is readable and writable. If a read or write operation is called on the *Variable* then the *Server* should transfer this request and return the corresponding *StatusCode* even if such a request is rejected. *StatusCodes* are defined in IEC 62541-4.

The *SemanticChange* flag of the *AccessLevel Attribute* is used for *Properties* that may change and define semantic aspects of the parent *Node*. For example, the EngineeringUnits *Property* describes the semantic of a *DataVariable*, whereas the Icon *Property* does not. In this example, if the EngineeringUnits *Property* may change while the *Server* is running, the *SemanticChange* flag shall be set for it.

Servers that support *Event* subscriptions shall generate a *SemanticChangeEvent* whenever a *Property* with *SemanticChange* flag set changes.

If a *Variable* having a *Property* with *SemanticChange* flag set is used in a *Subscription* and the *Property* value changes, then the *SemanticsChanged* bit of the *StatusCode* shall be set as defined in IEC 62541-4. *Clients* subscribing to a *Variable* should look at the *StatusCode* to identify if the semantic has changed and retrieve the relevant *Properties* before processing the value returned from the *Subscription*.

The *UserAccessLevel Attribute* indicates the accessibility of the *Value* of a *Variable* taking user access rights into account. If the OPC UA *Server* does not have the ability to get any user access rights related information from the underlying system then it should use the same bit mask as used in the *AccessLevel Attribute*. The *UserAccessLevel Attribute* can restrict the accessibility indicated by the *AccessLevel Attribute*, but not exceed it. *Clients* should not assume access rights based on the *UserAccessLevel Attribute*. For example it is possible that the *Server* returns an error due to some server specific change which was not reflected in the state of this *Attribute* at the time the *Client* accessed the *Variable*.

The *MinimumSamplingInterval Attribute* specifies how fast the *Server* can reasonably sample the *value* for changes. The accuracy of this value (the ability of the *Server* to attain "best case" performance) can be greatly affected by system load and other factors.

The *Historizing Attribute* indicates whether the *Server* is actively collecting data for the history of the *Variable*. See IEC 62541-11 for details on historizing *Variables*.

Clients may read or write *Variable* values, or monitor them for value changes, as specified in IEC 62541-4. IEC 62541-8 defines additional rules when using the *Services* for automation data.

To specify its *ModellingRule*, a *Variable* can use at most one *HasModellingRule Reference* pointing to a *ModellingRule Object*. *ModellingRules* are defined in 6.4.4.

If the *Variable* is created based on an *InstanceDeclaration* (see 4.5) it shall have the same *BrowseName* as its *InstanceDeclaration*.

The other *References* are described separately for *Properties* and *DataVariables* in the remainder of 5.6

5.6.3 Properties

Properties are used to define the characteristics of *Nodes*. *Properties* are defined using the *Variable NodeClass*, specified in Table 13. However, they restrict their use.

Properties are the leaf of any hierarchy; therefore they shall not be the *SourceNode* of any *Hierarchical References*. This includes the *HasComponent* or *HasProperty Reference*, that is, *Properties* do not contain *Properties* and cannot expose their complex structure. However, they may be the *SourceNode* of any non-hierarchical *References*.

The *HasTypeDefinition Reference* points to the *VariableType* of the *Property*. Since *Properties* are uniquely identified by their *BrowseName*, all *Properties* shall point to the *PropertyType* defined in IEC 62541-5.

Properties shall always be defined in the context of another *Node* and shall be the *TargetNode* of at least one *HasProperty Reference*. To distinguish them from *DataVariables*, they shall not be the *TargetNode* of any *HasComponent Reference*. Thus, a *HasProperty Reference* pointing to a *Variable Node* defines this *Node* as a *Property*.

The *BrowseName* of a *Property* is always unique in the context of a *Node*. It is not permitted for a *Node* to refer to two *Variables* using *HasProperty References* having the same *BrowseName*.

5.6.4 DataVariable

DataVariables represent the content of an *Object*. *DataVariables* are defined using the *Variable NodeClass*, specified in Table 13.

DataVariables identify their *Properties* using *HasProperty References*. Complex *DataVariables* use *HasComponent References* to expose their component *DataVariables*.

The *Property NodeVersion* indicates the version of the *DataVariable*.

The *Property LocalTime* indicates the difference between the SourceTimestamp of the value and the standard time at the location in which the value was obtained.

The *Property AllowNulls* indicates if null values are allowed for the *Value Attribute*.

The *Property ValueAsText* provides a localized text representation for enumeration values.

The *Property MaxStringLength* indicates the maximum number of bytes of a *String* value. If a *Server* does not impose a maximum number of bytes or is not able to determine the maximum number of bytes this *Property* shall not be provided. If this *Property* is provided then the *MaxCharacters Property* shall not be provided.

The *Property MaxCharacters* indicates the maximum number of Unicode characters of a string value. If a *Server* does not impose a maximum number of Unicode characters or is not able to determine the maximum number of Unicode characters this *Property* shall not be provided. If this *Property* is provided then the *MaxStringLength Property* shall not be provided.

The *Property MaxByteStringLength* indicates the maximum number of bytes of a *ByteString* value. If a *Server* does not impose a maximum number of bytes or is not able to determine the maximum number of bytes this *Property* shall not be provided.

The *Property MaxArrayLength* indicates the maximum allowed array length of the value.

The *Property EngineeringUnits* indicates the engineering units of the value. There are no additional *Properties* defined for *DataVariables* in this part of this document. Additional parts of IEC 62541 may define additional *Properties* for *DataVariables*. IEC 62541-8 defines a set of *Properties* that can be used for *DataVariables*.

DataVariables may use additional *References* to define relationships to other *Nodes*. No restrictions are placed on the types of *References* used or on the *NodeClasses* of the *Nodes* that may be referenced. However, restrictions may be defined by the *ReferenceType* excluding its use for *DataVariables*. Standard *ReferenceTypes* are described in Clause 7.

A *DataVariable* is intended to be defined in the context of an *Object*. However, complex *DataVariables* may expose other *DataVariables* and *ObjectTypes* and complex *VariableTypes* may also contain *DataVariables*. Therefore each *DataVariable* shall be the *TargetNode* of at least one *HasComponent Reference* coming from an *Object*, an *ObjectType*, a *DataVariable* or a *VariableType*. *DataVariables* shall not be the *TargetNode* of any *HasProperty References*. Therefore, a *HasComponent Reference* pointing to a *Variable Node* identifies it as a *DataVariable*.

The *HasTypeDefinition Reference* points to the *VariableType* used as type definition of the *DataVariable*.

If the *DataVariable* is used as *InstanceDeclaration* (see 4.5) all *Nodes* referenced with forward *Hierarchical References* shall have unique *BrowseNames* in the context of this *DataVariable*.

5.6.5 VariableType NodeClass

VariableTypes are used to provide type definitions for *Variables*. *VariableTypes* are defined using the *VariableType NodeClass*, as specified in Table 14.

Table 14 – VariableType NodeClass

Name	Use	Data Type	Description
Attributes			
Base NodeClass Attributes	M	--	Inherited from the *Base NodeClass*. See 5.2
Value	O	Defined by the *DataType attribute*	The default *Value* for instances of this type.
DataType	M	NodeId	*NodeId* of the data type definition for instances of this type.
ValueRank	M	Int32	This *Attribute* indicates whether the *Value Attribute* of the *VariableType* is an array and how many dimensions the array has. It may have the following values: $n > 1$: the Value is an array with the specified number of dimensions. OneDimension (1): The value is an array with one dimension. OneOrMoreDimensions (0): The value is an array with one or more dimensions. Scalar (−1): The value is not an array. Any (−2): The value can be a scalar or an array with any number of dimensions. ScalarOrOneDimension (−3): The value can be a scalar or a one dimensional array. NOTE All DataTypes are considered to be scalar, even if they have array-like semantics like ByteString and String.
ArrayDimensions	O	UInt32[]	This *Attribute* specifies the length of each dimension for an array value. The Attribute specifies the maximum supported length of each dimension. If the maximum is unknown the value is 0. The number of elements shall be equal to the value of the *ValueRank Attribute*. This *Attribute* shall be null if *ValueRank* ≤ 0. For example, if a *VariableType* is defined by the following C array: Int32 myArray[346]; then this *VariableType's DataType* would point to an Int32, the *VariableType's ValueRank* has the value 1 and the *ArrayDimensions* is an array with one entry having the value 346.
IsAbstract	M	Boolean	A boolean *Attribute* with the following values: TRUE it is an abstract *VariableType*, i.e. no *Variable* of this type shall exist, only of its subtypes. FALSE it is not an abstract *VariableType*, i.e. *Variables* of this type can exist.
References			
HasProperty	0..*		*HasProperty References* are used to identify the *Properties* of the *VariableType*. The referenced *Nodes* may be instantiated by the instances of this type, depending on the *ModellingRules* defined in 6.4.4.
HasComponent	0..*		*HasComponent References* are used for complex *VariableTypes* to identify their containing *DataVariables*. Complex *VariableTypes* can only be used for *DataVariables*. The referenced *Nodes* may be instantiated by the instances of this type, depending on the *ModellingRules* defined in 6.4.4.
HasSubtype	0..*		*HasSubtype References* identify *VariableTypes* that are subtypes of this type. The inverse *subtype of Reference* identifies the parent type of this type.
GeneratesEvent	0..*		*GeneratesEvent References* identify the type of *Events* instances of this type may generate.

Name	Use	Data Type	Description
<other References>	0..*		*VariableTypes* may contain other *References* that can be instantiated by *Variables* defined by this *VariableType*. *ModellingRules* are defined in 6.4.4.
Standard Properties			
NodeVersion	O	String	The *NodeVersion Property* is used to indicate the version of a *Node*. The *NodeVersion Property* is updated each time a *Reference* is added to or deleted from the *Node* the *Property* belongs to. *Attribute* value changes except for the *DataType Attribute* do not cause the *NodeVersion* to change. *Clients* may read the *NodeVersion Property* or subscribe to it to determine when the structure of a *Node* has changed. Although the relationship of a *VariableType* to its *DataType* is not modelled using *References*, changes to the *DataType Attribute* of a *VariableType* lead to an update of the *NodeVersion Property*.

The *VariableType NodeClass* inherits the base *Attributes* from the *Base NodeClass* defined in 5.2. The *VariableType NodeClass* also defines a set of *Attributes* that describe the default or initial value of its instance *Variables*. The *Value Attribute* represents the default value. The *DataType*, *ValueRank* and *ArrayDimensions Attributes* provide the capability to describe simple and complex values. The *IsAbstract Attribute* defines if the type can be directly instantiated.

The *VariableType NodeClass* uses *HasProperty References* to define the *Properties* and *HasComponent References* to define *DataVariables*. Whether they are instantiated depends on the *ModellingRules* defined in 6.4.4.

The *Property NodeVersion* indicates the version of the *VariableType*. There are no additional *Properties* defined for *VariableTypes* in this document. Additional parts of IEC 62541 may define additional *Properties* for *VariableTypes*. IEC 62541-8 defines a set of *Properties* that can be used for *VariableTypes*.

HasSubtype References are used to subtype *VariableTypes*. *VariableType* subtypes inherit the general semantics from the parent type. The general rules for subtyping are defined in Clause 6. It is not required to provide the *HasSubtype Reference* for the supertype, but it is required that the subtype provides the inverse *Reference* to its supertype.

GeneratesEvent References identify that *Variables* of the *VariableType* may be the source of an *Event* of the specified *EventType* or one of its subtypes. *Servers* should make *GeneratesEvent References* bidirectional *References*. However, it is allowed to be unidirectional when the *Server* is not able to expose the inverse direction pointing from the *EventType* to each *VariableType* supporting the *EventType*.

GeneratesEvent References are optional, i.e. *Variables* may generate *Events* of an *EventType* that is not exposed by its *VariableType*.

VariableTypes may use any additional *References* to define relationships to other *Nodes*. No restrictions are placed on the types of *References* used or on the *NodeClasses* of the *Nodes* that may be referenced. However, restrictions may be defined by the *ReferenceType* excluding its use for *VariableTypes*. Standard *ReferenceTypes* are described in Clause 7.

All *Nodes* referenced with forward *Hierarchical References* shall have unique *BrowseNames* in the context of the *VariableType* (see 4.5).

5.6.6 Client-side creation of Variables of an VariableType

Variables are always based on a *VariableType*, i.e. they have a *HasTypeDefinition Reference* pointing to its *VariableType*.

Clients can create *Variables* using the *AddNodes Service* defined in IEC 62541-4. The *Service* requires specifying the *TypeDefinitionNode* of the *Variable*. A *Variable* created by the *AddNodes Service* contains all components defined by its *VariableType* dependent on the *ModellingRules* specified for the components. However, the *Server* may add additional components and *References* to the *Variable* and its components that are not defined by the *VariableType*. This behaviour is *Server* dependent. The *VariableType* only specifies the minimum set of components that shall exist for each *Variable* of a *VariableType*.

5.7 Method NodeClass

Methods define callable functions. *Methods* are invoked using the *Call Service* defined in IEC 62541-4. Method invocations are not represented in the *AddressSpace*. Method invocations always run to completion and always return responses when complete. *Methods* are defined using the *Method NodeClass*, specified in Table 15.

Table 15 – Method NodeClass

Name	Use	Data Type	Description
Attributes			
Base NodeClass Attributes	M	--	Inherited from the *Base NodeClass*. See 5.2.
Executable	M	Boolean	The *Executable Attribute* indicates if the *Method* is currently executable ("False" means not executable, "True" means executable). The *Executable Attribute* does not take any user access rights into account, i.e. although the *Method* is executable this may be restricted to a certain user / user group.
UserExecutable	M	Boolean	The *UserExecutable Attribute* indicates if the *Method* is currently executable taking user access rights into account ("False" means not executable, "True" means executable).
References			
HasProperty	0..*		*HasProperty References* identify the *Properties* for the *Method*.
HasModellingRule	0..1		*Methods* can point to at most one *ModellingRule Object* using a *HasModellingRule Reference* (see 6.4.4 for details on *ModellingRules*).
GeneratesEvent	0..*		*GeneratesEvent References* identify the type of *Events* that will be generated whenever the *Method* is called.
AlwaysGeneratesEvent	0..*		*AlwaysGeneratesEvent References* identify the type of *Events* that shall be generated whenever the *Method* is called.
<other References>	0..*		*Methods* may contain other *References*.
Standard Properties			
NodeVersion	O	String	The *NodeVersion Property* is used to indicate the version of a *Node*. The *NodeVersion Property* is updated each time a *Reference* is added to or deleted from the *Node* the *Property* belongs to. *Attribute* value changes do not cause the *NodeVersion* to change. *Clients* may read the *NodeVersion Property* or subscribe to it to determine when the structure of a *Node* has changed.
InputArguments	O	Argument[]	The *InputArguments Property* is used to specify the arguments that shall be used by a *Client* when calling the *Method*.
OutputArguments	O	Argument[]	The *OutputArguments Property* specifies the result returned from the *Method* call.

The *Method NodeClass* inherits the base *Attributes* from the *Base NodeClass* defined in 5.2. The *Method NodeClass* defines no additional *Attributes*.

The *Executable Attribute* indicates whether the *Method* is executable, not taking user access rights into account. If the OPC UA *Server* cannot get the *Executable* information from the underlying system, it should state that it is executable. If a *Method* is called then the *Server* should transfer this request and return the corresponding *StatusCode* even if such a request is rejected. *StatusCodes* are defined in IEC 62541-4.

The *UserExecutable Attribute* indicates whether the *Method* is executable, taking user access rights into account. If the OPC UA *Server* cannot get any user rights related information from the underlying system, it should use the same value as used in the *Executable Attribute*. The *UserExecutable Attribute* can be set to "False", even if the *Executable Attribute* is set to "True", but it shall be set to "False" if the *Executable Attribute* is set to "False". *Clients* cannot assume a *Method* can be executed based on the *UserExecutable Attribute*. It is possible that

the *Server* may return an access denied error due to some *Server* specific change which was not reflected in the state of this *Attribute* at the time the *Client* accessed it.

Properties may be defined for *Methods* using *HasProperty References*. The *Properties InputArguments* and *OutputArguments* specify the input arguments and output arguments of the *Method*. Both contain an array of the *DataType Argument* as specified in 8.6. An empty array or a *Property* that is not provided indicates that there are no input arguments or output arguments for the *Method*.

The *Property NodeVersion* indicates the version of the *Method*. There are no additional *Properties* defined for *Methods* in this document. Additional parts of IEC 62541 may define additional *Properties* for *Methods*.

To specify its *ModellingRule*, a *Method* can use at most one *HasModellingRule Reference* pointing to a *ModellingRule Object*. *ModellingRules* are defined in 6.4.4.

GeneratesEvent References identify that *Methods* may generate an *Event* of the specified *EventType* or one of its subtypes for every call of the *Method*. A *Server* may generate one *Event* for each referenced *EventType* when a *Method* is successfully called.

AlwaysGeneratesEvent References identify that *Methods* will generate an *Event* of the specified *EventType* or one of its subtypes for every call of the *Method*. A *Server* shall always generate one *Event* for each referenced *EventType* when a *Method* is successfully called.

Servers should make *GeneratesEvent References* bidirectional *References*. However, it is allowed to be unidirectional when the *Server* is not able to expose the inverse direction pointing from the *EventType* to each *Method* generating the *EventType*.

GeneratesEvent References are optional, i.e. the call of a *Method* may produce *Events* of an *EventType* that is not referenced with a *GeneratesEvent Reference* by the *Method*.

Methods may use additional *References* to define relationships to other *Nodes*. No restrictions are placed on the types of *References* used or on the *NodeClasses* of the *Nodes* that may be referenced. However, restrictions may be defined by the *ReferenceType* excluding its use for *Methods*. Standard *ReferenceTypes* are described in Clause 7.

A *Method* shall always be the *TargetNode* of at least one *HasComponent Reference*. The *SourceNode* of these *HasComponent References* shall be an *Object* or an *ObjectType*. If a *Method* is called then the *NodeId* of one of those *Nodes* shall be put into the Call *Service* defined in IEC 62541-4 as parameter to detect the context of the *Method* operation.

If the *Method* is used as *InstanceDeclaration* (see 4.5) all *Nodes* referenced with forward *Hierarchical References* shall have unique *BrowseNames* in the context of this *Method*.

5.8 DataTypes

5.8.1 DataType Model

The DataType Model is used to define simple and structured data types. Data types are used to describe the structure of the *Value Attribute* of *Variables* and their *VariableTypes*. Therefore each *Variable* and *VariableType* is pointing with its *DataType Attribute* to a *Node* of the *DataType NodeClass* as shown in Figure 10.

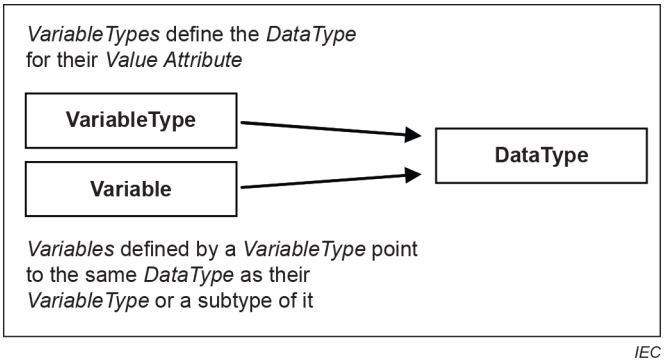


Figure 10 – Variables, VariableTypes and their DataTypes

In many cases, the *NodeId* of the *DataType Node* – the *DataTypeId* – will be well-known to *Clients* and *Servers*. Clause 8 defines *DataTypes* and IEC 62541-6 defines their *DataTypeIds*. In addition, other organizations may define *DataTypes* that are well-known in the industry. Well-known *DataTypeIds* provide for commonality across OPC UA *Servers* and allow *Clients* to interpret values without having to read the type description from the *Server*. Therefore, *Servers* may use well-known *DataTypeIds* without representing the corresponding *DataType Nodes* in their *AddressSpaces*.

In other cases, *DataTypes* and their corresponding *DataTypeIds* may be vendor-defined. *Servers* should attempt to expose the *DataType Nodes* and the information about the structure of those *DataTypes* for *Clients* to read, although this information might not always be available to the *Server*.

Figure 11 illustrates the *Nodes* used in the *AddressSpace* to describe the structure of a *DataType*. The *DataType* points to an *Object* of type *DataTypeEncodingType*. Each *DataType* can have several *DataTypeEncoding*, for example “Default”, “UA Binary” and “XML” encoding. Services in IEC 62541-4 allow *Clients* to request an encoding or choosing the “Default” encoding. Each *DataTypeEncoding* is used by exactly one *DataType*, that is, it is not permitted for two *DataTypes* to point to the same *DataTypeEncoding*.

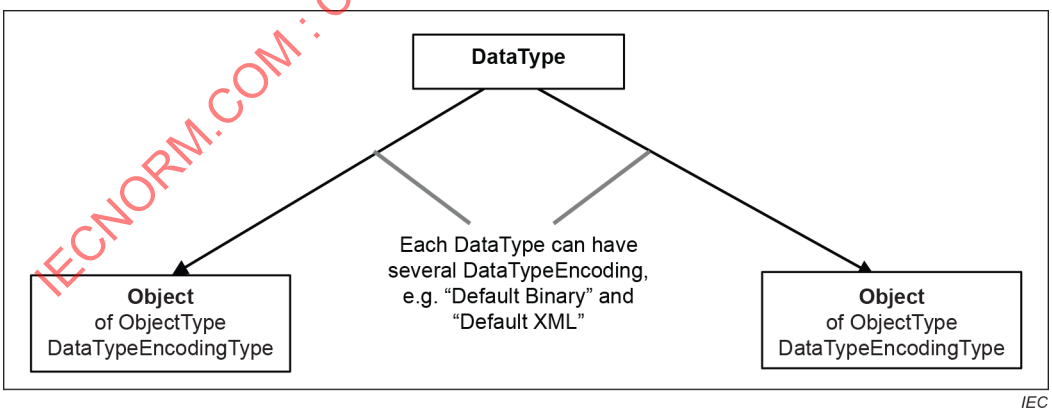


Figure 11 – DataType Model

Since the *NodeId* of the *DataTypeEncoding* will be used in some Mappings to identify the *DataType* and it’s encoding as defined in IEC 62541-6, those *NodeIds* may also be well-known for well-known *DataTypeIds*.

5.8.2 Encoding rules for different kinds of DataTypes

Different kinds of *DataTypes* are handled differently regarding their encoding and according to whether this encoding is represented in the *AddressSpace*.

Built-in DataTypes are a fixed set of *DataTypes* (see IEC 62541-6 for a complete list of *Built-in* DataTypes). They have no encodings visible in the *AddressSpace* since the encoding should be known to all OPC UA products. Examples of *Built-in DataTypes* are *Int32* (see 8.26) and *Double* (see 8.12).

Simple DataTypes are subtypes of the *Built-in DataTypes*. They are handled on the wire like the *Built-in DataType*, i.e. they cannot be distinguished on the wire from their *Built-in* supertypes. Since they are handled like *Built-in DataTypes* regarding the encoding they cannot have encodings defined in the *AddressSpace*. *Clients* can read the *DataType Attribute* of a *Variable* or *VariableType* to identify the *Simple DataType* of the *Value Attribute*. An example of a *Simple DataType* is *Duration*. It is handled on the wire as a *Double* but the Client can read the *DataType Attribute* and thus interpret the value as defined by *Duration* (see 8.13).

Structured DataTypes are *DataTypes* that represent structured data and are not defined as *Built-in DataTypes*. *Structured DataTypes* inherit directly or indirectly from the *DataType Structure* defined in 8.33. *Structured DataTypes* may have several encodings and the encodings are exposed in the *AddressSpace*. How the encoding of *Structured DataTypes* is handled on the wire is defined in IEC 62541-6. The encoding of the *Structured DataType* is transmitted with each value, thus *Clients* are aware of the *DataType* without reading the *DataType Attribute*. The encoding has to be transmitted so the Client is able to interpret the data. An example of a *Structured DataType* is *Argument* (see 8.6).

Enumeration DataTypes are *DataTypes* that represent discrete sets of named values. Enumerations are always encoded as Int32 on the wire as defined in IEC 62541-6. Enumeration *DataTypes* inherit directly or indirectly from the *DataType Enumeration* defined in 8.14. Enumerations have no encodings exposed in the *AddressSpace*. To expose the human-readable representation of an enumerated value the *DataType Node* may have the *EnumStrings Property* that contains an array of *LocalizedText*. The Integer representation of the enumeration value points to a position within that array. *EnumValues Property* can be used instead of the *EnumStrings* to support integer representation of enumerations that are not zero-based or have gaps. It contains an array of a *Structured DataType* containing the integer representation as well as the human-readable representation. An example of an enumeration *DataType* containing a sparse list of Integers is *NodeClass* which is defined in 8.30.

In addition to the *DataTypes* described above, abstract *DataTypes* are also supported, which do not have any encodings and cannot be exchanged on the wire. *Variables* and *VariableTypes* use abstract *DataTypes* to indicate that their *Value* may be any one of the subtypes of the abstract *DataType*. An example of an abstract *DataType* is Integer which is defined in 8.24.

5.8.3 DataType NodeClass

The *DataType NodeClass* describes the syntax of a *Variable Value*. *DataTypes* are defined using the *DataType NodeClass*, as specified in Table 16.

Table 16 – DataType NodeClass

Name	Use	Data Type	Description
Attributes			
Base NodeClass Attributes	M	--	Inherited from the *Base NodeClass*. See 5.2.
IsAbstract	M	Boolean	A boolean *Attribute* with the following values: TRUE it is an abstract *DataType*. FALSE it is not an abstract *DataType*.
DataTypeDefinition	O	DataTypeDefinition	The *DataTypeDefinition Attribute* is used to provide the metadata and encoding information for custom *DataTypes*. The abstract *DataTypeDefinition DataType* is defined in 8.48. Structure and Union DataTypes The *Attribute* is mandatory for *DataTypes* derived from *Structure* and *Union*. For such *DataTypes*, the *Attribute* contains a structure of the *DataType StructureDefinition*. The *StructureDefinition DataType* is defined in 8.49. It is a subtype of *DataTypeDefinition*. Enumeration and OptionSet DataTypes The *Attribute* is mandatory for *DataTypes* derived from *Enumeration, OptionSet* and subtypes of *UInteger* representing an *OptionSet*. For such *DataTypes*, the *Attribute* contains a structure of the *DataType EnumDefinition*. The *EnumDefinition DataType* is defined in 8.50. It is a subtype of *DataTypeDefinition*.
References			
HasProperty	0..*		*HasProperty References* identify the *Properties* for the *DataType*.
HasSubtype	0..*		*HasSubtype References* may be used to span a data type hierarchy.
HasEncoding	0..*		*HasEncoding References* identify the encodings of the *DataType* represented as *Objects* of type *DataTypeEncodingType*. Only concrete *Structured DataTypes* may use *HasEncoding References*. Abstract, *Built-in, Enumeration,* and *Simple DataTypes* are not allowed to be the *SourceNode* of a *HasEncoding Reference*. Each concrete *Structured DataType* shall point to at least one *DataTypeEncoding Object* with the *BrowseName* “Default Binary” or “Default XML” having the *NamespaceIndex* 0. The *BrowseName* of the *DataTypeEncoding Objects* shall be unique in the context of a *DataType*, i.e. a *DataType* shall not point to two *DataTypeEncodings* having the same *BrowseName*.
Standard Properties			
NodeVersion	O	String	The *NodeVersion Property* is used to indicate the version of a *Node*. The *NodeVersion Property* is updated each time a *Reference* is added to or deleted from the *Node* the *Property* belongs to. *Attribute* value changes do not cause the *NodeVersion* to change. Clients may read the *NodeVersion Property* or subscribe to it to determine when the structure of a *Node* has changed. *Clients* shall not use the content for programmatic purposes except for equality comparisions.

Name	Use	Data Type	Description
EnumStrings	O	LocalizedText[]	The *EnumStrings Property* only applies for *Enumeration DataTypes*. It shall not be applied for other *DataTypes*. If the *EnumValues Property* is provided, the *EnumStrings Property* shall not be provided. Each entry of the array of *LocalizedText* in this *Property* represents the human-readable representation of an enumerated value. The Integer representation of the enumeration value points to a position of the array.
EnumValues	O	EnumValueType[]	The *EnumValues Property* only applies for *Enumeration DataTypes*. It shall not be applied for other *DataTypes*. If the *EnumStrings Property* is provided, the *EnumValues Property* shall not be provided. Using the *EnumValues Property* it is possible to represent Enumerations with integers that are not zero-based or have gaps (e.g. 1, 2, 4, 8, and 16). Each entry of the array of *EnumValueType* in this *Property* represents one enumeration value with its integer notation, human-readable representation and help information.
OptionSetValues	O	LocalizedText[]	The *OptionSetValues Property* only applies for *OptionSet DataTypes* and *UInteger DataTypes*. An *OptionSet DataType* is used to represent a bit mask and the *OptionSetValues Property* contains the human-readable representation for each bit of the bit mask. The *OptionSetValues Property* provides an array of *LocalizedText* containing the human-readable representation for each bit.

The *DataType NodeClass* inherits the base *Attributes* from the *Base NodeClass* defined in 5.2. The *IsAbstract Attribute* specifies if the *DataType* is abstract or not. Abstract *DataTypes* can be used in the *AddressSpace*, i.e. *Variables* and *VariableTypes* can point with their *DataType Attribute* to an abstract *DataType*. However, concrete values can never be of an abstract *DataType* and shall always be of a concrete subtype of the abstract *DataType*.

HasProperty References are used to identify the *Properties* of a *DataType*. The *Property NodeVersion* is used to indicate the version of the *DataType*. The *Property EnumStrings* contains human-readable representations of enumeration values and is only applied to *Enumeration DataTypes*. Instead of the *EnumStrings Property* an *Enumeration DataType* can also use the *EnumValues Property* to represent *Enumerations* with integer values that are not zero-based or containing gaps. There are no additional *Properties* defined for *DataTypes* in this document. Additional parts of IEC 62541 may define additional *Properties* for *DataTypes*.

HasSubtype References may be used to expose a data type hierarchy in the *AddressSpace*. The semantic of subtyping is only defined to the point that a Server may provide instances of the subtype instead of the DataType. *Clients* should not make any assumptions about any other semantic with that information. For example, it might not be possible to cast a value of one data type to its base data type. *Servers* need not provide *HasSubtype References*, even if their *DataTypes* span a type hierarchy. Some restrictions apply for subtyping enumeration DataTypes as defined in 8.14.

HasEncoding References point from the *DataType* to its *DataTypeEncodings*. Each concrete *Structured DataType* can point to many *DataTypeEncodings*, but each *DataTypeEncoding* shall belong to one *DataType*, that is, it is not permitted for two *DataType Nodes* to point to the same *DataTypeEncoding Object* using *HasEncoding References*.

An abstract *DataType* is not the *SourceNode* of a *HasEncoding Reference*. The *DataTypeEncoding* of an abstract *DataType* is provided by its concrete subtypes.

DataType Nodes shall not be the *SourceNode* of other types of *References*. However, they may be the *TargetNode* of other *References*.

5.8.4 DataTypeEncoding and encoding information

If a *DataType Node* is exposed in the *AddressSpace*, it shall provide its *DataTypeEncodings* using *HasEncoding References*. These *References* shall be bi-directional. Figure 12 provides an example how *DataTypes* are modelled in the *AddressSpace*.

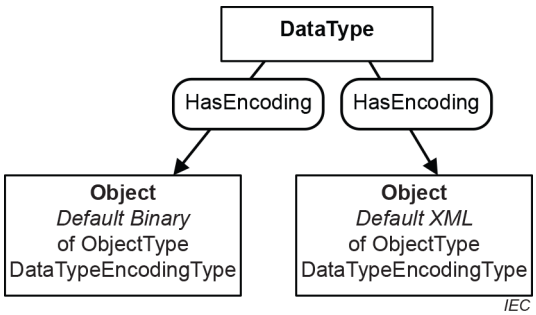


Figure 12 – Example of DataType Modelling

The information on how to encode the *DataType* is provided in the *Attribute DataTypeDefinition* of the *DataType Node*. The content of this *Attribute* shall not be changed once it had been provided to *Clients* since *Clients* might persistently cache this information. If the encoding of a *DataType* needs to be changed conceptually a new *DataType* needs to be provided, meaning that a new *NodeId* shall be used for the *DataType*. Since *Clients* identify the *DataType* via the *DataTypeEncodings*, also the *NodeIds* for the *DataTypeEncodings* of the *DataType* shall be changed, when the encoding changes.

5.9 Summary of Attributes of the NodeClasses

Table 17 summarizes all *Attributes* defined in this document and points out which *NodeClasses* use them either in an optional (O) or mandatory (M) way.

Table 17 – Overview of Attributes

Attribute	Variable	Variable Type	Object	Object Type	Reference Type	DataType	Method	View
AccessLevel	M							
AccessLevelEx	O							
ArrayDimensions	O	O						
AccessRestrictions	O	O	O	O	O	O	O	O
BrowseName	M	M	M	M	M	M	M	M
ContainsNoLoops								M
DataType	M	M						
DataTypeDefinition						O		
Description	O	O	O	O	O	O	O	O
DisplayName	M	M	M	M	M	M	M	M
EventNotifier			M					M
Executable							M	
Historizing	M							
InverseName					O			
IsAbstract		M		M	M	M		
MinimumSamplingInterval	O							
NodeClass	M	M	M	M	M	M	M	M
NodeId	M	M	M	M	M	M	M	M
RolePermissions	O	O	O	O	O	O	O	O
Symmetric					M			
UserAccessLevel	M							
UserExecutable							M	
UserRolePermissions	O	O	O	O	O	O	O	O
UserWriteMask	O	O	O	O	O	O	O	O
Value	M	O						
ValueRank	M	M						
WriteMask	O	O	O	O	O	O	O	O

6 Type Model for ObjectTypes and VariableTypes

6.1 Overview

In the remainder of Clause 6 the type model of *ObjectTypes* and *VariableTypes* is defined regarding subtyping and instantiation.

6.2 Definitions

6.2.1 InstanceDeclaration

An *InstanceDeclaration* is an *Object*, *Variable* or *Method* that references a *ModellingRule* with a *HasModellingRule Reference* and is the *TargetNode* of a *Hierarchical Reference* from a *TypeDefinitionNode* or another *InstanceDeclaration*. The type of an *InstanceDeclaration* may be abstract; however the instance shall be of a concrete type.

6.2.2 Instances without ModellingRules

If no *ModellingRule* exists then the *Node* is neither considered for instantiation of a type nor for subtyping.

If a *Node* referenced by a *TypeDefinitionNode* does not reference a *ModellingRule* it indicates that this *Node* only belongs to the *TypeDefinitionNode* and not to the instances. For example, an *ObjectType Node* may contain a *Property* that describes scenarios where the type could be used. This *Property* would not be considered when creating instances of the type. This is also true for subtyping, that is, subtypes of the type definition would not inherit the referenced *Node*.

6.2.3 InstanceDeclarationHierarchy

The *InstanceDeclarationHierarchy* of a *TypeDefinitionNode* contains the *TypeDefinitionNode* and all *InstanceDeclarations* that are directly or indirectly referenced from the *TypeDefinitionNode* using forward *Hierarchical References*.

6.2.4 Similar Node of InstanceDeclaration

A similar *Node* of an *InstanceDeclaration* is a *Node* that has the same *BrowseName* and *NodeClass* as the *InstanceDeclaration* and in cases of *Variables* and *Objects* the same *TypeDefinitionNode* or a subtype of it.

6.2.5 BrowsePath

All targets of forward *Hierarchical References* from a *TypeDefinitionNode* shall have a *BrowseName* that is unique within the *TypeDefinitionNode*. The same restriction applies to the targets of forward *Hierarchical References* from any *InstanceDeclaration*. This means that any *InstanceDeclaration* within the *InstanceDeclarationHierarchy* can be uniquely identified by a sequence of *BrowseNames*. This sequence of *BrowseNames* is called a *BrowsePath*.

6.2.6 Attribute Handling of InstanceDeclarations

Some restrictions exist regarding the *Attributes* of *InstanceDeclarations* when the *InstanceDeclaration* is overridden or instantiated. The *BrowseName* and the *NodeClass* shall never change and always be the same as the original *InstanceDeclaration*.

In addition, the rules defined in 6.2.7 apply for *InstanceDeclarations* of the *NodeClass Variable*.

6.2.7 Attribute Handling of Variable and VariableTypes

Some restrictions exist regarding the *Attributes* of a *VariableType* or a *Variable* used as an *InstanceDeclaration* with regard to the data type of the *Value Attribute*.

When a *Variable* used as *InstanceDeclaration* or a *VariableType* is overridden or instantiated the following rules apply:

a) The *DataType Attribute* can only be changed to a new *DataType* if the new *DataType* is a subtype of the *DataType* originally used.

b) The *ValueRank Attribute* may only be further restricted:

   1) ‘Any’ may be set to any other value;

   2) ‘ScalarOrOneDimension’ may be set to ‘Scalar’ or ‘OneDimension’;

   3) ‘OneOrMoreDimensions’ may be set to a concrete number of dimensions (value > 0).

   4) All other values of this *Attribute* shall not be changed.

c) The *ArrayDimensions Attribute* may be added if it was not provided or when modifying the value of an entry in the array from 0 to a different value. All other values in the array shall remain the same.

6.2.8 NodeIds of InstanceDeclarations

InstanceDeclarations are identified by their *BrowsePath*. Different *Servers* might use different *NodeIds* for the *InstanceDeclarations* of common *TypeDefinitionNodes*, unless the definition of the *TypeDefinitionNode* already defines a *NodeId* for the *InstanceDeclaration*. All *TypeDefinitionNodes* defined in IEC 62541-5 already define the *NodeIds* for their *InstanceDeclarations* and therefore shall be used in all *Servers*.

6.3 Subtyping of ObjectTypes and VariableTypes

6.3.1 Overview

The *HasSubtype ReferenceType* defines subtypes of types. Subtyping can only occur between *Nodes* of the same *NodeClass*. Rules for subtyping *ReferenceTypes* are described in 5.3.3.3. There is no common definition for subtyping *DataTypes*, as described in 5.8.3. The remainder of 6.3 specifies subtyping rules for single inheritance on *ObjectTypes* and *VariableTypes*.

6.3.2 Attributes

Subtypes inherit the parent type's *Attribute* values, except for the *NodeId*. Inherited *Attribute* values may be overridden by the subtype, the *BrowseName* and *DisplayName* values should be overridden. Special rules apply for some *Attributes* of *VariableTypes* as defined in 6.2.7. Optional *Attributes*, not provided by the parent type, may be added to the subtype.

6.3.3 InstanceDeclarations

6.3.3.1 Overview

Subtypes inherit the fully-inherited parent type's *InstanceDeclarations*.

As long as those *InstanceDeclarations* are not overridden they are not referenced by the subtype. *InstanceDeclarations* can be overridden by adding *References*, changing *References* to reference different *Nodes*, changing *References* to be subtypes of the original *ReferenceType*, changing values of the *Attributes* or adding optional *Attributes*. In order to get the full information about a subtype, the inherited *InstanceDeclarations* have to be collected from all types that can be found by recursively following the inverse *HasSubtype References* from the subtype. This collection of *InstanceDeclarations* is called the fully-inherited *InstanceDeclarationHierarchy* of a subtype.

The remainder of 6.3.3 defines how to construct the fully-inherited *InstanceDeclarationHierarchy* and how *InstanceDeclarations* can be overridden.

6.3.3.2 Fully-inherited InstanceDeclarationHierarchy

An instance of a *TypeDefinitionNode* is described by the fully-inherited *InstanceDeclaration-Hierarchy* of the *TypeDefinitionNode*. The fully-inherited *InstanceDeclarationHierarchy* can be created by starting with the *InstanceDeclarationHierarchy* of the *TypeDefinitionNode* and merging the fully-inherited *InstanceDeclarationHierarchy* of its parent type.

The process of merging *InstanceDeclarationHierarchies* is straightforward and can be illustrated with the example shown in Figure 13 which specifies a *TypeDefinitionNode* "BetaType" which is a subtype of "AlphaType". The name in each box is the *BrowseName* and the number is the *NodeId*.

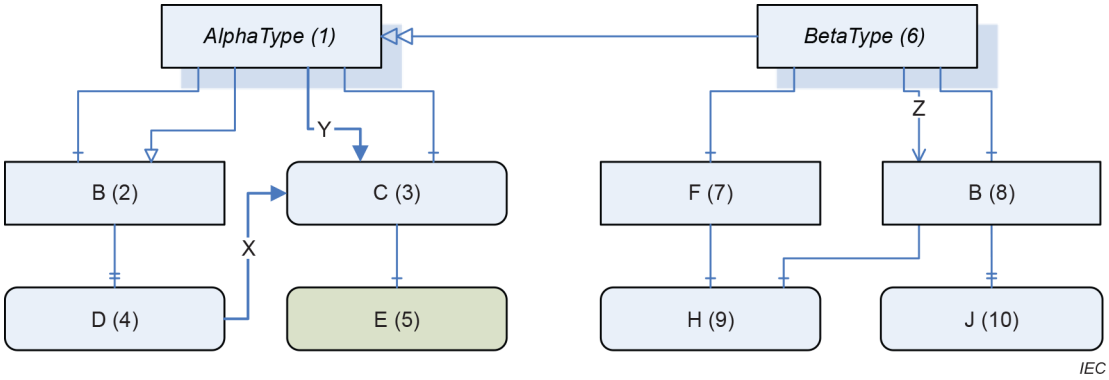


Figure 13 – Subtyping TypeDefinitionNodes

An *InstanceDeclarationHierarchy* can be fully described as a table of *Nodes* identified by their *BrowsePaths* with a corresponding table of *References*. The *InstanceDeclarationHierarchy* for "BetaType" is described in Table 18 where the top half of the table is the table of *Nodes* and the bottom half is the table of *References* (the *HasModellingRule* references have been omitted from the table for the sake of clarity; all Nodes except for 1, 6, and 5 have *ModellingRules*). All *InstanceDeclarations* of the *InstanceDeclarationHierarchy* and all *Nodes* referenced with a non-hierarchical *Reference* from such an *InstanceDeclaration* are added to the table. *Hierarchical References* to *Nodes* without a *ModellingRule* are not considered.

Table 18 – The InstanceDeclarationHierarchy for BetaType

BrowsePath	NodeId
/	6
/F	7
/B	8
/F/H	9
/B/J	10
/B/H	9

Source Path	ReferenceType	Target Path	TargetNodeId
/	HasComponent	/F	-
/	HasComponent	/B	-
/	Z	/B	-
/	HasTypeDefinition	-	BetaType
/F	HasComponent	/F/H	-
/F	HasTypeDefinition	-	BaseObjectType
/B	HasProperty	/B/J	-
/B	HasTypeDefinition	-	BaseObjectType
/F/H	HasTypeDefinition	-	PropertyType
/B/J	HasTypeDefinition	-	PropertyType
/B	HasComponent	/B/H	-
/B/H	HasTypeDefinition	-	BaseDataVariableType

Multiple *BrowsePaths* to the same *Node* shall be treated as separate *Nodes*. An *Instance* may provide different *Nodes* for each *BrowsePath*.

The fully-inherited *InstanceDeclarationHierarchy* for “BetaType” can now be constructed by merging the *InstanceDeclarationHierarchy* for “AlphaType”. The result is shown in Table 19 where the entries added from “AlphaType” are shaded with grey.

Table 19 – The Fully-Inherited InstanceDeclarationHierarchy for BetaType

BrowsePath	NodeId
/	6
/F	7
/B	8
/F/H	9
/B/J	10
/B/H	9
/B/D	4
/C	3

Source Path	ReferenceType	Target Path	TargetNodeId
/	HasComponent	/F	-
/	HasComponent	/B	-
/	Z	/B	-
/	HasTypeDefinition	-	BetaType
/F	HasComponent	/F/H	-
/F	HasTypeDefinition	-	BaseObjectType
/B	HasProperty	/B/J	-
/B	HasTypeDefinition	-	BaseObjectType
/F/H	HasTypeDefinition	-	PropertyType
/B/J	HasTypeDefinition	-	PropertyType
/B	HasComponent	/B/H	-
/B/H	HasTypeDefinition	-	BaseDataVariableType
/	HasNotifier	/B	-
/B	HasProperty	/B/D	-
/	HasComponent	/C	-
/	Y	/C	-
/C	HasTypeDefinition	-	BaseDataVariableType
/B/D	HasTypeDefinition	-	PropertyType
/B/D	X	/C	-

The *BrowsePath* “/B” already exists in the table so it does not need to be added. However, the *HasNotifier* reference from “/” to “/B” does not exist and was added.

The *Nodes* and *References* defined in Table 19 can be used to create the fully-inherited *InstanceDeclarationHierarchy* shown in Figure 14. The fully-inherited *InstanceDeclarationHierarchy* contains all necessary information about a *TypeDefinitionNode* regarding its complex structure without needing any additional information from its supertypes.

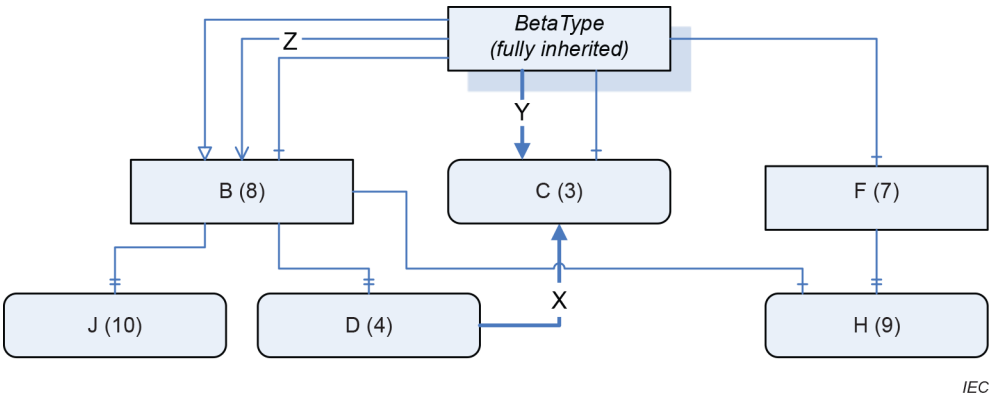


Figure 14 – The Fully-Inherited InstanceDeclarationHierarchy for BetaType

6.3.3.3 Overriding InstanceDeclarations

A subtype overrides an *InstanceDeclaration* by specifying an *InstanceDeclaration* with the same *BrowsePath*. An overridden *InstanceDeclaration* shall have the same *NodeClass* and *BrowseName*. The *TypeDefinitionNode* of the overridden *InstanceDeclaration* shall be the same or a subtype of the *TypeDefinitionNode* specified in the supertype.

When overriding an *InstanceDeclaration* it is necessary to provide *Hierarchical References* that link the new *Node* back to the subtype (the *References* are used to determine the *BrowsePath* of the *Node*).

It is only possible to override *InstanceDeclarations* that are directly referenced from the *TypeDefinitionNode*. If an indirect referenced *InstanceDeclaration*, such as “J” in Figure 14, has to be overridden, then the directly referenced *InstanceDeclarations* that includes “J”, in that case “B”, have to be overridden first and then “J” can be overridden in a second step.

A *Reference* is replaced if it goes between two overridden *Nodes* and has the same *ReferenceType* as a *Reference* defined in the supertype. The *Reference* specified in the subtype may be a subtype of the *ReferenceType* used in the parent type.

Any non-hierarchical *References* specified for the overridden *InstanceDeclaration* are treated as new *References* unless the *ReferenceType* only allows a single *Reference* per *SourceNode*. If this situation exists the subtype can change the target of the *Reference* but the new target shall have the same *NodeClass* and for *Objects* and *Variables* also the same type or a subtype of the type specified in the parent.

The overriding *Node* may specify new values for the *Node Attributes* other than the *NodeClass* or *BrowseName*, however, the restrictions on *Attributes* specified in 6.2.6 apply. Any *Attribute* provided by the overridden *InstanceDeclaration* shall be provided by the overriding *InstanceDeclaration*, additional optional *Attributes* may be added.

The *ModellingRule* of the overriding *InstanceDeclaration* may be changed as defined in 6.4.4.3.

Each overriding *InstanceDeclaration* needs its own *HasModellingRule* and *HasTypeDefinition References*, even if they have not been changed.

A subtype should not override a *Node* unless it needs to change it.

The semantics of certain *TypeDefinitionNodes* and *ReferenceTypes* may impose additional restrictions with regard to overriding *Nodes*.

6.4 Instances of ObjectTypes and VariableTypes

6.4.1 Overview

Any *Instance* of a *TypeDefinitionNode* will be the root of a hierarchy which mirrors the *InstanceDeclarationHierarchy* for the *TypeDefinitionNode*. Each *Node* in the hierarchy of the Instance will have a *BrowsePath* which may be the same as the *BrowsePath* for one of the *InstanceDeclarations* in the hierarchy of the *TypeDefinitionNode*. The *InstanceDeclaration* with the same *BrowsePath* is called the *InstanceDeclaration* for the *Node*. If a *Node* has an *InstanceDeclaration* then it shall have the same *BrowseName* and *NodeClass* as the *InstanceDeclaration* and, in cases of *Variables* and *Objects*, the same *TypeDefinitionNode* or a subtype of it.

Instances may reference several *Nodes* with the same *BrowsePath*. *Clients* that need to distinguish between the *Nodes* based on the *InstanceDeclarationHierarchy* and the *Nodes* that are not based on the *InstanceDeclarationHierarchy* can accomplish this using the TranslateBrowsePathsToNodeIds *Service* defined in IEC 62541-4.

6.4.2 Creating an Instance

Instances inherit the initial values for the *Attributes* that they have in common with the *TypeDefinitionNode* from which they are instantiated, with the exceptions of the *NodeClass* and *NodeId*.

When a *Server* creates an instance of a *TypeDefinitionNode* it shall create the same hierarchy of *Nodes* beneath the new *Object* or *Variable* depending on the *ModellingRule* of each *InstanceDeclaration*. Standard *ModellingRules* are defined in 6.4.4.5. The *Nodes* within the newly created hierarchy may be copies of the *InstanceDeclarations*, the *InstanceDeclaration* itself or another *Node* in the *AddressSpace* that has the same *TypeDefinitionNode* and *BrowseName*. If new copies are created, then the *Attribute* values of the *InstanceDeclarations* are used as the initial values.

Figure 15 provides a simple example of a *TypeDefinitionNode* and an *Instance*. *Nodes* referenced by the *TypeDefinitionNode* without a *ModellingRule* do not appear in the instance. *Instances* may have children with duplicate *BrowseNames*; however, only one of those children will correspond to the *InstanceDeclaration*.

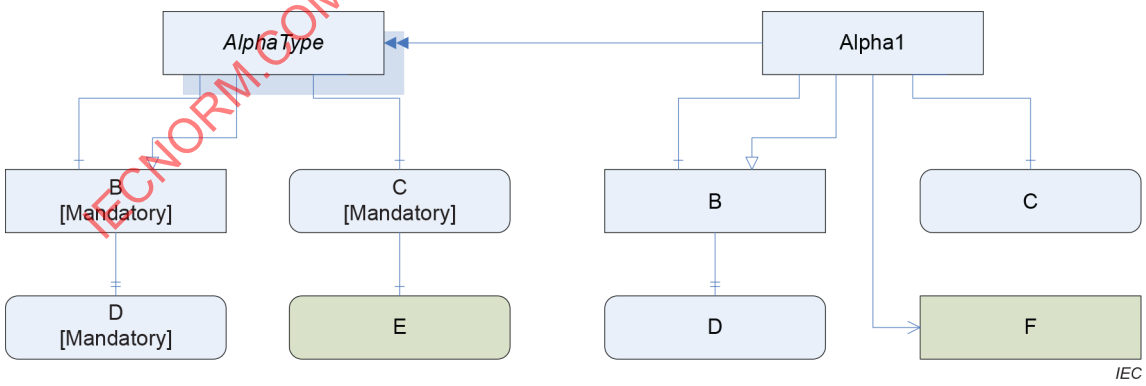


Figure 15 – An Instance and its TypeDefinitionNode

It is up to the *Server* to decide which *InstanceDeclarations* appear in any single instance. In some cases, the *Server* will not define the entire instance and will provide remote references to *Nodes* in another *Server*. The *ModellingRules* described in 6.4.4.5 allow *Servers* to indicate that some *Nodes* are always present; however, the *Client* shall be prepared for the case where the *Node* exists in a different *Server*.

A *Client* can use the information of *TypeDefinitionNodes* to access *Nodes* which are in the hierarchy of the instance. It shall pass the *NodeId* of the instance and the *BrowsePath* of the child *Nodes* based on the *TypeDefinitionNode* to the TranslateBrowsePathsToNodeIds *Service* (see IEC 62541-4). This *Service* returns the *NodeId* for each of the child *Nodes*. If a child *Node* exists then the *BrowseName* and *NodeClass* shall match the *InstanceDeclaration*. In the case of *Objects* or *Variables*, also the *TypeDefinitionNode* shall either match or be a subtype of the original *TypeDefinitionNode*.

6.4.3 Constraints on an Instance

Objects and *Variables* may change their *Attribute* values after being created. Special rules apply for some *Attributes* as defined in 6.2.6.

Additional *References* may be added to the *Nodes*, and *References* may be deleted as long as the *ModellingRules* defined on the *InstanceDeclarations* of the *TypeDefinitionNode* are still fulfilled.

For *Variables* and *Objects* the *HasTypeDefinition Reference* shall always point to the same *TypeDefinitionNode* as the *InstanceDeclaration* or a subtype of it.

If two *InstanceDeclarations* of the fully-inherited *InstanceDeclarationHierarchy* have been connected directly with several *References*, all those *References* shall connect the same *Nodes*. An example is given in Figure 16. The instances A1 and A2 are allowed since B1 references the same *Node* with both *References*, whereas A3 is not allowed since two different *Nodes* are referenced. Note that this restriction only applies for directly connected *Nodes*. For example, A2 references a C1 directly and a different C1 via B1.

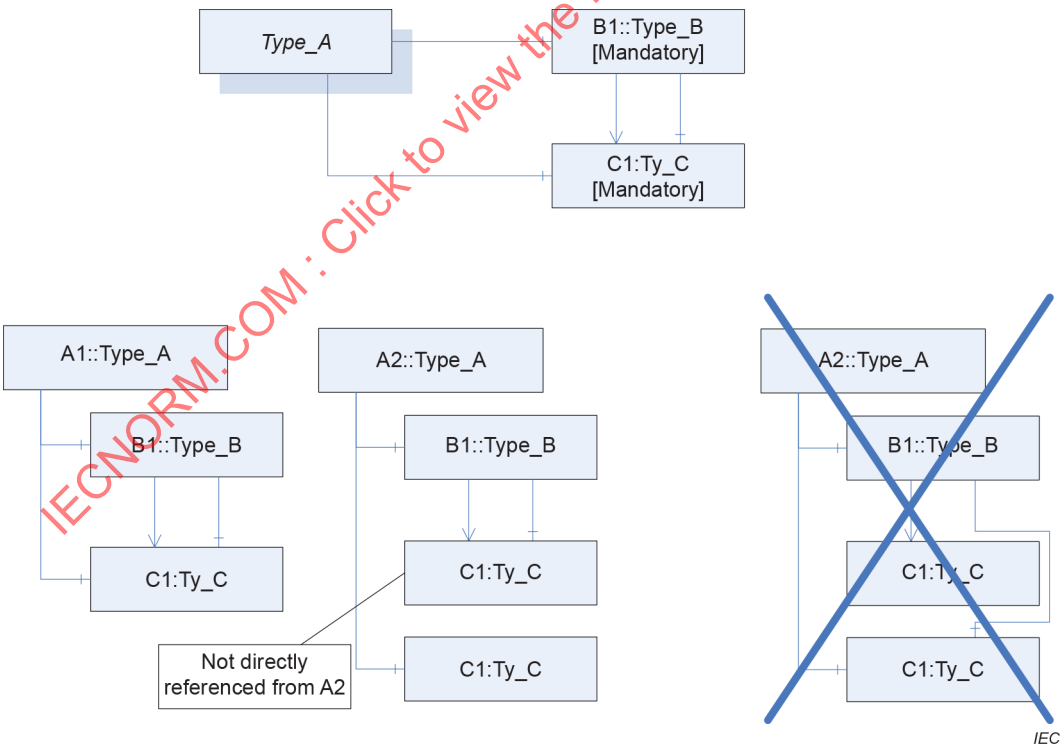


Figure 16 – Example of several References between InstanceDeclarations

6.4.4 ModellingRules

6.4.4.1 General

For a definition of *ModellingRules*, see 6.4.4.5. Other parts of IEC 62541 may define additional *ModellingRules*. *ModellingRules* are an extendable concept in OPC UA; therefore vendors may define their own *ModellingRules*.

Note that the *ModellingRules* defined in this document do not define how to deal with non-hierarchical *References* between *InstanceDeclarations*, i.e. it is *Server*-specific if those *References* exist in an instance hierarchy or not. Other *ModellingRules* may define behaviour for non-hierarchical *References* between *InstanceDeclaration* as well.

ModellingRules are represented in the *AddressSpace* as *Objects* of the *ObjectType ModellingRuleType*. There are some *Properties* defining common semantic of *ModellingRules.* This document only specifies one *Property* for *ModellingRules*. Future editions of IEC 62541-3 may define additional *Properties* for *ModellingRules*. IEC 62541-5 specifies the representation of the *ModellingRule Objects,* their *Properties* and their type in the *AddressSpace*. The semantic of the *Properties* for *ModellingRules* is defined in 6.4.4.2.

Subclause 6.4.4.4 defines how the *ModellingRule* may be changed when instantiating *InstanceDeclarations* with respect to the *Properties*. Subclause 6.4.4.3 defines how the *ModellingRule* may be changed when overriding *InstanceDeclarations* in subtypes with respect to the *Properties*.

6.4.4.2 Properties describing ModellingRules – NamingRule

NamingRule is a mandatory *Property* of a *ModellingRule*. It specifies the purpose of an *InstanceDeclaration*. Each *InstanceDeclaration* references a *ModellingRule* and thus the *NamingRule* is defined per *InstanceDeclaration*.

Three values are allowed for the *NamingRule* of a *ModellingRule*: *Optional*, *Mandatory*, and *Constraint*.

The following semantic is valid for the entire life-time of an instance that is based on a *TypeDefinitionNode* having an *InstanceDeclaration*.

For an instance A1 of a *TypeDefinitionNode* AlphaType with an *InstanceDeclaration* B1 having a *ModellingRule* using the *NamingRule Optional* the following rule applies: For each *BrowsePath* from AlphaType to B1 the instance A1 may or may not have a *similar Node* (see 6.2.4) for B1 with the same *BrowsePath*. If such a *Node* exists then the TranslateBrowsePathsToNodeIds *Service* (see IEC 62541-4) returns this *Node* as the first entry in the list.

For an instance A1 of a *TypeDefinitionNode* AlphaType with an *InstanceDeclaration* B1 having a *ModellingRule* using the *NamingRule Mandatory* the following rule applies: For each *BrowsePath* from AlphaType to B1 the instance A1 shall have a *similar Node* (see 6.2.4) for B1 using the same *BrowsePath* if all *Nodes* of the *BrowsePath* exist. For example, if a *Node* in the *BrowsePath* has a *NamingRule Optional* and is omitted in the instance, then all children of this *Node* would also be omitted, independent of their *ModellingRules*.

If an *InstanceDeclaration* has a *ModellingRule* using the *NamingRule Constraint* it identifies that the *BrowseName* of the *InstanceDeclaration* is of no significance but other semantic is defined with the *ModellingRule*. The TranslateBrowsePathsToNodeIds *Service* (see IEC 62541-4) can typically not be used to access instances based on those *InstanceDeclarations*.

6.4.4.3 Subtyping rules for Properties of ModellingRules

It is allowed that subtypes override *ModellingRules* on their *InstanceDeclarations*. As a general rule for subtyping, constraints shall only be tightened, not loosened. Therefore, it is not allowed to specify on the supertype that an instance shall exist with the name (*NamingRule Mandatory*) and on the subtype make this optional (*NamingRule Optional*). Table 20 specifies the allowed changes on the *Properties* when exchanging the *ModellingRules* in the subtype.

Table 20 – Rule for ModellingRules Properties when Subtyping

	Value on supertype	Value on subtype
NamingRule	Mandatory	Mandatory
NamingRule	Optional	Mandatory or Optional
NamingRule	Constraint	Constraint

6.4.4.4 Instantiation rules for Properties of ModellingRules

There are two different use cases when creating an instance 'A' based on a *TypeDefinitionNode* 'A_Type'. Either 'A' is used as normal instance or it is used as *InstanceDeclaration* of another *TypeDefinitionNode*.

In the first case, it is not required that newly created or referenced instances based on *InstanceDeclarations* have a *ModellingRule*, however, it is allowed that they have any *ModellingRule* independent of the *ModellingRule* of their *InstanceDeclaration*.

In Figure 17 an example is given. The instances A1, A2, and A3 are all valid instances of Type_A, although B of A1 has no *ModellingRule* and B of A3 has a different *ModellingRule* than B of Type_A.

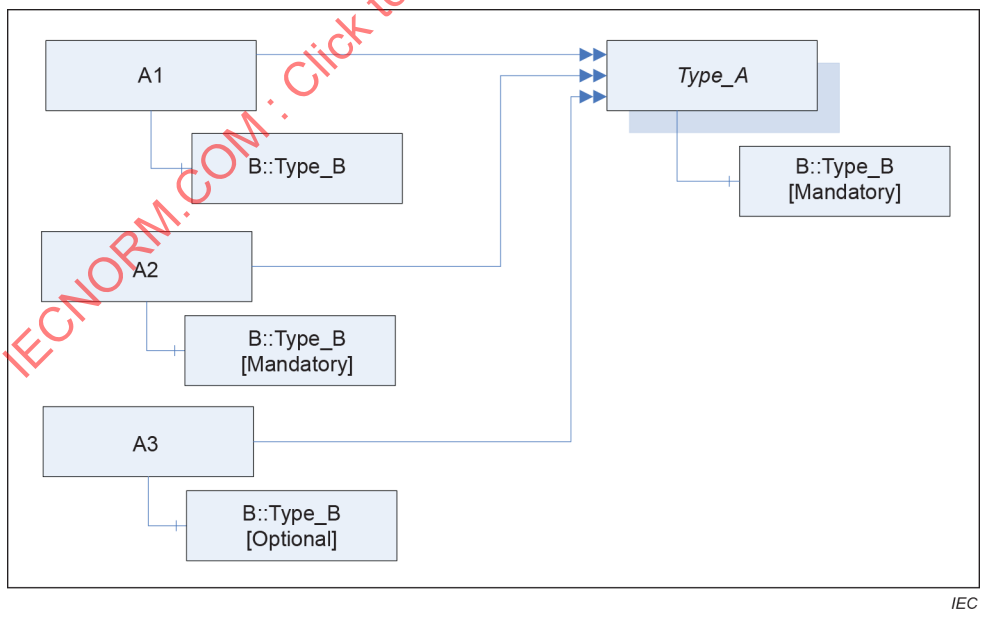


Figure 17 – Example of changing instances based on InstanceDeclarations

In the second case, all instances that are referenced directly or indirectly from 'A' based on *InstanceDeclarations* of 'A_Type' initially maintain the same *ModellingRule* as their *InstanceDeclarations*. The *ModellingRules* may be updated; the allowed changes to the *ModellingRules* of these *Nodes* are the same as those defined for subtyping in 6.4.4.3.

In Figure 18 an example of such a scenario is given. Type_B uses an *InstanceDeclaration* based on Type_A (upper part of the figure). Later on the *ModellingRule* of the *InstanceDeclaration* A1 is changed (lower part of the figure). A1 has become the *NamingRule* of *Mandatory* (changed from *Optional*).

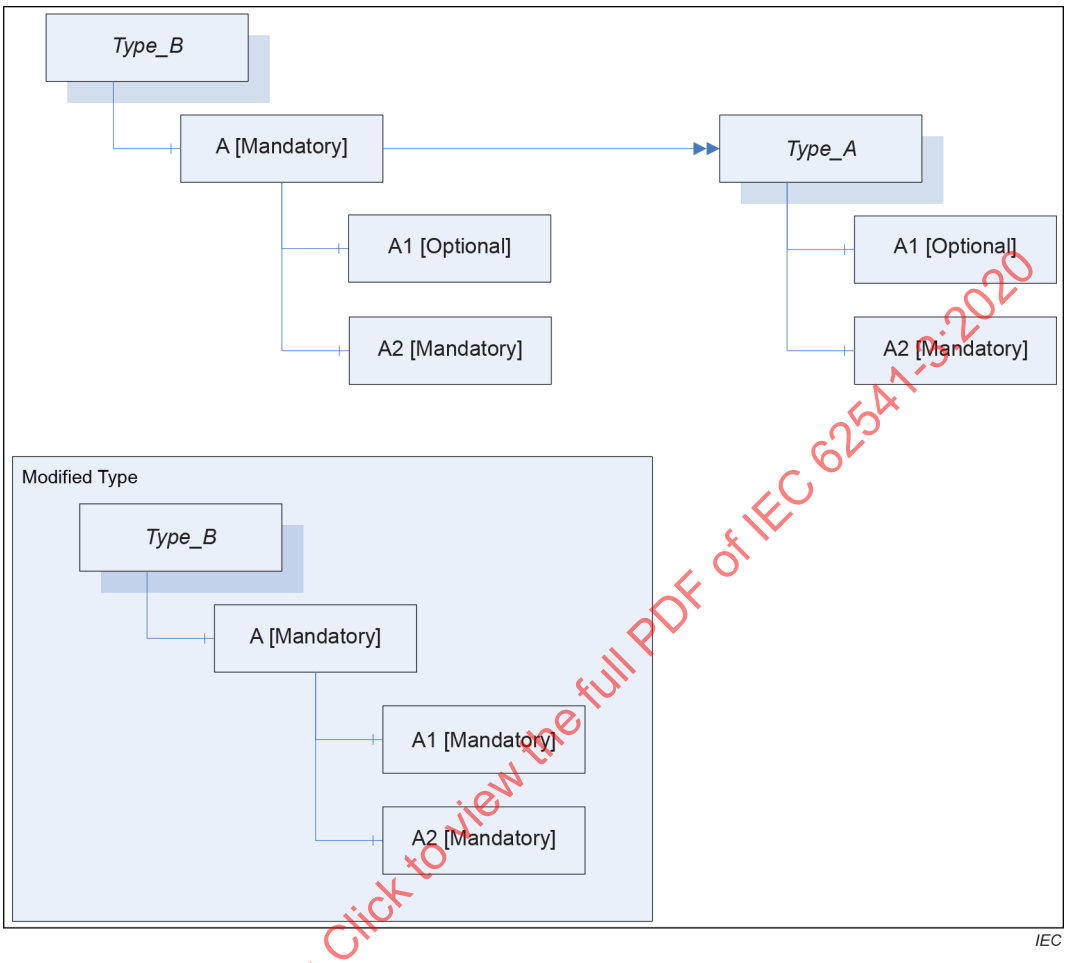


Figure 18 – Example of changing InstanceDeclarations based on an InstanceDeclaration

6.4.4.5 Standard ModellingRules

6.4.4.5.1 Titles of standard ModellingRules

The remainder of 6.4.4.5 defines *ModellingRules*. In Table 21 the *Properties* of those *ModellingRules* are summarized.

Table 21 – Properties of ModellingRules

Title	NamingRule
Mandatory	Mandatory
Optional	Optional
ExposesItsArray	Constraint
OptionalPlaceholder	Constraint
MandatoryPlaceholder	Constraint

6.4.4.5.2 Mandatory

An *InstanceDeclaration* marked with the *ModellingRule Mandatory* fulfils exactly the semantic defined for the *NamingRule Mandatory*. That means that for each existing *BrowsePath* on the instance a similar *Node* shall exist, but it is not defined whether a new *Node* is created or an existing *Node* is referenced.

For example, the *TypeDefinitionNode* of a functional block "AI_BLK_TYPE" will have a setpoint "SP1". An instance of this type "AI_BLK_1" will have a newly-created setpoint "SP1" as a similar Node to the *InstanceDeclaration* SP1. Figure 19 illustrates the example.

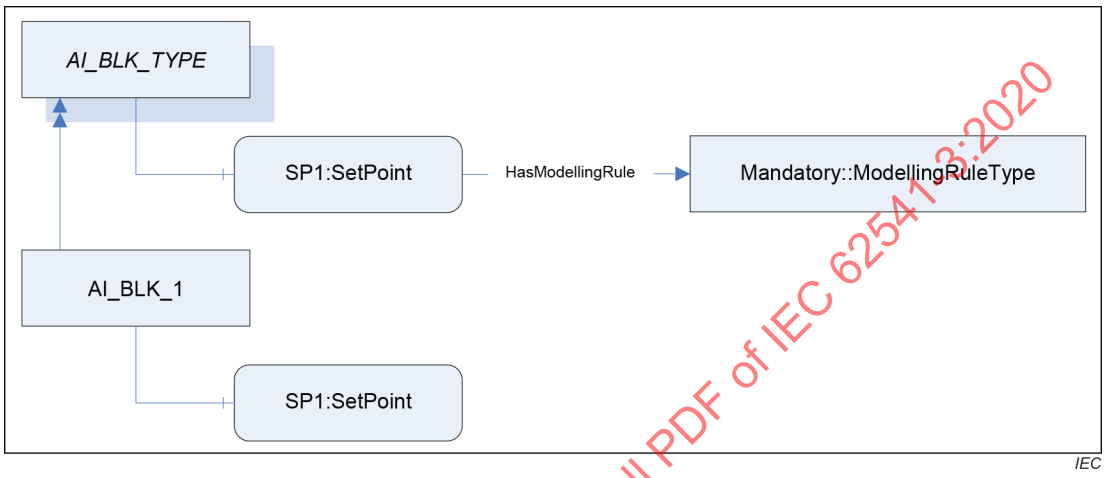


Figure 19 – Use of the Standard ModellingRule Mandatory

In 6.4.4.5.3 a complex example combining the *Mandatory* and *Optional ModellingRules* is given.

6.4.4.5.3 Optional

An *InstanceDeclaration* marked with the ModellingRule *Optional* fulfils exactly the semantic defined for the *NamingRule Optional*. That means that for each existing *BrowsePath* on the instance a similar *Node* may exist, but it is not defined whether a new *Node* is created or an existing *Node* is referenced.

In Figure 20 an example using the *ModellingRules Optional* and *Mandatory* is shown. The example contains an *ObjectType* Type_A and all valid combinations of instances named A1 to A13. Note that if the optional B is provided, the mandatory E has to be provided as well, otherwise not. F is referenced by C and D. On the instance, this can be the same *Node* or two different *Nodes* with the same *BrowseName* (similar *Node* to *InstanceDeclaration* F). Not considered in the example is if the instances have *ModellingRules* or not. It is assumed that each F is similar to the *InstanceDeclaration* F, etc.

If there would be a non-hierarchical *Reference* between E and F in the *InstanceDeclaration-Hierarchy*, it is not specified if it occurs in the instance hierarchy or not. In the case of A10, there could be a reference from E to one F but not to the other F, or to both or none of them.

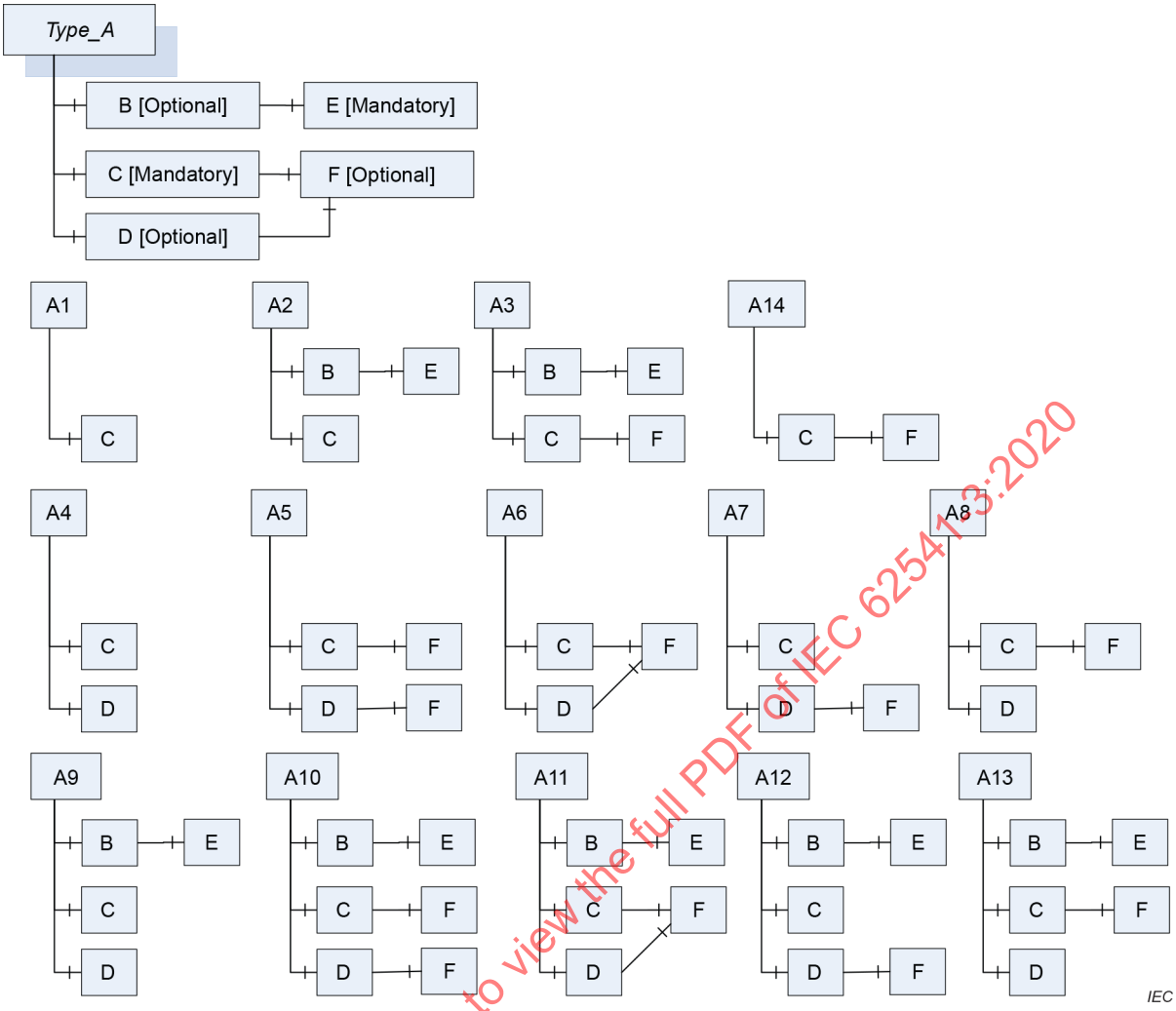


Figure 20 – Example using the Standard ModellingRules Optional and Mandatory

6.4.4.5.4 ExposesItsArray

The *ExposesItsArray ModellingRule* exposes a special semantic on *VariableTypes* having a single- or multidimensional array as the data type. It indicates that each value of the array will also be exposed as a *Variable* in the *AddressSpace*.

The *ExposesItsArray ModellingRule* can only be applied on *InstanceDeclarations* of *NodeClass Variable* that are part of a *VariableType* having a single- or multidimensional array as its data type.

The *Variable* A having this *ModellingRule* shall be referenced by a forward *Hierarchical Reference* from a *VariableType* B. B shall have a *ValueRank* value that is equal to or larger than zero. A should have a data type that reflects at least parts of the data that is managed in the array of B. Each instance of B shall reference one instance of A for each of its array elements. The used *Reference* shall be of the same type as the *Hierarchical Reference* that connects B with A or a subtype of it. If there are more than one forward *Hierarchical References* between A and B, then all instances based on B shall be referenced with all those *References*.

Figure 21 gives an example. A is an instance of Type_A having two entries in its value array. Therefore it references two instances of the same type as the *InstanceDeclaration* ArrayExpose. The *BrowseNames* of those instances are not defined by the *ModellingRule*. In general, it is not possible to get a *Variable* representing a specific entry in the array (e.g. the

second). *Clients* will typically either get the array or access the *Variables* directly, so there is no need to provide that information.

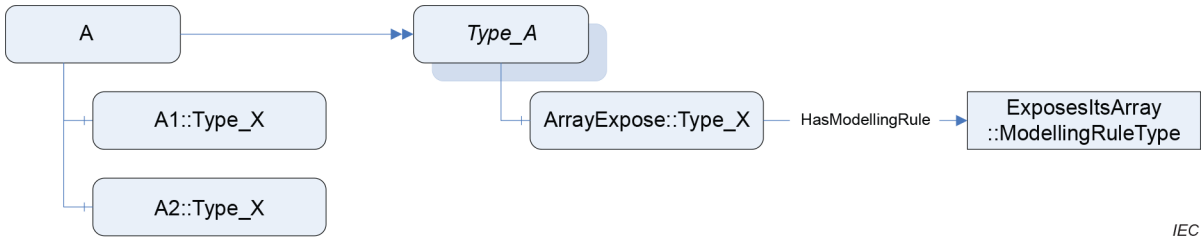


Figure 21 – Example of using ExposesItsArray

It is allowed to reference A by other *InstanceDeclarations* as well. Those *References* shall be reflected on each instance based on A.

Figure 22 gives an example. The *Property* EUUnit is referenced by ArrayExpose and therefore each instance based on ArrayExpose references the instance based on the *InstanceDeclaration* EUUnit.

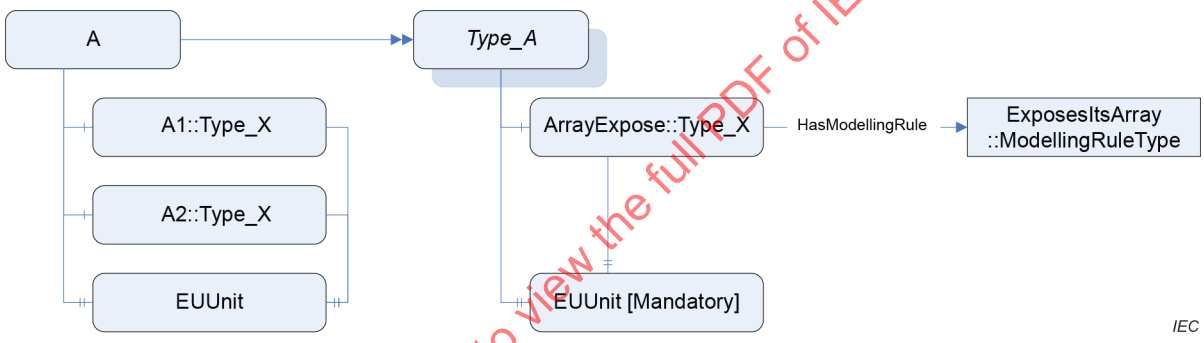


Figure 22 – Complex example of using ExposesItsArray

6.4.4.5.5 OptionalPlaceholder

For *Object* and *Variable* the intention of the *ModellingRule OptionalPlaceholder* is to expose the information that a complex *TypeDefinition* expects from instances of the *TypeDefinition* to add instances with specific *References* without defining *BrowseNames* for the instances. For example, a Device might have a Folder for DeviceParameters, and the DeviceParameters should be connected with a *HasComponent Reference*. However, the names of the DeviceParameters are specific to the instances. The example is shown in Figure 23, where an instance Device A adds two DeviceParameters in the Folder.

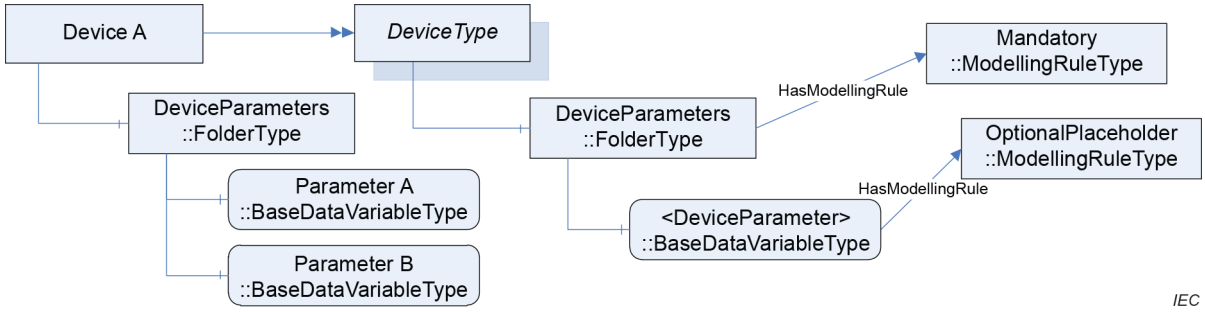


Figure 23 – Example using OptionalPlaceholder with an Object and Variable

The *ModellingRule OptionalPlaceholder* adds no additional constraints on instances of the *TypeDefinition*. It just provides useful information when exposing a *TypeDefinition*. When the *InstanceDeclaration* is complex, i.e. it references other *InstanceDeclarations* with *Hierarchical References*, these *InstanceDeclarations* are not further considered for instantiating the *TypeDefinition*.

It is recommended that the *BrowseName* and the *DisplayName* of *InstanceDeclarations* having the *OptionalPlaceholder ModellingRule* should be enclosed within angle brackets.

When overriding the *InstanceDeclaration,* the *ModellingRule* shall remain *OptionalPlaceholder*.

For *Methods,* the *ModellingRule OptionalPlaceholder* is used to define the *BrowseName* where subtypes and instances provide more information. The *Method* definition with the *OptionalPlaceholder* only defines the *BrowseName*. An instance or subtype defines the *InputArguments* and *OutputArguments*. A subtype shall also change the *ModellingRule* to *Optional* or *Mandatory*. The *Method* is optional for *instances.* For example, a Device might have a *Method* to perform calibration, however the specific arguments for the *Method* depend on the instance of the Device. In this example Device A does not implement the *Method*, Device B implements the *Method* with no arguments and Device C implements the *Method* accepting a mode argument to select how the calibration is to be performed. The example is shown in Figure 24.

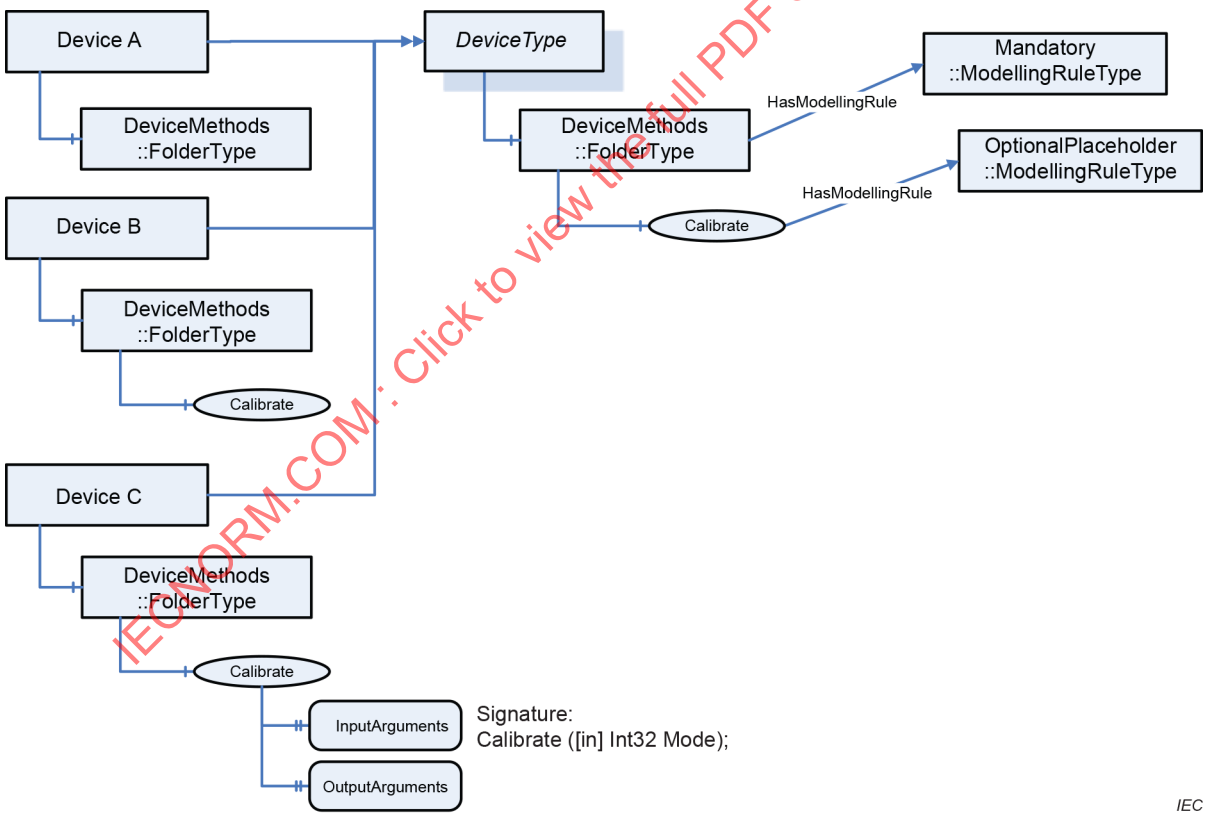


Figure 24 – Example using OptionalPlaceholder with a Method

6.4.4.5.6 MandatoryPlaceholder

For *Object* and *Variable* the *ModellingRule MandatoryPlaceholder* has a similar intention as the *ModellingRule OptionalPlaceholder*. It exposes the information that a *TypeDefinition* expects of instances of the *TypeDefinition* to add instances defined by the *InstanceDeclaration*. However, *MandatoryPlaceholder* requires that at least one of those instances shall exist.

For example, when the DeviceType requires that at least one DeviceParameter shall exist without specifying the *BrowseName* for it, it uses *MandatoryPlaceholder* as shown in Figure 25. Device A is a valid instance as it has the required DeviceParameter. Device B is not valid as it uses the wrong *ReferenceType* to reference a DeviceParameter (*Organizes* instead of *HasComponent*) and Device C is not valid because it does not provide a DeviceParameter at all.

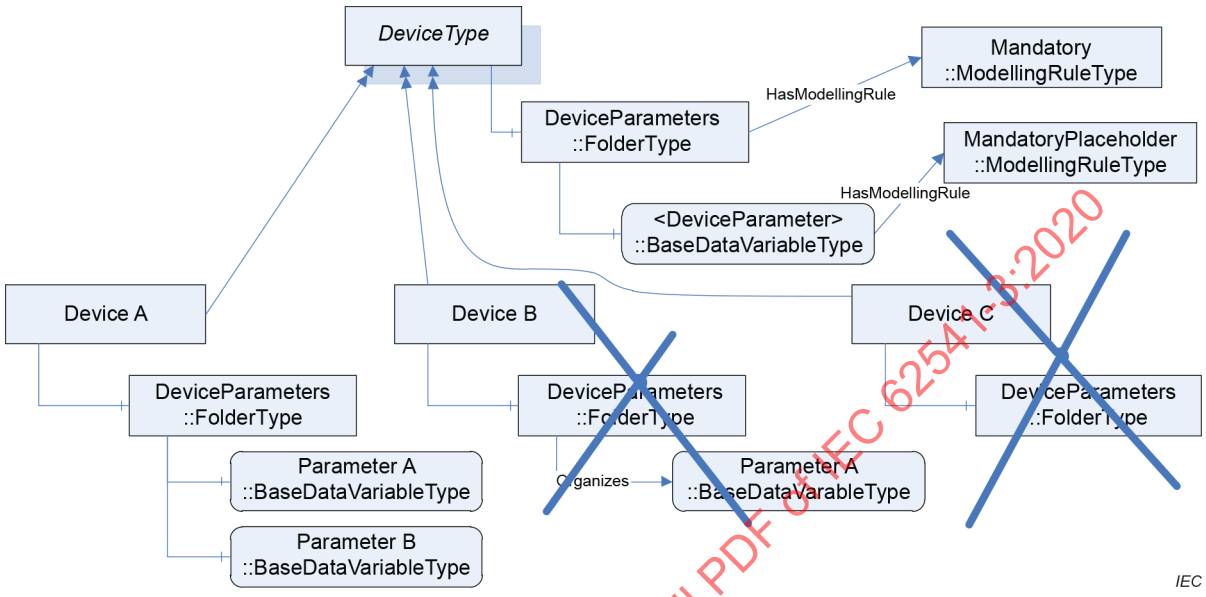


Figure 25 – Example of using MandatoryPlaceholder for Object and Variable

The *ModellingRule MandatoryPlaceholder* requires that each instance provides at least one instance with the *TypeDefinition* of the *InstanceDeclaration* or a subtype, and is referenced with the same *ReferenceType* or a subtype as the *InstanceDeclaration*. It does not require a specific *BrowseName* and thus cannot be used for the TranslateBrowsePathsToNodeIds *Service* (see IEC 62541-4).

When the *InstanceDeclaration* is complex, i.e. it references other *InstanceDeclarations* with *Hierarchical References*, these *InstanceDeclarations* are not further considered for instantiating the *TypeDefinition*.

It is recommended that the *BrowseName* and the *DisplayName* of *InstanceDeclarations* having the *MandatoryPlaceholder ModellingRule* should be enclosed within angle brackets.

When overriding the *InstanceDeclaration,* the *ModellingRule* shall remain *MandatoryPlaceholder*.

For *Methods,* the *ModellingRule MandatoryPlaceholder* is used to define the *BrowseName* where subtypes and instances provide more information. The *Method* definition with the *MandatoryPlaceholder* only defines the *BrowseName*. An instance or subtype defines the *InputArguments* and *OutputArguments*. A subtype shall also change the *ModellingRule* to *Mandatory*. The *Method* is mandatory for *instances.*

6.5 Changing type definitions that are already used

There is no behaviour specified regarding subtypes and instances when changing *ObjectTypes* and *VariableTypes*. It is *Server*-dependent, if those changes are reflected on the subtypes and instances of the types. However, all constraints defined for subtypes and instances shall be fulfilled. For example, it is not allowed to add a *Property* using the *ModellingRule Mandatory* on a type if instances of this type exist without the *Property*. In that

case, the *Server* either shall add the *Property* to all instances of the type or adding the *Property* on the type shall be rejected.

7 Standard ReferenceTypes

7.1 General

This document defines *ReferenceTypes* as an inherent part of the OPC UA Address Space Model. Figure 26 informally describes the hierarchy of these *ReferenceTypes*. Other parts of IEC 62541 may specify additional *ReferenceTypes*. The remainder of Clause 7 defines the *ReferenceTypes*. IEC 62541-5 defines their representation in the *AddressSpace*.

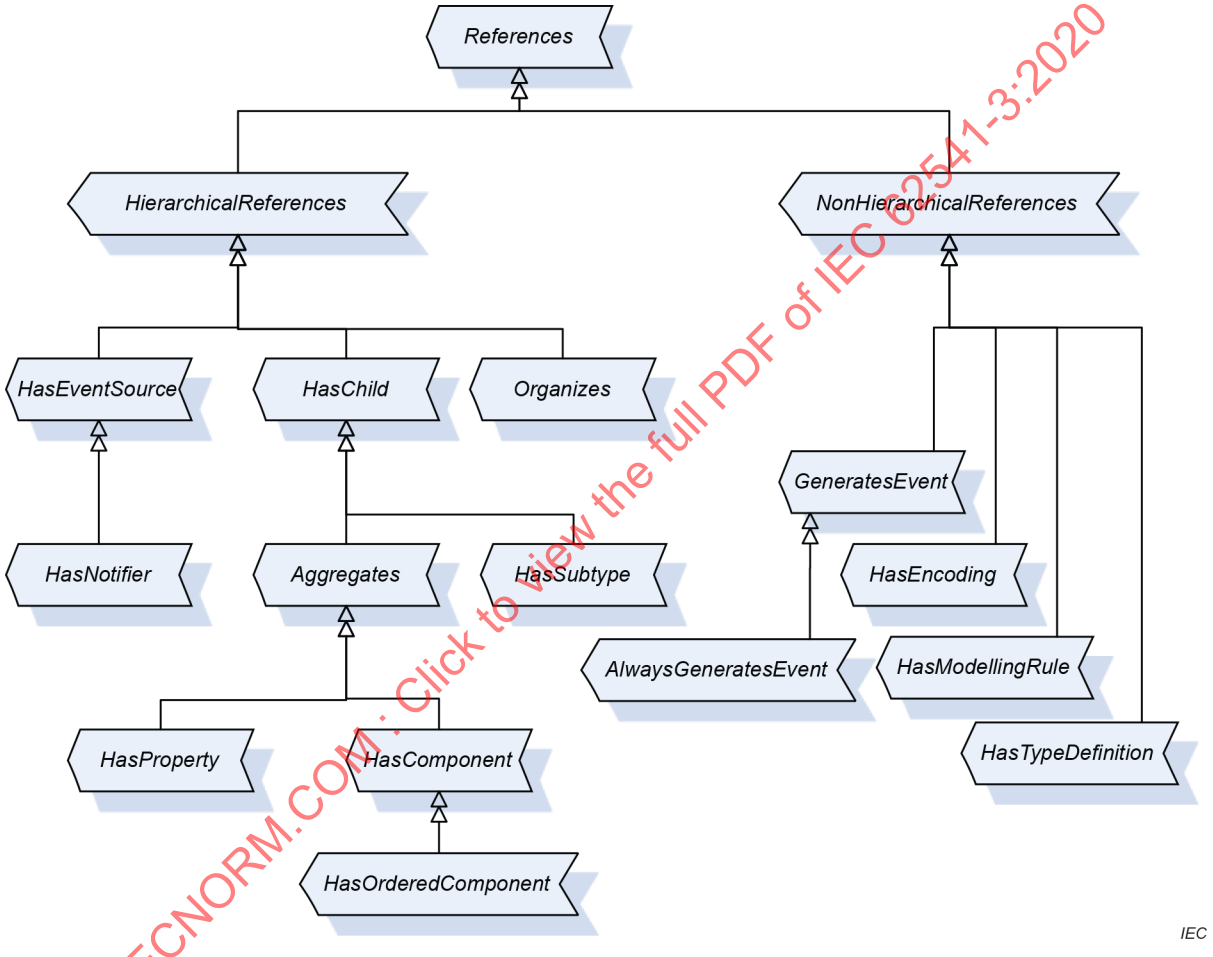


Figure 26 – Standard ReferenceType Hierarchy

7.2 References ReferenceType

The *References ReferenceType* is an abstract *ReferenceType*; only subtypes of it can be used.

There is no semantic associated with this *ReferenceType*. This is the base type of all *ReferenceTypes*. All *ReferenceTypes* shall be a subtype of this base *ReferenceType* – either direct or indirect. The main purpose of this *ReferenceType* is allowing simple filter and queries in the corresponding *Services* of IEC 62541-5.

There are no constraints defined for this abstract *ReferenceType*.

7.3 HierarchicalReferences ReferenceType

The *HierarchicalReferences ReferenceType* is an abstract *ReferenceType*; only subtypes of it can be used.

The semantic of *HierarchicalReferences* is to denote that *References* of *HierarchicalReferences* span a hierarchy. It means that it may be useful to present *Nodes* related with *References* of this type in a hierarchical-like way. *HierarchicalReferences* does not forbid loops. For example, starting from *Node* "A" and following *HierarchicalReferences* it may be possible to browse to *Node* "A", again.

It is not permitted to have a *Property* as *SourceNode* of a *Reference* of any subtype of this abstract *ReferenceType*.

It is not allowed that the *SourceNode* and the *TargetNode* of a *Reference* of the *ReferenceType HierarchicalReferences* are the same, that is, it is not allowed to have self-references using *HierarchicalReferences*.

7.4 NonHierarchicalReferences ReferenceType

The *NonHierarchicalReferences ReferenceType* is an abstract *ReferenceType*; only subtypes of it can be used.

The semantic of *NonHierarchicalReferences* is to denote that its subtypes do not span a hierarchy and should not be followed when trying to present a hierarchy. To distinguish hierarchical and non-hierarchical *References*, all concrete *ReferenceTypes* shall inherit from either *Hierarchical References* or non-hierarchical *References*, either direct or indirect.

There are no constraints defined for this abstract *ReferenceType*.

7.5 HasChild ReferenceType

The *HasChild ReferenceType* is an abstract *ReferenceType*; only subtypes of it can be used. It is a subtype of *HierarchicalReferences*.

The semantic is to indicate that *References* of this type span a non-looping hierarchy.

Starting from *Node* "A" and only following *References* of the subtypes of the *HasChild ReferenceType* it shall never be possible to return to "A". But it is allowed that following the *References* there may be more than one path leading to another *Node* "B".

7.6 Aggregates ReferenceType

The *Aggregates ReferenceType* is an abstract *ReferenceType*; only subtypes of it can be used. It is a subtype of *HasChild*.

The semantic is to indicate a part (the *TargetNode*) belongs to the *SourceNode*. It does not specify the ownership of the *TargetNode*.

There are no constraints defined for this abstract *ReferenceType*.

7.7 HasComponent ReferenceType

The *HasComponent ReferenceType* is a concrete *ReferenceType* that can be used directly. It is a subtype of the *Aggregates ReferenceType*.

The semantic is a part-of relationship. The *TargetNode* of a *Reference* of the *HasComponent ReferenceType* is a part of the *SourceNode*. This *ReferenceType* is used to relate *Objects* or

ObjectTypes with their containing *Objects*, *DataVariables*, and *Methods*. This *ReferenceType* is also used to relate complex *Variable*s or *VariableTypes* with their *DataVariables*.

Like all other *ReferenceTypes*, this *ReferenceType* does not specify anything about the ownership of the parts, although it represents a part-of relationship semantic. That is, it is not specified if the *TargetNode* of a *Reference* of the *HasComponent ReferenceType* is deleted when the *SourceNode* is deleted.

The *TargetNode* of this *ReferenceType* shall be a *Variable*, an *Object* or a *Method*.

If the *TargetNode* is a *Variable*, the *SourceNode* shall be an *Object*, an *ObjectType*, a *DataVariable* or a *VariableType*. By using the *HasComponent Reference*, the *Variable* is defined as *DataVariable*.

If the *TargetNode* is an *Object* or a *Method*, the *SourceNode* shall be an *Object* or *ObjectType*.

7.8 HasProperty ReferenceType

The *HasProperty ReferenceType* is a concrete *ReferenceType* that can be used directly. It is a subtype of the *Aggregates ReferenceType*.

The semantic is to identify the *Properties* of a *Node*. *Properties* are described in 4.4.2.

The *SourceNode* of this *ReferenceType* can be of any *NodeClass*. The *TargetNode* shall be a *Variable*. By using the *HasProperty Reference*, the *Variable* is defined as *Property*. Since *Properties* shall not have *Properties*, a *Property* shall never be the *SourceNode* of a *HasProperty Reference*.

7.9 HasOrderedComponent ReferenceType

The *HasOrderedComponent ReferenceType* is a concrete *ReferenceType* that can be used directly. It is a subtype of the *HasComponent ReferenceType*.

The semantic of the *HasOrderedComponent ReferenceType* – besides the semantic of the *HasComponent ReferenceType* – is that when browsing from a *Node* and following *References* of this type or its subtype all *References* are returned in the Browse *Service* defined in IEC 62541-4 in a well-defined order. The order is *Server*-specific, but the *Client* can assume that the *Server* always returns them in the same order.

There are no additional constraints defined for this *ReferenceType*.

7.10 HasSubtype ReferenceType

The *HasSubtype ReferenceType* is a concrete *ReferenceType* that can be used directly. It is a subtype of the *HasChild ReferenceType*.

The semantic of *this ReferenceType* is to express a subtype relationship of types. It is used to span the *ReferenceType* hierarchy, whose semantic is specified in 5.3.3.3; a *DataType* hierarchy is specified in 5.8.3, and other subtype hierarchies are specified in Clause 6.

The *SourceNode* of *References* of this type shall be an *ObjectType*, a *VariableType*, a *DataType* or a *ReferenceType* and the *TargetNode* shall be of the same *NodeClass* as the *SourceNode*. Each *ReferenceType* shall be the *TargetNode* of at most one *Reference* of type *HasSubtype*.

7.11 Organizes ReferenceType

The *Organizes ReferenceType* is a concrete *ReferenceType* and can be used directly. It is a subtype of *HierarchicalReferences*.

The semantic of this *ReferenceType* is to organize *Nodes* in the *AddressSpace*. It can be used to span multiple hierarchies independent of any hierarchy created with the non-looping *Aggregates References*.

The *SourceNode* of *References* of this type shall be an *Object* or a *View.* If it is an *Object* then it should be an *Object* of the *ObjectType FolderType* or one of its subtypes (see 5.5.3).

The *TargetNode* of this *ReferenceType* can be of any *NodeClass*.

7.12 HasModellingRule ReferenceType

The *HasModellingRule ReferenceType* is a concrete *ReferenceType* and can be used directly. It is a subtype of *NonHierarchicalReferences*.

The semantic of this *ReferenceType* is to bind the *ModellingRule* to an *Object*, *Variable* or *Method*. The *ModellingRule* mechanisms are described in 6.4.4.

The *SourceNode* of this *ReferenceType* shall be an *Object*, *Variable* or *Method*. The *TargetNode* shall be an *Object* of the *ObjectType* “ModellingRule” or one of its subtypes.

Each *Node* shall be the *SourceNode* of at most one *HasModellingRule Reference*.

7.13 HasTypeDefinition ReferenceType

The *HasTypeDefinition ReferenceType* is a concrete *ReferenceType* and can be used directly. It is a subtype of *NonHierarchicalReferences*.

The semantic of this *ReferenceType* is to bind an *Object* or *Variable* to its *ObjectType* or *VariableType*, respectively. The relationships between types and instances are described in 4.5.

The *SourceNode* of this *ReferenceType* shall be an *Object* or *Variable*. If the *SourceNode* is an *Object*, then the *TargetNode* shall be an *ObjectType*; if the *SourceNode* is a *Variable*, then the *TargetNode* shall be a *VariableType*.

Each *Variable* and each *Object* shall be the *SourceNode* of exactly one *HasTypeDefinition Reference*.

7.14 HasEncoding ReferenceType

The *HasEncoding ReferenceType* is a concrete *ReferenceType* and can be used directly. It is a subtype of *NonHierarchicalReferences*.

The semantic of this *ReferenceType* is to reference *DataTypeEncodings* of a subtype of the *Structure DataType*.

The *SourceNode* of *References* of this type shall be a subtype of the *Structure DataType*.

The *TargetNode* of this *ReferenceType* shall be an *Object* of the *ObjectType DataTypeEncodingType* or one of its subtypes (see 5.8.4).

7.15 GeneratesEvent

The *GeneratesEvent ReferenceType* is a concrete *ReferenceType* and can be used directly. It is a subtype of *NonHierarchicalReferences*.

The semantic of this *ReferenceType* is to identify the types of *Events* instances of *ObjectTypes* or *VariableTypes* may generate and *Methods* may generate on each *Method* call.

The *SourceNode* of *References* of this type shall be an *ObjectType*, a *VariableType* or a *Method*.

The *TargetNode* of this *ReferenceType* shall be an *ObjectType* representing *EventTypes*, that is, the *BaseEventType* or one of its subtypes.

7.16 AlwaysGeneratesEvent

The *AlwaysGeneratesEvent ReferenceType* is a concrete *ReferenceType* and can be used directly. It is a subtype of *GeneratesEvent*.

The semantic of this *ReferenceType* is to identify the types of *Events Methods* shall generate on each *Method* call.

The *SourceNode* of *References* of this type shall be a *Method*.

The *TargetNode* of this *ReferenceType* shall be an *ObjectType* representing *EventTypes*, that is, the *BaseEventType* or one of its subtypes.

7.17 HasEventSource

The *HasEventSource ReferenceType* is a concrete *ReferenceType* and can be used directly. It is a subtype of *HierarchicalReferences*.

The semantic of this *ReferenceType* is to relate event sources in a hierarchical, non-looping organization. This *ReferenceType* and any subtypes are intended to be used for discovery of *Event* generation in a *Server*. They are not required to be present for a *Server* to generate an *Event* from its source (causing the *Event*) to its notifying *Nodes*. In particular, the root notifier of a *Server*, the *Server Object* defined in IEC 62541-5, is always capable of supplying all *Events* from a *Server* and as such has implied *HasEventSource References* to every event source in a *Server*.

The *SourceNode* of this *ReferenceType* shall be an *Object* that is a source of event subscriptions. A source of event subscriptions is an *Object* that has its “SubscribeToEvents” bit set within the *EventNotifier Attribute*.

The *TargetNode* of this *ReferenceType* can be a *Node* of any *NodeClass* that can generate event notifications via a subscription to the reference source.

Starting from *Node* “A” and only following *References* of the *HasEventSource ReferenceType* or of its subtypes it shall never be possible to return to “A”. But it is permitted that, following the *References*, there may be more than one path leading to another *Node* “B”.

7.18 HasNotifier

The *HasNotifier ReferenceType* is a concrete *ReferenceType* and can be used directly. It is a subtype of *HasEventSource*.

The semantic of this *ReferenceType* is to relate *Object Nodes* that are notifiers with other notifier *Object Nodes*. The *ReferenceType* is used to establish a hierarchical organization of event notifying *Objects*. It is a subtype of the *HasEventSource ReferenceType* defined in 7.17.

The *SourceNode* of this *ReferenceType* shall be *Objects* or *Views* that are a source of event subscriptions. The *TargetNode* of this *ReferenceType* shall be *Objects* that are a source of event subscriptions. A source of event subscriptions is an *Object* that has its "SubscribeToEvents" bit set within the *EventNotifier Attribute*.

If the *TargetNode* of a *Reference* of this type generates an *Event*, then this *Event* shall also be provided in the *SourceNode* of the *Reference*.

An example of a possible organization of *Event References* is represented in Figure 27. In this example an unfiltered *Event* subscription directed to the "Pump" *Object* will provide the *Event* sources "Start" and "Stop" to the subscriber. An unfiltered *Event* subscription directed to the "Area 1" *Object* will provide *Event* sources from "Machine B", "Tank A" and all notifier sources below "Tank A".

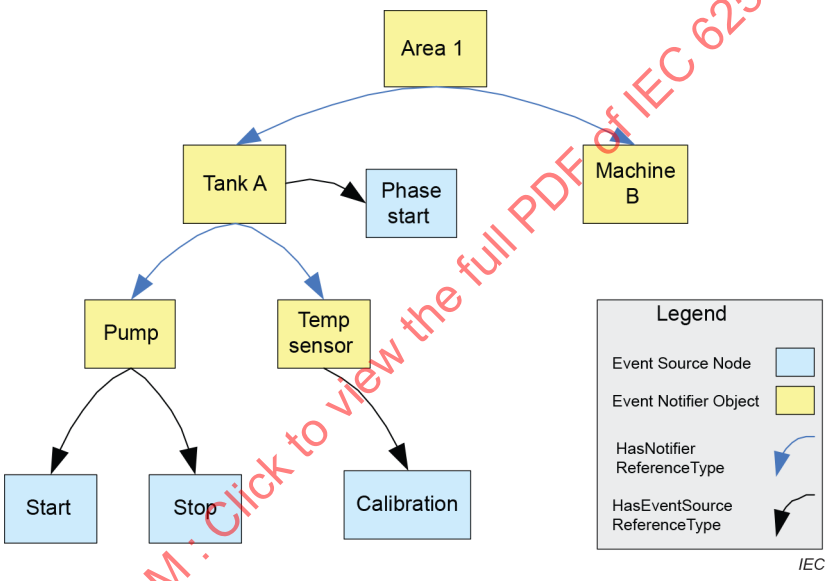


Figure 27 – Event Reference Example

A second example of a more complex organization of *Event References* is represented in Figure 28. In this example, explicit *References* are included from the *Server*'s *Server Object*, which is a source of all *Server Events*. A second *Event* organization has been introduced to collect the *Events* related to "Tank Farm 1". An unfiltered *Event* subscription directed to the "Tank Farm 1" *Object* will provide *Event* sources from "Tank B", "Tank A" and all notifier sources below "Tank B" and "Tank A".

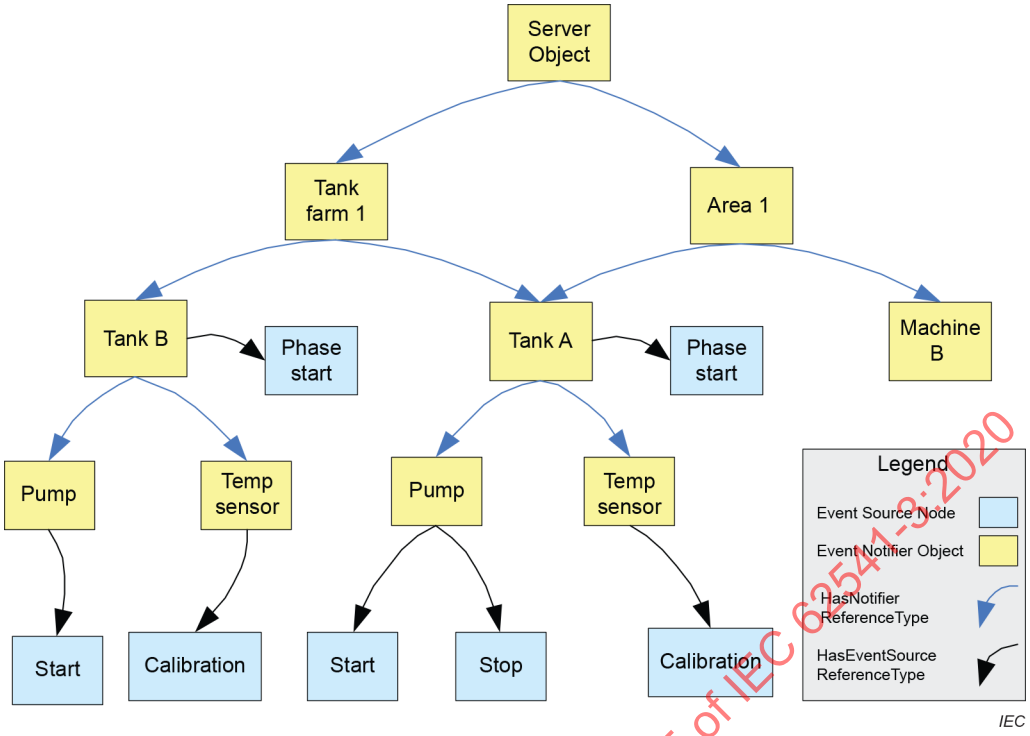


Figure 28 – Complex Event Reference Example

8 Standard DataTypes

8.1 General

The remainder of Clause 8 defines *DataTypes*. Their representation in the *AddressSpace* and the *DataType* hierarchy is specified in IEC 62541-5. Other parts of IEC 62541 may specify additional *DataTypes*.

8.2 NodeId

8.2.1 General

This *Built-in DataType* is composed of three elements that identify a *Node* within a *Server*. They are defined in Table 22.

Table 22 – NodeId Definition

Name	Type	Description
NodeId	structure	
namespaceIndex	UInt16	The index for a namespace URI (see 8.2.2).
identifierType	Enum	The format and data type of the identifier (see 8.2.3).
identifier	*	The identifier for a *Node* in the *AddressSpace* of an OPC UA *Server* (see 8.2.4).

See IEC 62541-6 for a description of the encoding of the identifier into OPC UA Messages.

8.2.2 NamespaceIndex

The namespace is a URI that identifies the naming authority responsible for assigning the identifier element of the *NodeId*. Naming authorities include the local *Server*, the underlying

system, standards bodies and consortia. It is expected that most *Nodes* will use the URI of the *Server* or of the underlying system.

Using a namespace URI allows multiple OPC UA *Servers* attached to the same underlying system to use the same identifier to identify the same *Object*. This enables *Clients* that connect to those *Servers* to recognise *Objects* that they have in common.

Namespace URIs, like *Server* names, are identified by numeric values in OPC UA *Services* to permit more efficient transfer and processing (e.g. table lookups). The numeric values used to identify namespaces correspond to the index into the *NamespaceArray*. The *NamespaceArray* is a *Variable* that is part of the *Server Object* in the *AddressSpace* (see IEC 62541-5 for its definition).

The URI for the OPC UA namespace is:

"http://opcfoundation.org/UA/"

Its corresponding index in the namespace table is 0.

The namespace URI is case sensitive.

8.2.3 IdentifierType

The IdentifierType element identifies the type of the *NodeId*, its format and its scope. Its values are defined in Table 23.

Table 23 – IdentifierType Values

Value	Description
NUMERIC_0	Numeric value
STRING_1	String value
GUID_2	Globally Unique Identifier
OPAQUE_3	Namespace specific format

Normally the scope of *NodeIds* is the *Server* in which they are defined. For certain types of *NodeIds*, *NodeIds* can uniquely identify a *Node* within a system, or across systems (e.g. GUIDs). System-wide and globally-unique identifiers allow *Clients* to track *Nodes*, such as work orders, as they move between OPC UA *Servers* as they progress through the system.

Opaque identifiers are identifiers that are free-format byte strings that might or might not be human interpretable.

String identifiers are case sensitive. That is, *Clients* shall consider them case sensitive. *Servers* are allowed to provide alternative *NodeIds* (see 5.2.2) and using this mechanism severs can handle *NodeIds* as case insensitive.

8.2.4 Identifier value

The identifier value element is used within the context of the first three elements to identify the *Node*. Its data type and format is defined by the IdType.

Identifier values of IdType STRING_1 are restricted to 4 096 characters. Identifier values of IdType OPAQUE_3 are restricted to 4 096 bytes.

A null *NodeId* has special meaning. For example, many services defined in IEC 62541-4 define special behaviour if a null *NodeId* is passed as a parameter. Each IdType has a set of identifier values that represent a null *NodeId*. These values are summarized in Table 24.

Table 24 – NodeId Null Values

IdType	Identifier
NUMERIC_0	0
STRING_1	A null or Empty String (“”)
GUID_2	A Guid initialised with zeros (e.g. 00000000-0000-0000-0000-000000)
OPAQUE_3	A ByteString with Length=0

A null *NodeId* always has a NamespaceIndex equal to 0.

A *Node* in the *AddressSpace* shall not have a null as its *NodeId*.

8.3 QualifiedName

This *Built-in DataType* contains a qualified name. It is, for example, used as *BrowseName*. Its elements are defined in Table 25. The name part of the *QualifiedName* is restricted to 512 characters.

Table 25 – QualifiedName Definition

Name	Type	Description
QualifiedName	structure	
namespaceIndex	UInt16	Index that identifies the namespace that defines the name. This index is the index of that namespace in the local *Server*'s *NamespaceArray*. The *Client* may read the *NamespaceArray Variable* to access the string value of the namespace.
name	String	The text portion of the QualifiedName.

8.4 LocaleId

This *Simple DataType* is specified as a string that is composed of a language component and a country/region component as specified by IETF RFC 5646. The <country/region> component is always preceded by a hyphen. The format of the *LocaleId* string is shown below:

<language>[-<country/region>], where
<language> is the two letter ISO 639 code for a language,
<country/region> is the two letter ISO 3166 code for the country/region.

The rules for constructing *LocaleIds* defined by IETF RFC 5646 are restricted as follows:

a) this specification permits only zero or one <country/region> component to follow the <language> component;

b) this specification also permits the “-CHS” and “-CHT” three-letter <country/region> codes for “Simplified” and “Traditional” Chinese locales;

c) this specification also allows the use of other <country/region> codes as deemed necessary by the *Client* or the *Server*.

Table 26 shows examples of OPC UA *LocaleIds*. *Clients* and *Servers* always provide *LocaleIds* that explicitly identify the language and the country/region.

Table 26 – LocaleId Examples

Locale	OPC UA LocaleId
English	en
English (US)	en-US
German	de
German (Germany)	de-DE
German (Austrian)	de-AT

An empty or null string indicates that the LocaleId is unknown.

8.5 LocalizedText

This *Built-in DataType* defines a structure containing a String in a locale-specific translation specified in the identifier for the locale. Its elements are defined in Table 27.

Table 27 – LocalizedText Definition

Name	Type	Description
LocalizedText	structure	
locale	LocaleId	The identifier for the locale (e.g. “en-US”).
text	String	The localized text.

8.6 Argument

This *Structured DataType* defines a *Method* input or output argument specification. It is for example used in the input and output argument *Properties* for *Methods*. Its elements are described in Table 28.

Table 28 – Argument Definition

Name	Type	Description
Argument	structure	
name	String	The name of the argument.
dataType	NodeId	The *NodeId* of the *DataType* of this argument.
valueRank	Int32	Indicates whether the *dataType* is an array and how many dimensions the array has. It may have the following values: *n* > 1: the dataType is an array with the specified number of dimensions. OneDimension (1): The dataType is an array with one dimension. OneOrMoreDimensions (0): The dataType is an array with one or more dimensions. Scalar (−1): The dataType is not an array. Any (−2): The dataType can be a scalar or an array with any number of dimensions. ScalarOrOneDimension (−3): The dataType can be a scalar or a one dimensional array. NOTE All DataTypes are considered to be scalar, even if they have array-like semantics like ByteString and String.
arrayDimensions	UInt32[]	This field specifies the maximum supported length of each dimension. If the maximum is unknown the value shall be 0. The number of elements shall be equal to the value of the v*alueRank field*. This field shall be null if v*alueRank* ≤ 0. The maximum number of elements of an array transferred on the wire is 2147483647 (max Int32).
description	LocalizedText	A localized description of the argument.

8.7 BaseDataType

This abstract *DataType* defines a value that can have any valid *DataType*.

It defines a special value null indicating that a value is not present.

8.8 Boolean

This *Built-in DataType* defines a value that is either TRUE or FALSE.

8.9 Byte

This *Built-in DataType* defines a value in the range of 0 to 255.

8.10 ByteString

This *Built-in DataType* defines a value that is a sequence of Byte values.

8.11 DateTime

This *Built-in DataType* defines a Gregorian calendar date. Details about this *DataType* are defined in IEC 62541-6.

8.12 Double

This *Built-in DataType* defines a value that adheres to the ISO/IEC/IEEE 60559:2011 double precision data type definition.

8.13 Duration

This *Simple DataType* is a *Double* that defines an interval of time in milliseconds (fractions can be used to define sub-millisecond values). Negative values are generally invalid but may have special meanings where the *Duration* is used.

8.14 Enumeration

This abstract *DataType* is the base *DataType* for all enumeration *DataTypes* like *NodeClass* defined in 8.30. All *DataTypes* inheriting from this *DataType* have special handling for the encoding as defined in IEC 62541-6. All enumeration *DataTypes* shall inherit from this *DataType*.

Some special rules apply when subtyping enumerations. Any enumeration *DataType* not directly inheriting from the *Enumeration DataType* can only restrict the enumeration values of its supertype. That is, it shall neither add enumeration values nor change the text associated to the enumeration value. As an example, the enumeration Days having {'Mo', 'Tu', 'We', 'Th', 'Fr', 'Sa', 'Su'} as values can be subtyped to the enumeration Workdays having {'Mo', 'Tu', 'We', 'Th', 'Fr'}. The other direction, subtyping Workdays to Days would not be allowed as Days has values not allowed by Workdays ('Sa' and 'Su').

8.15 Float

This *Built-in DataType* defines a value that adheres to the ISO/IEC/IEEE 60559:2011 single precision data type definition.

8.16 Guid

This *Built-in DataType* defines a value that is a 128-bit Globally Unique Identifier. Details about this *DataType* are defined in IEC 62541-6.

8.17 SByte

This *Built-in DataType* defines a value that is a signed integer between −128 and 127 inclusive.

8.18 IdType

This *DataType* is an enumeration that identifies the IdType of a *NodeId*. Its values are defined in Table 23. See 8.2.3 for a description of the use of this *DataType* in *NodeIds*.

8.19 Image

This abstract *DataType* defines a *ByteString* representing an image.

8.20 ImageBMP

This *Simple DataType* defines a *ByteString* representing an image in BMP format.

8.21 ImageGIF

This *Simple DataType* defines a *ByteString* representing an image in GIF format.

8.22 ImageJPG

This *Simple DataType* defines a *ByteString* representing an image in JPG format.

8.23 ImagePNG

This *Simple DataType* defines a *ByteString* representing an image in PNG format.

8.24 Integer

This abstract *DataType* defines an integer whose length is defined by its subtypes.

8.25 Int16

This *Built-in DataType* defines a value that is a signed integer between −32 768 and 32 767 inclusive.

8.26 Int32

This *Built-in DataType* defines a value that is a signed integer between −2 147 483 648 and 2 147 483 647 inclusive.

8.27 Int64

This *Built-in DataType* defines a value that is a signed integer between −9 223 372 036 854 775 808 and 9 223 372 036 854 775 807 inclusive.

8.28 TimeZoneDataType

This *Structured DataType* defines the local time that may or may not take daylight saving time into account. Its elements are described in Table 29.

Table 29 – TimeZoneDataType Definition

Name	Type	Description
TimeZoneDataType	structure	
offset	Int16	The offset in minutes from UtcTime
daylightSavingInOffset	Boolean	If TRUE, then daylight saving time (DST) is in effect and *offset* includes the DST correction. If FALSE then the *offset* does not include the DST correction and DST may or may not have been in effect.

8.29 NamingRuleType

This *DataType* is an enumeration that identifies the *NamingRule* (see 6.4.4.2). Its values are defined in Table 30.

Table 30 – NamingRuleType Values

Name
MANDATORY_1
OPTIONAL_2
CONSTRAINT_3

8.30 NodeClass

This *DataType* is an enumeration that identifies a *NodeClass*. Its values are defined in Table 31.

Table 31 – NodeClass Values

Name
OBJECT_1
VARIABLE_2
METHOD_4
OBJECT_TYPE_8
VARIABLE_TYPE_16
REFERENCE_TYPE_32
DATA_TYPE_64
VIEW_128

8.31 Number

This abstract *DataType* defines a number. Details are defined by its subtypes.

8.32 String

This *Built-in DataType* defines a Unicode character string that should exclude control characters that are not whitespaces.

8.33 Structure

This abstract *DataType* is the base *DataType* for all *Structured DataTypes* like *Argument* defined in 8.6. All *DataTypes* inheriting from this *DataType* have special handling for the encoding as defined in IEC 62541-6.

8.34 UInteger

This abstract *DataType* defines an unsigned integer whose length is defined by its subtypes.

8.35 UInt16

This *Built-in DataType* defines a value that is an unsigned integer between 0 and 65 535 inclusive.

8.36 UInt32

This *Built-in DataType* defines a value that is an unsigned integer between 0 and 4 294 967 295 inclusive.

8.37 UInt64

This *Built-in DataType* defines a value that is an unsigned integer between 0 and 18 446 744 073 709 551 615 inclusive.

8.38 UtcTime

This *simple DataType* is a *DateTime* used to define Coordinated Universal Time (UTC) values. All time values conveyed between OPC UA *Servers* and *Clients* are UTC values. *Clients* shall provide any conversions between UTC and local time.

UTC has the concept of leap seconds. Leap seconds can lead to repeating seconds. Therefore applications are allowed to use TAI (International Atomic Time) instead of UTC in any place where UtcTime is used. Details on time synchronization are discussed in IEC 62541-6.

It should be noted that the Source and Server Timestamps may originate from different clocks that have no synchronization. It is also possible that one may use UTC while the other uses TAI.

8.39 XmlElement

This *Built-in DataType* is used to define XML elements. IEC 62541-6 defines details about this *DataType*.

XML data can always be modelled as a subtype of the *Structure DataType* with a single *DataTypeEncoding* that represents the XML complexType that defines the XML element (it is not necessary to have access to the XML Schema to define a *DataTypeEncoding*). For this reason a *Server* should never define *Variables* that use the *XmlElement DataType* unless the *Server* has no information about the XML elements that might be in the *Variable Value*.

8.40 EnumValueType

This *Structured DataType* is used to represent a human-readable representation of an Enumeration. Its elements are described in Table 32. When this type is used in an array representing human-readable representations of an enumeration, each Value shall be unique in that array.

Table 32 – EnumValueType Definition

Name	Type	Description
EnumValueType	structure	
value	Int64	The Integer representation of an Enumeration.
displayName	LocalizedText	A human-readable representation of the Value of the Enumeration.
description	LocalizedText	A localized description of the enumeration value. This field can contain an empty string if no description is available.

Note that the *EnumValueType* has been defined with an Int64 Value to meet a variety of usages. When it is used to define the string representation of an Enumeration *DataType*, the value range is limited to Int32, because the Enumeration *DataType* is a subtype of Int32. IEC 62541-8 specifies other usages where the actual value might be between 8 Bit and 64 Bit.

8.41 OptionSet

This abstract *DataType* is the base *DataType* for all *DataTypes* representing a bit mask. All *OptionSet DataTypes* representing bit masks shall inherit from this *DataType*. Its elements are described in Table 33.

Table 33 – OptionSet Definition

Name	Type	Description
OptionSet	structure	
value	ByteString	Array of bytes representing the bits in the option set. The length of the ByteString depends on the number of bits. The number of bytes may be larger than needed for the valid bits in the case of a spare allocation.
validBits	ByteString	Array of bytes with same size as value representing the valid bits in the value parameter. When the *Server* returns the value to the *Client*, the *validBits* provides information of which bits in the bit mask have a meaning. If a bit is 1 then the corresponding bit in the value is used by the Server. If it is set to a 0 it should be ignored as it has no meaning. When the *Client* passes the value to the *Server*, the *validBits* defines which bits should be written. Only those bits defined in *validBits* are changed in the bit mask, all others are not written.

The *DataType Nodes* representing concrete subtypes of the *OptionSet* shall have an *OptionSetValues Property* defined in Table 16.

8.42 Union

This abstract *DataType* is the base *DataType* for all union *DataTypes*. The *DataType* is a subtype of *Structure DataType*. All *DataTypes* inheriting from this *DataType* have special handling for the encoding as defined in IEC 62541-6. All union *DataTypes* shall inherit directly from this *DataType*.

8.43 DateString

This *Simple DataType* defines a value which is a day in the Gregorian calendar in string. Lexical representation of the string shall conform to calendar date defined in ISO 8601.

NOTE 1 According to ISO 8601, 'calendar date representations are in the form [YYYY-MM-DD]. [YYYY] indicates a four-digit year, 0000 through 9999. [MM] indicates a two-digit month of the year, 01 through 12. [DD] indicates a two-digit day of that month, 01 through 31. For example, "the 5th of April 1981" may be represented as either "1981-04-05" in the extended format or "19810405" in the basic format.'

NOTE 2 ISO 8601 also allows for calendar dates to be written with reduced precision. For example, one may write "1981-04" to mean "1981 April", and one may simply write "1981" to refer to that year or "19" to refer to the century from 1900 to 1999 inclusive.

NOTE 3 Although ISO 8601 allows both the YYYY-MM-DD and YYYYMMDD formats for complete calendar date representations, if the day [DD] is omitted then only the YYYY-MM format is allowed. By disallowing dates of the form YYYYMM, ISO 8601 avoids confusion with the truncated representation YYMMDD (still often used).

8.44 DecimalString

This *Simple DataType* defines a value that represents a decimal number as a string. Lexical representation of the string shall conform to decimal type defined in W3C XML Schema Definition Language (XSD) Part 2: DataTypes.

The *DecimalString* is a numeric string with an optional sign and decimal point.

8.45 DurationString

This *Simple DataType* defines a value that represents a duration of time as a string. It shall conform to duration as defined in ISO 8601.

NOTE According to ISO 8601, durations are represented by the format P[n]Y[n]M[n]DT[n]H[n]M[n]S or P[n]W as shown to the right. In these representations, the [n] is replaced by the value for each of the date and time elements that follow the [n]. Leading zeros are not required, but the maximum number of digits for each element should be agreed to by the communicating parties. The capital letters *P*, *Y*, *M*, *W*, *D*, *T*, *H*, *M*, and *S* are designators for each of the date and time elements and are not replaced.

- *P* is the duration designator (historically called "period") placed at the start of the duration representation.
- *Y* is the year designator that follows the value for the number of years.
- *M* is the month designator that follows the value for the number of months.
- *W* is the week designator that follows the value for the number of weeks.
- *D* is the day designator that follows the value for the number of days.
- *T* is the time designator that precedes the time components of the representation.
- *H* is the hour designator that follows the value for the number of hours.
- *M* is the minute designator that follows the value for the number of minutes.
- *S* is the second designator that follows the value for the number of seconds.

For example, "P3Y6M4DT12H30M5S" represents a duration of "three years, six months, four days, twelve hours, thirty minutes, and five seconds". Date and time elements including their designator may be omitted if their value is zero, and lower order elements may also be omitted for reduced precision. For example, "P23DT23H" and "P4Y" are both acceptable duration representations.

8.46 NormalizedString

This *Simple DataType* defines a string value that shall be normalized according to Unicode Standard Annex #15, Normalization Form C.

NOTE Some Unicode characters have multiple equivalent binary representations consisting of sets of combining and/or composite Unicode characters. Unicode defines a process called normalization that returns one binary representation when given any of the equivalent binary representations of a character. The Win32 and the .NET Framework currently support normalization forms C, D, KC, and KD, as defined in Unicode Standard Annex #15. *NormalizedString* uses Normalization Form C for all content, because this form avoids potential interoperability problems caused by the use of canonically equivalent, yet different, character sequences in document formats.

8.47 TimeString

This *Simple DataType* defines a value that represents a time as a string. It shall conform to time of day as defined in ISO 8601.

NOTE ISO 8601 uses the 24-hour clock system. The *basic format* is [hh][mm][ss] and the *extended format* is [hh]:[mm]:[ss].

- [hh] refers to a zero-padded hour between 00 and 24 (where 24 is only used to denote midnight at the end of a calendar day).
- [mm] refers to a zero-padded minute between 00 and 59.
- [ss] refers to a zero-padded second between 00 and 60 (where 60 is only used to denote an added leap second).

Thus a time might appear as either "134730" in the *basic format* or "13:47:30" in the *extended format*.

It is also acceptable to omit lower order time elements for reduced accuracy: [hh]:[mm], [hh][mm] and [hh] are all used.

Midnight is a special case and can be referred to as both "00:00" and "24:00". The notation "00:00" is used at the beginning of a calendar day and is the more frequently used. "24:00" is used at the end of a day.

8.48 DataTypeDefinition

This abstract DataType is the base type for all *DataTypes* used to provide the metadata for custom *DataTypes* like *Structures and Enumerations*.

8.49 StructureDefinition

This *Structured DataType* is used to provide the metadata for a custom *Structure DataType*. It is derived from the *DataType DataTypeDefinition*. The *StructureDefinition* is formally defined in Table 34.

Table 34 – StructureDefinition Structure

Name	Type	Description
StructureDefinition	Structure	
defaultEncodingId	NodeId	The *NodeId* of the default DataTypeEncoding for the *DataType*. The default depends on the message encoding, Default Binary for UA Binary encoding, Default JSON for JSON encoding and Default XML for XML encoding. If the *DataType* is only used inside nested *Structures* and is not directly contained in an *ExtensionObject*, the encoding *NodeId* is null.
baseDataType	NodeId	The *NodeId* of the direct supertype of the *DataType*. This might be the abstract *Structure* or the *Union DataType*.
structureType	Enum StructureType	An enumeration that specifies the type of Structure defined by the *DataType*. It has the following values: Structure_0 A *Structure* without optional fields. StructureWithOptionalFields_1 A *Structure* with optional fields. Union_2 A *Union DataType* Only one of the fields defined for the data type is encoded into a value if the data type is a *Union*.
fields	StructureField []	The list of fields that make up the data type. This definition assumes the structure has a sequential layout. The StructureField *DataType* is defined in 8.51. For *Structures* derived from another *Structure DataType* this list shall begin with the fields of the baseDataType followed by the fields of this *StructureDefinition*.

8.50 EnumDefinition

This *Structured DataType* is used to provide the metadata for a custom *Enumeration* or *OptionSet* DataType. It is derived from the *DataType DataTypeDefinition*. The *EnumDefinition* is formally defined in Table 35.

Table 35 – EnumDefinition Structure

Name	Type	Description
EnumDefinition	Structure	
fields	EnumField []	The list of fields that make up the data type. The EnumField DataType is defined in 8.52.

8.51 StructureField

This *Structured DataType* is used to provide the metadata for a field of a custom *Structure DataType*. The *StructureField* is formally defined in Table 36.

Table 36 – StructureField Structure

Name	Type	Description
StructureField	Structure	
name	String	A name for the field that is unique within the *StructureDefinition*.
description	LocalizedText	A localized description of the field
dataType	NodeId	The *NodeId* of the *DataType* for the field.
valueRank	Int32	The value rank for the field. It shall be Scalar (-1) or a fixed rank Array (≥ 1).
arrayDimensions	UInt32[]	This field specifies the maximum supported length of each dimension. If the maximum is unknown the value shall be 0. The number of elements shall be equal to the value of the *valueRank field*. This field shall be null if *valueRank* ≤ 0. The maximum number of elements of an array transferred on the wire is 2147483647 (max Int32).
maxStringLength	UInt32	If the dataType field is a String or ByteString then this field specifies the maximum supported length. If the maximum is unknown the value shall be 0. If the dataType field is not a String or ByteString the value shall be 0. If the valueRank is greater than 0 this field applies to each element of the array.
isOptional	Boolean	The field indicates if a data type field in a *Structure* is optional. If the *structureType* is *Union_2* this field shall be ignored. If the *structureType* is *Structure_0* this field shall be false.

StructureFields can be exposed as *DataVariables* that are children of the *Variable* that contains the *Structure Value*. In this case the *BrowseName* of the *DataVariable* shall be the same as the *StructureField* name and the *NamespaceIndex* of the BrowseName shall be the same as the *Structure DataType Node NamespaceIndex*.

8.52 EnumField

This *Structured DataType* is used to provide the metadata for a field of a custom *Enumeration* or *OptionSet DataType*. It is derived from the *DataType EnumValueType*. If used for an *OptionSet,* the corresponding *Value* in the base type contains the number of the bit associated with the field. The *EnumField* is formally defined in Table 37.

Table 37 – EnumField Structure

Name	Type	Description
EnumField	Structure	
name	String	A name for the field that is unique within the *EnumDefinition*.

8.53 AudioDataType

This abstract *DataType* defines a *ByteString* representing audio data. The audio stored in the *ByteString* could be formats like WAV or MP3 or any number of other audio formats. These formats are self-describing as part of the *ByteString* and are not specified in this document.

8.54 Decimal

This *Simple DataType* defines a high-precision signed number. It consists of an arbitrary precision integer unscaled value and an integer scale. The scale is the inverse power of ten that is applied to the unscaled value.

8.55 PermissionType

This is a subtype of the *UInt32 DataType* with the *OptionSetValues Property* defined. It is used to define the permissions of a *Node*. The *PermissionType* is formally defined in Table 38.

Table 38 – PermissionType Definition

Name	Bit	Description
Browse	0	The *Client* is allowed to see the references to and from the *Node*. This implies that the *Client* is able to Read to *Attributes* other than the *Value* or the *RolePermissions Attribute*. This *Permission* is valid for all *NodeClasses*.
ReadRolePermissions	1	The *Client* is allowed to read the *RolePermissions Attribute*. This *Permission* is valid for all *NodeClasses*.
WriteAttribute	2	The *Client* is allowed to write to *Attributes* other than the *Value*, *Historizing* or *RolePermissions Attribute* if the *WriteMask* indicates that the *Attribute* is writable. This bit affects the value of a *UserWriteMask Attribute*. This *Permission* is valid for all *NodeClasses*.
WriteRolePermissions	3	The *Client* is allowed to write to the *RolePermissions Attribute* if the WriteMask indicates that the *Attribute* is writable. This bit affects the value of the *UserWriteMask Attribute*. This *Permission* is valid for all *NodeClasses*.
WriteHistorizing	4	The *Client* is allowed to write to the *Historizing Attributes* if the *WriteMask* indicates that the *Attribute* is writable. This bit affects the value of the *UserWriteMask Attribute*. This *Permission* is only valid for *Variables*.
Read	5	The *Client* is allowed to read the *Value Attribute*. This bit affects the *CurrentRead* bit of the *UserAccessLevel Attribute*. This *Permission* is only valid for *Variables*.
Write	6	The *Client* is allowed to write the *Value Attribute*. This bit affects the *CurrentWrite* bit of the *UserAccessLevel Attribute*. This *Permission* is only valid for *Variables*.
ReadHistory	7	The *Client* is allowed to read the history associated with a *Node*. This bit affects the *HistoryRead* bit of the *UserAccessLevel Attribute*. This *Permission* is only valid for *Variables*, *Objects* or *Views*.
InsertHistory	8	The *Client* is allowed to insert the history associated with a *Node*. This bit affects the *HistoryWrite* bit of the *UserAccessLevel Attribute*. This *Permission* is only valid for *Variables*, *Objects* or *Views*.
ModifyHistory	9	The *Client* is allowed to modify the history associated with a *Node*. This bit affects the *HistoryWrite* bit of the *UserAccessLevel Attribute*. This *Permission* is only valid for *Variables*, *Objects* or *Views*.
DeleteHistory	10	The *Client* is allowed to delete the history associated with a *Node*. This bit affects the *HistoryWrite* bit of the *UserAccessLevel Attribute*. This *Permission* is only valid for *Variables*, *Objects* or *Views*.
ReceiveEvents	11	A *Client* only receives an *Event* if this bit is set on the *Node* identified by the *EventTypeId* field and on the *Node* identified by the *SourceNode* field. This *Permission* is only valid for *EventType Nodes* or *SourceNodes*.

Name	Bit	Description
Call	12	The *Client* is allowed to call the *Method* if this bit is set on the *Object or ObjectType Node* passed in the *Call* request and the *Method Instance* associated with that *Object or ObjectType*. This bit affects the *UserExecutable Attribute* when set on *Method Node*. This *Permission* is only valid for *Objects, ObjectType* or *Methods*.
AddReference	13	The *Client* is allowed to add references to the *Node*. This *Permission* is valid for all *NodeClasses*.
RemoveReference	14	The *Client* is allowed to remove references from the *Node*. This *Permission* is valid for all *NodeClasses*.
DeleteNode	15	The *Client* is allowed to delete the *Node*. This *Permission* is valid for all *NodeClasses*.
AddNode	16	The *Client* is allowed to add *Nodes* to the *Namespace*. This *Permission* is only used in the *DefaultRolePermissions* and *DefaultUserRolePermissions Properties* of a *NamespaceMetadata Object*
Reserved	17-31	These bits are reserved for use by OPC UA.

8.56 AccessRestrictionsType

This is a subtype of the *UInt16 DataType* with the *OptionSetValues Property* defined. It is used to define the access restrictions of a *Node*. The *AccessRestrictionsType* is formally defined in Table 39.

Table 39 – AccessRestrictionsType Definition

Name	Bit	Description
SigningRequired	0	The *Client* can only access the *Node* when using a *SecureChannel* which digitally signs all messages. This does not apply to the *Browse* permission if the *ApplyRestrictionsToBrowse* is not set.
EncryptionRequired	1	The *Client* can only access the *Node* when using a *SecureChannel* which encrypts all messages. This does not apply to the *Browse* permission if the *ApplyRestrictionsToBrowse* is not set.
SessionRequired	2	The *Client* cannot access the *Node* when using *SessionlessInvoke Service* invocation.
ApplyRestrictionsToBrowse	3	The access restrictions *SigningRequired* and *EncryptionRequired* are also applied to for the *Browse* permission, if this bit is set.

8.57 AccessLevelType

This is a subtype of the *Byte DataType* with the *OptionSetValues Property* defined. It is used to indicate how the *Value* of a *Variable* can be accessed (read/write) and if it contains current and/or historic data. The *AccessLevelType* is formally defined in Table 40.

Table 40 – AccessLevelType Definition

Name	Bit	Description
CurrentRead	0	Indicates if the current value is readable. It also indicates if the current value of the *Variable* is available. (0 means not readable, 1 means readable).
CurrentWrite	1	Indicates if the current value is writable. It also indicates if the current value of the *Variable* is available (0 means not writable, 1 means writable).
HistoryRead	2	Indicates if the history of the value is readable. It also indicates if the history of the *Variable* is available via the OPC UA *Server.* (0 means not readable, 1 means readable).
HistoryWrite	3	Indicates if the history of the value is writable It also indicates if the history of the *Variable* is available via the OPC UA *Server.* (0 means not writable, 1 means writable).
SemanticChange	4	This flag is set for *Properties* that define semantic aspects of the parent *Node* of the *Property* and where the *Property Value*, and thus the semantic, may change during operation. (0 means is not a semantic, 1 means is a semantic).
StatusWrite	5	Indicates if the current *StatusCode* of the value is writable (0 means only *StatusCode* Good is writable, 1 means any *StatusCode* is writable).
TimestampWrite	6	Indicates if the current *SourceTimestamp* is writable (0 means only null timestamps are writable, 1 means any timestamp value is writable).
Reserved	7	Reserved for future use. Shall always be zero.

8.58 AccessLevelExType

This is a subtype of the *UInt32 DataType* with the *OptionSetValues Property* defined. It is used to indicate how the *Value* of a *Variable* can be accessed (read/write), if it contains current and/or historic data and its atomicity.

The *AccessLevelExType DataType* is an extended version of the *AccessLevelType DataType* and as such contains the 8 bits of the AccessLevelType as the first 8 bits.

The *NonatomicRead*, and *NonatomicWrite Fields* represent the atomicity of a *Variable*. In general Atomicity is expected of OPC UA read and write operations. These Fields are used by systems, in particular hard-realtime controllers, which can not ensure atomicity.

The *AccessLevelExType* is formally defined in Table 41.

Table 41 – AccessLevelExType Definition

Name	Bit	Description
	0:7	Formally defined by the AccessLevelType in Table 40.
NonatomicRead	8	Indicates non-atomicity for Read access (0 means that atomicity is assured).
NonatomicWrite	9	Indicates non-atomicity for Write access (0 means that atomicity is assured).
WriteFullArrayOnly	10	Indicates if Write of IndexRange is supported. (0 means Write of IndexRange is supported)
	11:31	Reserved for future use. Shall always be zero.

8.59 EventNotifierType

This is a subtype of the *Byte DataType* with the *OptionSetValues Property* defined. It is used to indicate if a Node can be used to subscribe to *Events* or read/write historic *Events*.

The *EventNotifierType* is formally defined in Table 42.

Table 42 – EventNotifierType Definition

Name	Bit	Description
SubscribeTo Events	0	Indicates if it can be used to subscribe to *Events* (0 means cannot be used to subscribe to *Events*, 1 means can be used to subscribe to *Events*).
	1	Reserved for future use. Shall always be zero.
HistoryRead	2	Indicates if the history of the *Events* is readable. (0 means not readable, 1 means readable).
HistoryWrite	3	Indicates if the history of the *Events* is writable. (0 means not writable, 1 means writable).
	4:7	Reserved for future use. Shall always be zero.

8.60 AttributeWriteMask

This is a subtype of the *UInt32 DataType* with the *OptionSetValues Property* defined. It is used to define the *Attribute* access restrictions of a *Node*. The *AttributeWriteMask* is formally defined in Table 43.

If a bit is set to 0, it means the *Attribute* is not writable. If a bit is set to 1, it means it is writable. If a *Node* does not support a specific *Attribute*, the corresponding bit shall be set to 0.

Table 43 – Bit mask for WriteMask and UserWriteMask

Field	Bit	Description
AccessLevel	0	Indicates if the AccessLevel Attribute is writable.
ArrayDimensions	1	Indicates if the ArrayDimensions Attribute is writable.
BrowseName	2	Indicates if the BrowseName Attribute is writable.
ContainsNoLoops	3	Indicates if the ContainsNoLoops Attribute is writable.
DataType	4	Indicates if the DataType Attribute is writable.
Description	5	Indicates if the Description Attribute is writable.
DisplayName	6	Indicates if the DisplayName Attribute is writable.
EventNotifier	7	Indicates if the EventNotifier Attribute is writable.
Executable	8	Indicates if the Executable Attribute is writable.
Historizing	9	Indicates if the Historizing Attribute is writable.
InverseName	10	Indicates if the InverseName Attribute is writable.
IsAbstract	11	Indicates if the IsAbstract Attribute is writable.
MinimumSamplingInterval	12	Indicates if the MinimumSamplingInterval Attribute is writable.
NodeClass	13	Indicates if the NodeClass Attribute is writable.
NodeId	14	Indicates if the NodeId Attribute is writable.
Symmetric	15	Indicates if the Symmetric Attribute is writable.
UserAccessLevel	16	Indicates if the UserAccessLevel Attribute is writable.
UserExecutable	17	Indicates if the UserExecutable Attribute is writable.
UserWriteMask	18	Indicates if the UserWriteMask Attribute is writable.
ValueRank	19	Indicates if the ValueRank Attribute is writable.
WriteMask	20	Indicates if the WriteMask Attribute is writable.
ValueForVariableType	21	Indicates if the *Value Attribute* is writable for a *VariableType*. It does not apply for *Variables* since this is handled by the *AccessLevel* and *UserAccessLevel Attributes* for the *Variable*. For *Variables* this bit shall be set to 0.
DataTypeDefinition	22	Indicates if the DataTypeDefinition Attribute is writable.
RolePermissions	23	Indicates if the RolePermissions Attribute is writable.
AccessRestrictions	24	Indicates if the AccessRestrictions Attribute is writable.
AccessLevelEx	25	Indicates if the AccessLevelEx Attribute is writable.
Reserved	26:31	Reserved for future use. Shall always be zero.

9 Standard EventTypes

9.1 General

The remainder of Clause 9 defines *EventTypes*. Their representation in the *AddressSpace* is specified in IEC 62541-5. Other parts of of IEC 62541 may specify additional *EventTypes*. Figure 29 informally describes the hierarchy of these *EventTypes*.

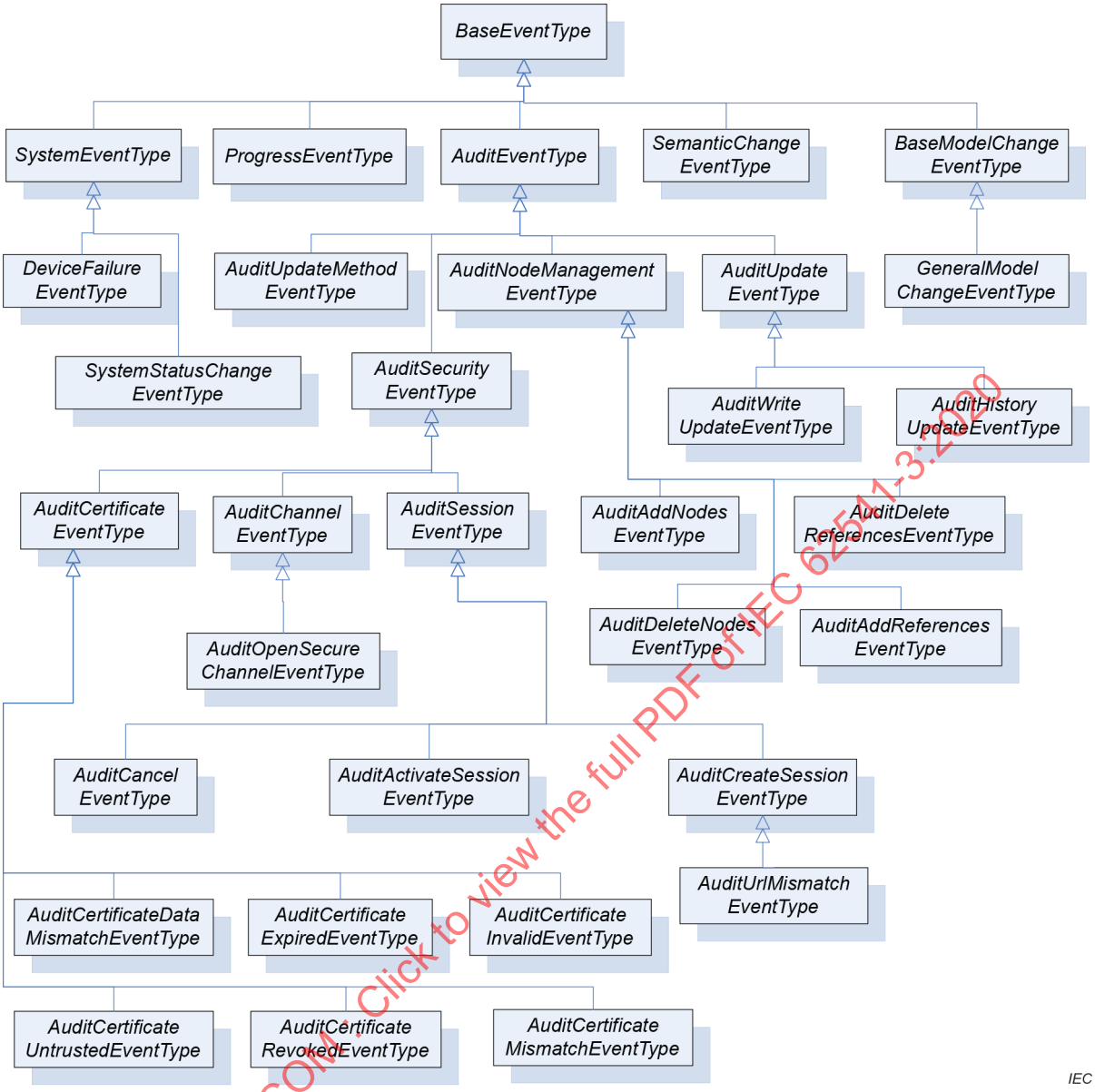


Figure 29 – Standard EventType Hierarchy

9.2 BaseEventType

The *BaseEventType* defines all general characteristics of an *Event*. All other *EventTypes* derive from it. There is no other semantic associated with this type.

9.3 SystemEventType

SystemEvents are *Events* of *SystemEventType* that are generated as a result of some *Event* that occurs within the *Server* or by a system that the *Server* is representing.

9.4 ProgressEventType

ProgressEvents are *Events* of *ProgressEventType* that are generated to identify the progress of an operation. An operation can be a service call or something application specific like a program execution.

9.5 AuditEventType

AuditEvents are *Events* of *AuditEventType* that are generated as a result of an action taken on the *Server* by a *Client* of the *Server*. For example, in response to a *Client* issuing a write to a *Variable*, the *Server* would generate an *AuditEvent* describing the *Variable* as the source and the user and *Client* session as the initiators of the *Event*.

Figure 30 illustrates the defined behaviour of an OPC UA *Server* in response to an auditable action request. If the action is accepted, then an action *AuditEvent* is generated and processed by the *Server*. If the action is not accepted due to security reasons, a security *AuditEvent* is generated and processed by the *Server*. The *Server* may involve the underlying device or system in the process but it is the *Server*'s responsibility to provide the *Event* to any interested *Clients*. *Clients* are free to subscribe to *Events* from the *Server* and will receive the *AuditEvents* in response to normal Publish requests.

All action requests include a human readable *AuditEntryId*. The *AuditEntryId* is included in the *AuditEvent* to allow human readers to correlate an *Event* with the initiating action. The *AuditEntryId* typically contains who initiated the action and from where it was initiated.

The *Server* may elect to optionally persist the *AuditEvents* in addition to the mandatory *Event Subscription* delivery to *Clients*.

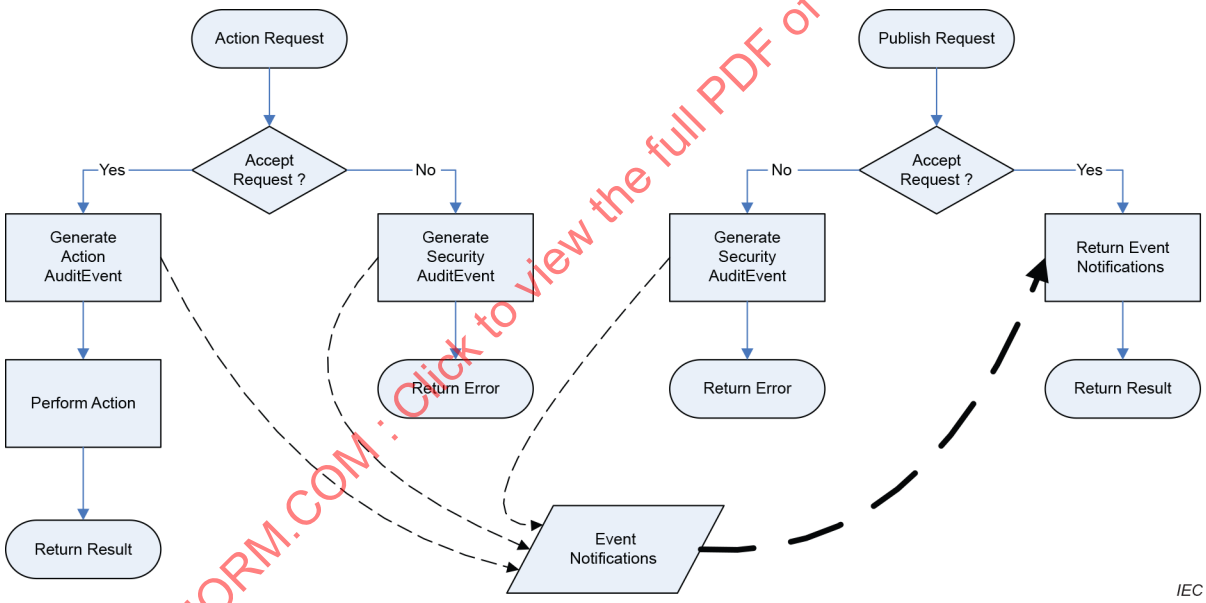


Figure 30 – Audit Behaviour of a Server

Figure 31 illustrates the expected behaviour of an aggregating *Server* in response to an auditable action request. This use case involves the aggregating *Server* passing on the action to one of its aggregated *Servers*. The general behaviour described above is extended by this behaviour and not replaced. That is, the request could fail and generate a security *AuditEvent* within the aggregating *Server*. The normal process is to pass the action down to an aggregated *Server* for processing. The aggregated *Server* will, in turn, follow this behaviour or the general behaviour and generate the appropriate *AuditEvents*. The aggregating *Server* periodically issues publish requests to the aggregated *Servers*. These collected *Events* are merged with self-generated *Events* and made available to subscribing *Clients*. If the aggregating *Server* supports the optional persisting of *AuditEvent*, then the collected *Events* are persisted along with locally-generated *Events*.

The aggregating *Server* may map the authenticated user account making the request to one of its own accounts when passing on the request to an aggregated *Server*. It shall, however, preserve the *AuditEntryId* by passing it on as received. The aggregating *Server* may also

generate its own *AuditEvent* for the request prior to passing it on to the aggregated *Server*, in particular, if the aggregating *Server* needs to break a request into multiple requests that are each directed to separate aggregated *Servers* or if part of a request is denied due to security on the aggregating *Server*.

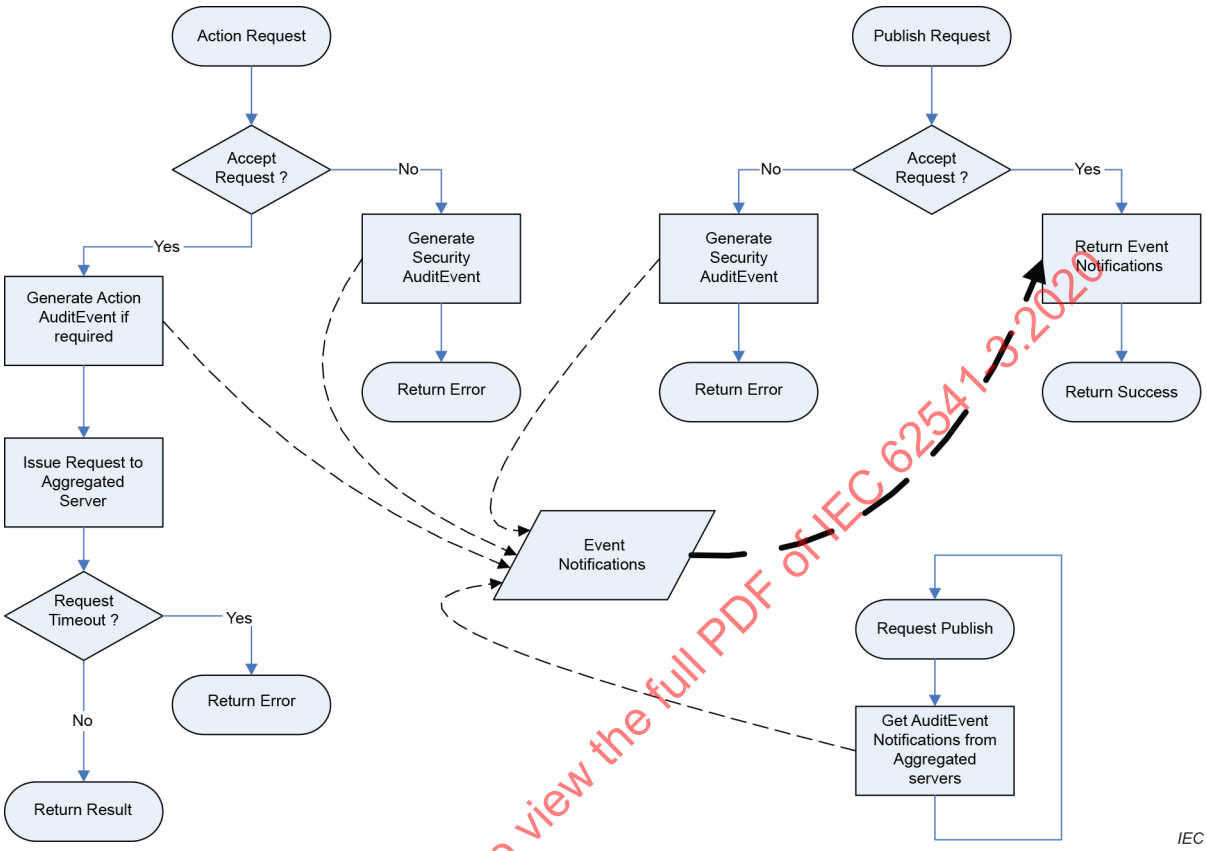


Figure 31 – Audit Behaviour of an Aggregating Server

9.6 AuditSecurityEventType

This is a subtype of *AuditEventType* and is used only for categorization of security-related *Events*. This type follows all behaviour of its parent type.

9.7 AuditChannelEventType

This is a subtype of *AuditSecurityEventType* and is used for categorization of security-related *Events* from the *SecureChannel Service Set* defined in IEC 62541-4.

9.8 AuditOpenSecureChannelEventType

This is a subtype of *AuditChannelEventType* and is used for *Events* generated from calling the OpenSecureChannel *Service* defined in IEC 62541-4.

9.9 AuditSessionEventType

This is a subtype of *AuditSecurityEventType* and is used for categorization of security-related *Events* from the *Session Service Set* defined in IEC 62541-4.

9.10 AuditCreateSessionEventType

This is a subtype of *AuditSessionEventType* and is used for *Events* generated from calling the CreateSession *Service* defined in IEC 62541-4.

9.11 AuditUrlMismatchEventType

This is a subtype of *AuditCreateSessionEventType* and is used for *Events* generated from calling the CreateSession *Service* defined in IEC 62541-4 if the EndpointUrl used in the service call does not match the *Server*'s *HostNames* (see IEC 62541-4 for details).

9.12 AuditActivateSessionEventType

This is a subtype of *AuditSessionEventType* and is used for *Events* generated from calling the ActivateSession *Service* defined in IEC 62541-4.

9.13 AuditCancelEventType

This is a subtype of *AuditSessionEventType* and is used for *Events* generated from calling the Cancel *Service* defined in IEC 62541-4.

9.14 AuditCertificateEventType

This is a subtype of *AuditSecurityEventType* and is used only for categorization of Certificate related *Events*. This type follows all behaviours of its parent type. These *AuditEvents* will be generated for Certificate errors in addition to other *AuditEvents* related to service calls.

9.15 AuditCertificateDataMismatchEventType

This is a subtype of *AuditCertificateEventType* and is used only for categorization of Certificate related *Events*. This type follows all behaviours of its parent type. This *AuditEvent* is generated if the HostName in the URL used to connect to the *Server* is not the same as one of the HostNames specified in the Certificate or if the Application and Software Certificates contain an application or product URI that does not match the URI specified in the ApplicationDescription provided with the Certificate. For more details on Certificates see IEC 62541-4.

9.16 AuditCertificateExpiredEventType

This is a subtype of *AuditCertificateEventType* and is used only for categorization of Certificate related *Events*. This type follows all behaviours of its parent type. This *AuditEvent* is generated if the current time is outside the validity period's start date and end date.

9.17 AuditCertificateInvalidEventType

This is a subtype of *AuditCertificateEventType* and is used only for categorization of Certificate related *Events*. This type follows all behaviours of its parent type. This *AuditEvent* is generated if the certificate structure is invalid or if the Certificate has an invalid signature.

9.18 AuditCertificateUntrustedEventType

This is a subtype of *AuditCertificateEventType* and is used only for categorization of Certificate related *Events*. This type follows all behaviours of its parent type. This *AuditEvent* is generated if the Certificate is not trusted, that is, if the Issuer Certificate is unknown.

9.19 AuditCertificateRevokedEventType

This is a subtype of *AuditCertificateEventType* and is used only for categorization of Certificate related *Events*. This type follows all behaviours of its parent type. This *AuditEvent* is generated if a Certificate has been revoked or if the revocation list is not available (i.e. a network interruption prevents the Application from accessing the list).

9.20 AuditCertificateMismatchEventType

This is a subtype of *AuditCertificateEventType* and is used only for categorization of Certificate related *Events*. This type follows all behaviours of its parent type. This *AuditEvent* is generated if a Certificate set of uses does not match the requested use for the Certificate (i.e. Application, Software or Certificate Authority).

9.21 AuditNodeManagementEventType

This is a subtype of *AuditEventType* and is used for categorization of node management related *Events*. This type follows all behaviours of its parent type.

9.22 AuditAddNodesEventType

This is a subtype of *AuditNodeManagementEventType* and is used for *Events* generated from calling the AddNodes *Service* defined in IEC 62541-4.

9.23 AuditDeleteNodesEventType

This is a subtype of *AuditNodeManagementEventType* and is used for *Events* generated from calling the DeleteNodes *Service* defined in IEC 62541-4.

9.24 AuditAddReferencesEventType

This is a subtype of *AuditNodeManagementEventType* and is used for *Events* generated from calling the AddReferences *Service* defined in IEC 62541-4.

9.25 AuditDeleteReferencesEventType

This is a subtype of *AuditNodeManagementEventType* and is used for *Events* generated from calling the DeleteReferences *Service* defined in IEC 62541-4.

9.26 AuditUpdateEventType

This is a subtype of *AuditEventType* and is used for categorization of update related *Events*. This type follows all behaviours of its parent type.

9.27 AuditWriteUpdateEventType

This is a subtype of *AuditUpdateEventType* and is used for categorization of write update related *Events*. This type follows all behaviours of its parent type.

9.28 AuditHistoryUpdateEventType

This is a subtype of *AuditUpdateEventType* and is used for categorization of history update related *Events*. This type follows all behaviours of its parent type.

9.29 AuditUpdateMethodEventType

This is a subtype of *AuditEventType* and is used for categorization of *Method* related *Events*. This type follows all behaviours of its parent type.

9.30 DeviceFailureEventType

A *DeviceFailureEvent* is an *Event* of *DeviceFailureEventType* that indicates a failure in a device of the underlying system.

9.31 SystemStatusChangeEventType

A *SystemStatusChangeEvent* is an *Event* of *SystemStatusChangeEventType* that indicates a status change in a system. For example, if the status indicates an underlying system is not running, then a *Client* cannot expect any *Events* from the underlying system. A *Server* can identify its own status changes using this *EventType*.

9.32 ModelChangeEvents

9.32.1 General

ModelChangeEvents are generated to indicate a change of the *AddressSpace* structure. The change may consist of adding or deleting a *Node* or *Reference*. Although the relationship of a *Variable* or *VariableType* to its *DataType* is not modelled using *References*, changes to the *DataType Attribute* of a *Variable* or *VariableType* are also considered as model changes and therefore a *ModelChangeEvent* is generated if the *DataType Attribute* changes.

9.32.2 NodeVersion Property

There is a correlation between *ModelChangeEvents* and the *NodeVersion Property* of *Nodes*. Every time a *ModelChangeEvent* is issued for a *Node*, its *NodeVersion* shall be changed, and every time the *NodeVersion* is changed, a *ModelChangeEvent* shall be generated. A *Server* shall support both the *ModelChangeEvent* and the *NodeVersion Property* or neither, but never only one of the two mechanisms.

This relation also implies that only those *Nodes* of the *AddressSpace* having a *NodeVersion* shall trigger a *ModelChangeEvent*. Other *Nodes* shall not trigger a *ModelChangeEvent*.

9.32.3 Views

A *ModelChangeEvent* is always generated in the context of a *View,* including the default *View* where the whole *AddressSpace* is considered. Therefore the only *Notifiers* which report the *ModelChangeEvents* are *View Nodes* and the *Server Object* representing the default *View*. Each action generating a *ModelChangeEvent* may lead to several *Events* since it may affect different *Views*. If, for example, a *Node* was deleted from the *AddressSpace*, and this *Node* was also contained in a View "A", there would be one *Event* having the *AddressSpace* as context and another having the View "A" as context. If a *Node* would only be removed from *View* "A", but still exists in the *AddressSpace*, it would generate only a *ModelChangeEvent* for *View* "A".

If a *Client* does not want to receive duplicates of changes then it shall use the filter mechanisms of the *Event* subscription to filter only for the default *View* and suppress the *ModelChangeEvents* having other *Views* as the context.

When a *ModelChangeEvent* is issued on a *View* and the *View* supports the *ViewVersion Property*, then the *ViewVersion* shall be updated.

9.32.4 Event compression

An implementation is not required to issue an *Event* for every update as it occurs. An OPC UA *Server* may be capable of grouping a series of transactions or simple updates into a larger unit. This series may constitute a logical grouping or a temporal grouping of changes. A single *ModelChangeEvent* may be issued after the last change of the series, to cover all of the changes. This is referred to as *Event compression*. A change in the *NodeVersion* and the *ViewVersion* may thus reflect a group of changes and not a single change.

9.32.5 BaseModelChangeEventType

BaseModelChangeEvents are *Events* of the *BaseModelChangeEventType*. The *BaseModelChangeEventType* is the base type for *ModelChangeEvents* and does not contain

information about the changes but only indicates that changes occurred. Therefore the *Client* shall assume that any or all of the *Nodes* may have changed.

9.32.6 GeneralModelChangeEventType

GeneralModelChangeEvents are *Events* of the *GeneralModelChangeEventType*. The *GeneralModelChangeEventType* is a subtype of the *BaseModelChangeEventType*. It contains information about the *Node* that was changed and the action that occurred to cause the *ModelChangeEvent* (e.g. add a *Node*, delete a *Node*, etc.). If the affected *Node* is a *Variable* or *Object*, then the *TypeDefinitionNode* is also present.

To allow *Event* compression, a *GeneralModelChangeEvent* contains an array of changes.

9.32.7 Guidelines for ModelChangeEvents

Two types of *ModelChangeEvents* are defined: the *BaseModelChangeEvent* that does not contain any information about the changes and the *GeneralModelChangeEvent* that identifies the changed *Nodes* via an array. The precision used depends on both the capability of the OPC UA *Server* and the nature of the update. An OPC UA *Server* may use either *ModelChangeEvent* type depending on circumstances. It may also define subtypes of these *EventTypes* adding additional information.

To ensure interoperability, the following guidelines for *Events* should be observed.

- If the array of the *GeneralModelChangeEvent* is present, then it should identify every *Node* that has changed since the preceding *ModelChangeEvent*.
- The OPC UA *Server* should emit exactly one *ModelChangeEvent* for an update or series of updates. It should not issue multiple types of *ModelChangeEvent* for the same update.
- Any *Client* that responds to *ModelChangeEvents* should respond to any *Event* of the *BaseModelChangeEventType* including its subtypes like the *GeneralModelChangeEventType*.

If a *Client* is not capable of interpreting additional information of the subtypes of the *BaseModelChangeEventType*, it should treat *Events* of these types the same way as *Events* of the *BaseModelChangeEventType*.

9.33 SemanticChangeEventType

9.33.1 General

SemanticChangeEvents are *Events* of *SemanticChangeEventType* that are generated to indicate a change of the *AddressSpace* semantics. The change consists of a change to the *Value Attribute* of a *Property*.

The *SemanticChangeEvent* contains information about the *Node* owning the *Property* that was changed. If this is a *Variable* or *Object*, the *TypeDefinitionNode* is also present.

The SemanticChange bit of the *AccessLevel Attribute* of a *Property* indicates whether changes of the *Property* value are considered for *SemanticChangeEvents* (see 5.6.2).

9.33.2 ViewVersion and NodeVersion Properties

The *ViewVersion* and *NodeVersion Properties* do not change due to the publication of a *SemanticChangeEvent*.

9.33.3 Views

SemanticChangeEvents are handled in the context of a *View* the same way as *ModelChangeEvents*. This is defined in 9.32.3.

9.33.4 Event compression

SemanticChangeEvents can be compressed the same way as *ModelChangeEvents*. This is defined in 9.32.4.

Annex A
(informative)

How to use the Address Space Model

A.1 Overview

Annex A points out some general considerations on how the Address Space Model can be used. Annex A is for information only, that is, each *Server* vendor can model its data in the appropriate way that fits its needs. However, it gives some hints the *Server* vendor may consider.

Typically OPC UA *Servers* will offer data provided by an underlying system like a device, a configuration database, an OPC COM *Server*, etc. Therefore the modelling of the data depends on the model of the underlying system as well as the requirements of the *Clients* accessing the OPC UA *Server*. It is also expected that companion specifications will be developed on top of OPC UA with additional rules on how to model the data. However, the remainder of Annex A will give some general considerations about the different concepts of OPC UA to model data and when they should be used, and when not.

IEC 62541-5:–, Annex A, provides an overview of the design decisions made when modelling the information about the *Server* defined in IEC 62541-5.

A.2 Type definitions

Type definitions should be used whenever it is expected that the type information may be used more than once in the same system or for interoperability between different systems supporting the same type definitions.

A.3 ObjectTypes

Subclause 5.5.1 states: "*Objects* are used to represent systems, system components, real-world objects, and software objects." Therefore *ObjectTypes* should be used if a type definition of those *ObjectTypes* is useful (see A.2).

From a more abstract point of view *Objects* are used to group *Variables* and other *Objects* in the *AddressSpace*. Therefore *ObjectTypes* should be used when some common structures/groups of *Objects* and/or *Variables* should be described. *Clients* can use this knowledge to program against the *ObjectType* structure and use the TranslateBrowsePathsToNodeIds *Service* defined in IEC 62541-4 on the instances.

Simple objects only having one value (e.g. a simple heat sensor) can also be modelled as *VariableTypes*. However, extensibility mechanisms should be considered (e.g. a complex heat sensor subtype could have several values) and whether that object should be exposed as an object in the *Client*'s GUI or just as a value. Whenever a modeller is in doubt as to which solution to use the *ObjectType* having one *Variable* should be preferred.

A.4 VariableTypes

A.4.1 General

VariableTypes are only used for *DataVariables*[1] and should be used when there are several *Variables* having the same semantic (e.g. set point). It is not necessary to define a *VariableType* that only reflects the *DataType* of a *Variable*, e.g. an “Int32VariableType”.

A.4.2 Properties or DataVariables

Besides the semantic differences of *Properties* and *DataVariables* described in Clause 4 there are also syntactical differences. A *Property* is identified by its *BrowseName*, that is, if *Properties* having the same semantic are used several times, they should always have the same *BrowseName*. The same semantic of *DataVariables* is captured in the *VariableType*.

If it is not clear which concept to use based on the semantic described in Clause 4, then the different syntax can help. The following points identify when it shall be a *DataVariable*.

- If it is a complex *Variable* or it should contain additional information in the form of *Properties*.
- If the type definition may be refined (subtyping).
- If the type definition should be made available so the *Client* can use the AddNodes *Service* defined in IEC 62541-4 to create new instances of the type definition.
- If it is a component of a complex *Variable* exposing a part of the value of the complex *Variable*.

A.4.3 Many Variables and/or structured DataTypes

When structured data structures should be made available to the *Client* there are basically three different approaches:

a) Create several simple *Variables* using simple *DataTypes* always reflecting parts of the simple structure. *Objects* are used to group the *Variables* according to the structure of the data.

b) Create a structured *DataType* and a simple *Variable* using this *DataType*.

c) Create a structured *DataType* and a complex *Variable* using this *DataType* and also exposing the structured data structure as *Variables* of the complex *Variable* using simple *DataTypes*.

The advantages of the first approach are that the complex structure of the data is visible in the *AddressSpace*. A generic *Client* can easily access the data without knowledge of user-defined *DataTypes* and the *Client* can access individual parts of the structured data. The disadvantages of the first approach are that accessing the individual data does not provide any transactional context and for a specific *Client* the *Server* first has to convert the data and the *Client* has to convert the data, again, to get the data structure the underlying system provides.

The advantages of the second approach are, that the data is accessed in a transactional context and the structured *DataType* can be constructed in a way that the *Server* does not have to convert the data and can pass directly to the specific *Client* that can directly use them. The disadvantages are that the generic *Client* might not be able to access and interpret the data or has at least the burden to read the *DataTypeDefinition* to interpret the data. The structure of the data is not visible in the *AddressSpace*; additional *Properties* describing the data structure cannot be added to the adequate places since they do not exist in the *AddressSpace*. Individual parts of the data cannot be read without accessing the whole data structure.

---

[1] *VariableTypes* other than the *PropertyType* which is used for all *Properties*.

The third approach combines the other two approaches. Therefore a specific *Client* can access data in its native format in a transactional context, whereas a generic *Client* can access simple *DataTypes* of the components of the complex *Variable*. The disadvantage is that the *Server* must be able to provide the native format and also interpret it to be able to provide the information in simple *DataTypes*.

It is recommended to use the first approach. When a transactional context is needed or the *Client* should be able to get a large amount of data instead of subscribing to several individual values, then the third approach is suitable. However, the *Server* might not always have the knowledge to interpret the structured data of the underlying system and therefore has to use the second approach just passing the data to the specific *Client* who is able to interpret the data.

A.5 Views

Server-defined *Views* can be used to present an excerpt of the *AddressSpace* suitable for a special class of *Clients*, for example maintenance *Clients*, engineering *Clients*, etc. The *View* only provides the information needed for the purpose of the *Client* and hides unnecessary information.

A.6 Methods

Methods should be used whenever some input is expected and the *Server* delivers a result. One should avoid using *Variables* to write the input values and other *Variables* to get the output results as it was necessary to do in OPC COM since there was no concept of a *Method* available. However, a simple OPC COM wrapper might not be able to do this.

Methods can also be used to trigger some execution in the *Server* that does not require input and/or output parameters.

Global *Methods*, that is, *Methods* that cannot directly be assigned to a special *Object*, should be assigned to the *Server Object* defined in IEC 62541-5.

A.7 Defining ReferenceTypes

Defining new *ReferenceTypes* should only be done if the predefined *ReferenceTypes* are not suitable. Whenever a new *ReferenceType* is defined, the most appropriate *ReferenceType* should be used as its supertype.

It is expected that *Servers* will have new defined hierarchical *ReferenceTypes* to expose different hierarchies, and new non-hierarchical *References* to expose relationships between *Nodes* in the *AddressSpace*.

A.8 Defining ModellingRules

New *ModellingRules* have to be defined if the predefined *ModellingRules* are not appropriate for the model exposed by the *Server*.

Depending on the model used by the underlying system the *Server* may need to define new *ModellingRules*, since the OPC UA *Server* may only pass the data to the underlying system and this system may use its own internal rules for instantiation, subtyping, etc.

Beside this, the predefined *ModellingRules* might not be sufficient to specify the required behaviour for instantiation and subtyping.

Annex B
(informative)

OPC UA Meta Model in UML

B.1 Background

The OPC UA Meta Model (the OPC UA Address Space Model) is represented by UML classes and UML objects marked with the stereotype <<TypeExtension>>. Those stereotyped UML objects represent *DataTypes* or *ReferenceTypes*. The domain model can contain user-defined *ReferenceTypes* and *DataTypes*, also marked as <<TypeExtension>>. In addition, the domain model contains *ObjectTypes*, *VariableTypes*, etc. represented as UML objects (see Figure B.1).

The OPC Foundation specifies not only the OPC UA Meta Model, but also defines some *Nodes* to organize the *AddressSpace* and to provide information about the *Server* as specified in IEC 62541-5.

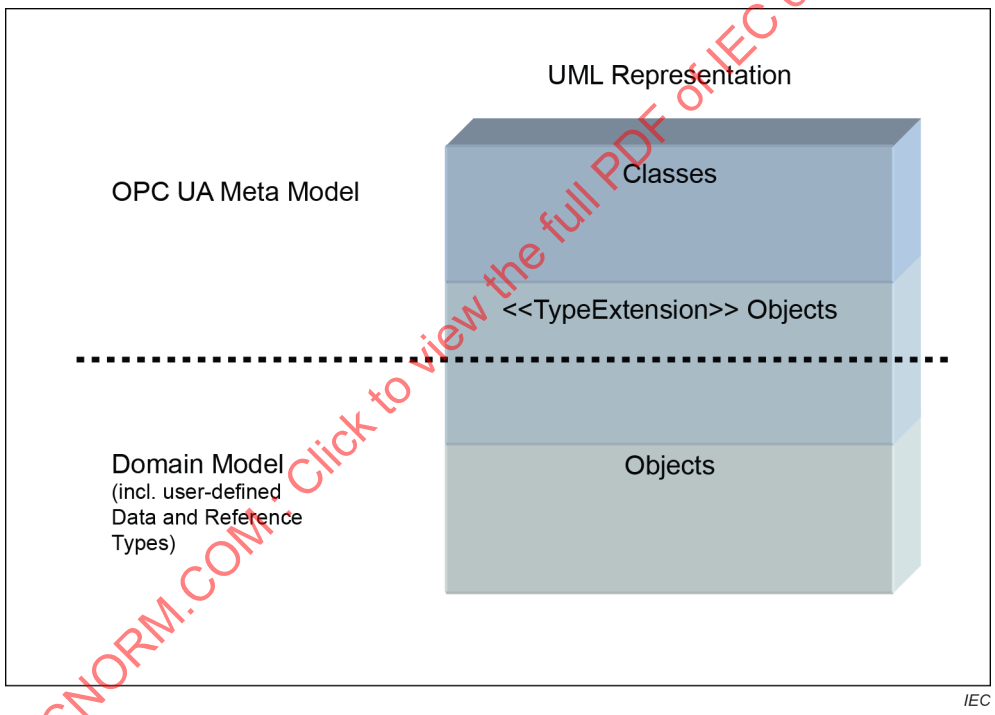


Figure B.1 – Background of OPC UA Meta Model

B.2 Notation

An example of a UML class representing the OPC UA concept *Base* is given in the UML class diagram in Figure B.2. OPC Attributes inherit from the abstract class Attribute and have a value identifying their data type. They are composed of a *Node* which is either optional (0..1) or required (1), such as *BrowseName* to *Base* in Figure B.2.

Figure B.2 – Notation (I)

UML object diagrams are used to display <<TypeExtension>> objects (e.g. *HasComponent* in Figure B.3). In object diagrams, OPC *Attributes* are represented as UML attributes without data types and marked with the stereotype <<Attribute>>, like *InverseName* in the UML object *HasComponent*. They have values, like *InverseName* = *ComponentOf* for *HasComponent*. To keep the object diagrams simple, not all *Attributes* are shown (e.g. the *NodeId* of *HasComponent*).

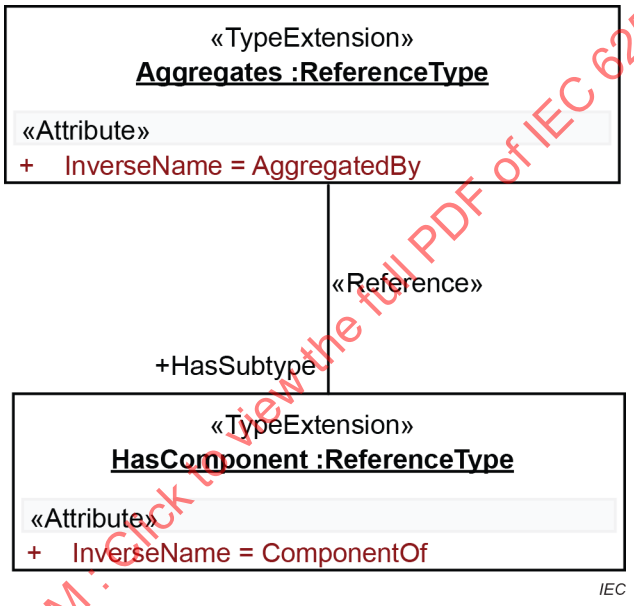


Figure B.3 – Notation (II)

OPC *References* are represented as UML associations marked with the stereotype <<Reference>>. If a particular *ReferenceType* is used, its name is used as the role name, identifying the direction of the *Reference* (e.g. *Aggregates* has the subtype *HasComponent*). For simplicity, the inverse role name is not shown (in the example *SubtypeOf*). When no role name is provided, it means that any *ReferenceType* can be used (only valid for class diagrams).

There are some special *Attributes* in OPC UA containing a *NodeId* and thereby referencing another *Node*. Those *Attributes* are represented as associations marked with the stereotype <<Attribute>>. The name of the *Attribute* is displayed as the role name of the *TargetNode*.

The value of the OPC *Attribute BrowseName* is represented by the UML object name, for example the *BrowseName* of the UML object *HasComponent* in Figure B.3 is "HasComponent".

To highlight the classes explained in a class diagram, they are marked in grey (e.g. *Base* in Figure B.2). Only those classes have all of their relationships to other classes and attributes shown in the diagram. For the other classes, we provide only those attributes and relationships needed to understand the main classes of the diagram.

B.3 Meta Model

NOTE Other parts of IEC 62541 can extend the OPC UA Meta Model by adding *Attributes* and defining new *ReferenceTypes*.

B.3.1 Base

Base is shown in Figure B.4.

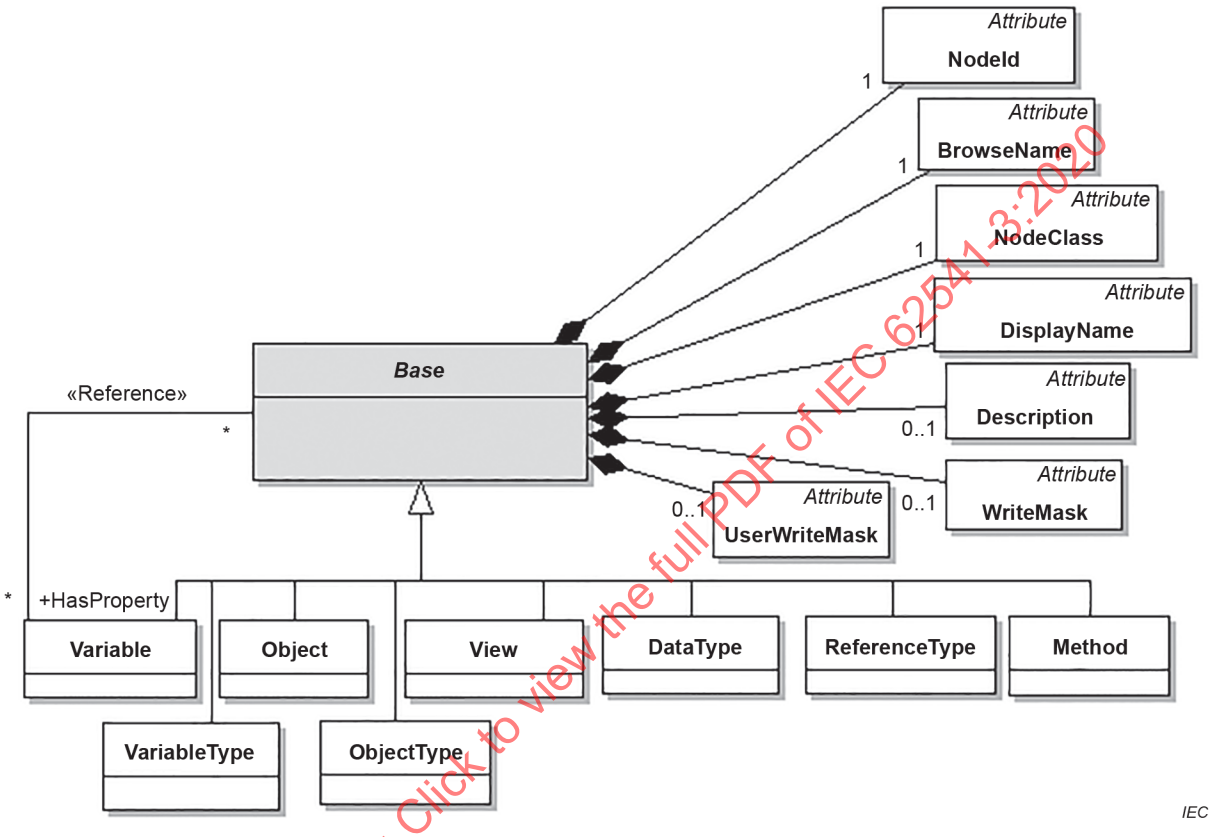


Figure B.4 – Base

B.3.2 ReferenceType

ReferenceType is shown in Figure B.5 and predefined *ReferenceTypes* in Figure B.6.

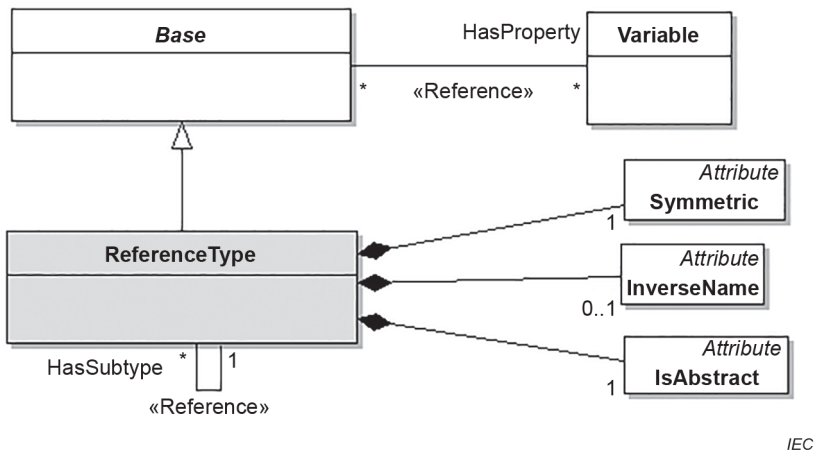


Figure B.5 – Reference and ReferenceType

If *Symmetric* is “false” and *IsAbstract* is “false” an *InverseName* shall be provided.

B.3.3 Predefined ReferenceTypes

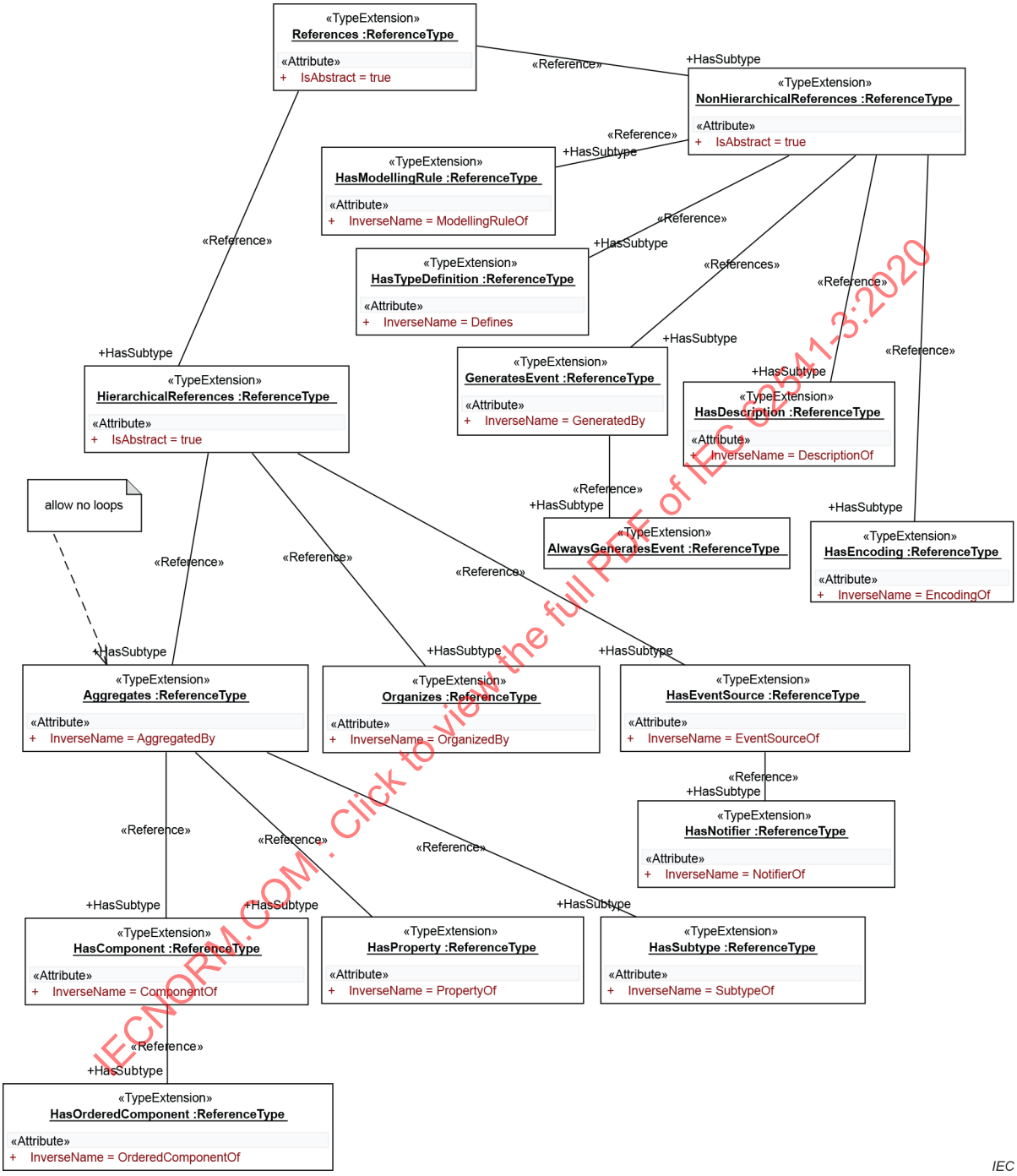


Figure B.6 – Predefined ReferenceTypes

B.3.4 Attributes

Attributes are shown in Figure B.7.

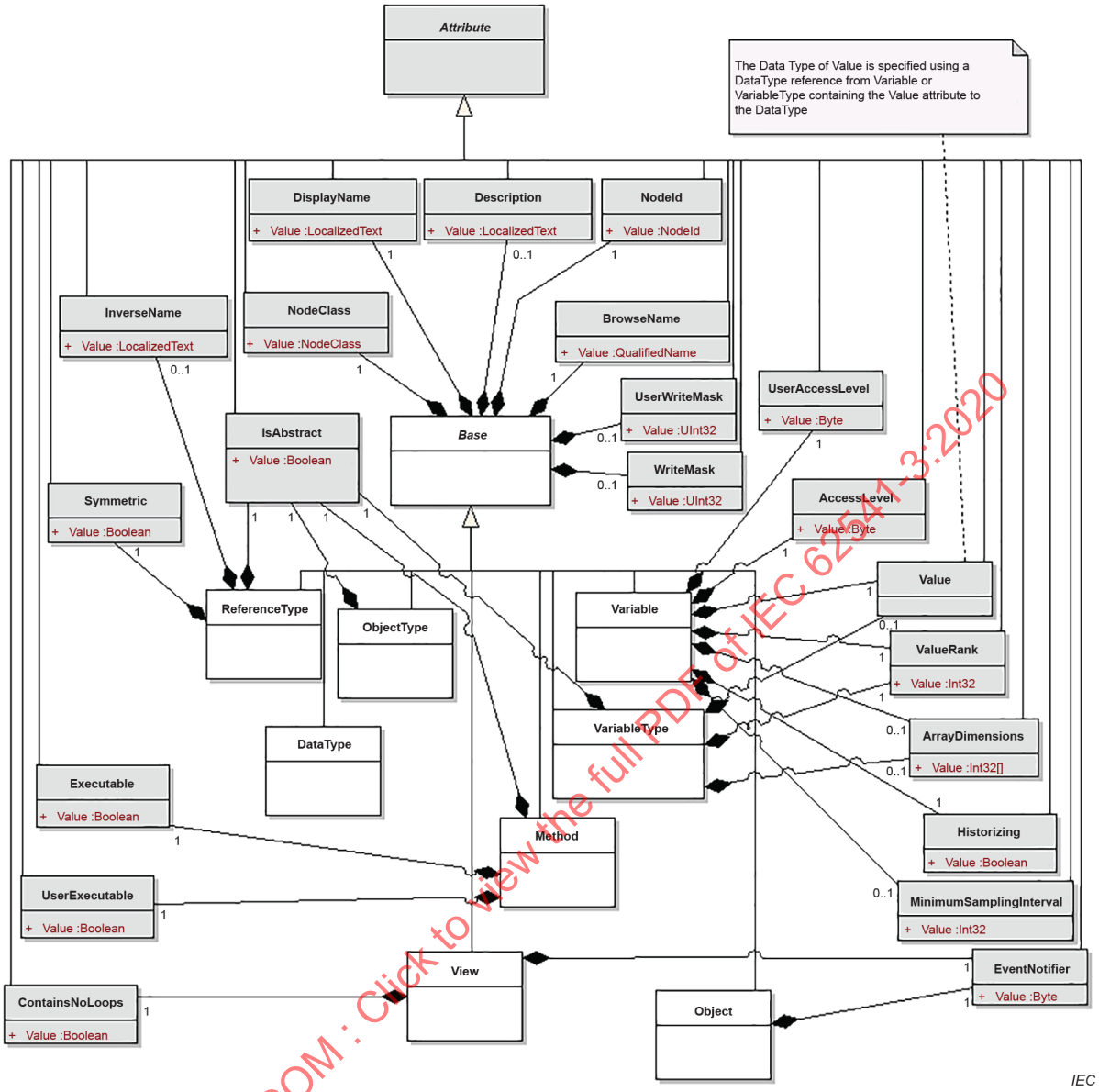


Figure B.7 – Attributes

There may be more *Attributes* defined in other parts of IEC 62541.

Attributes used for references, which have a *NodeId* as *DataType*, are not shown in this diagram but are shown as stereotyped associations in the other diagrams.

B.3.5 Object and ObjectType

Objects and *ObjectTypes* are shown in Figure B.8.

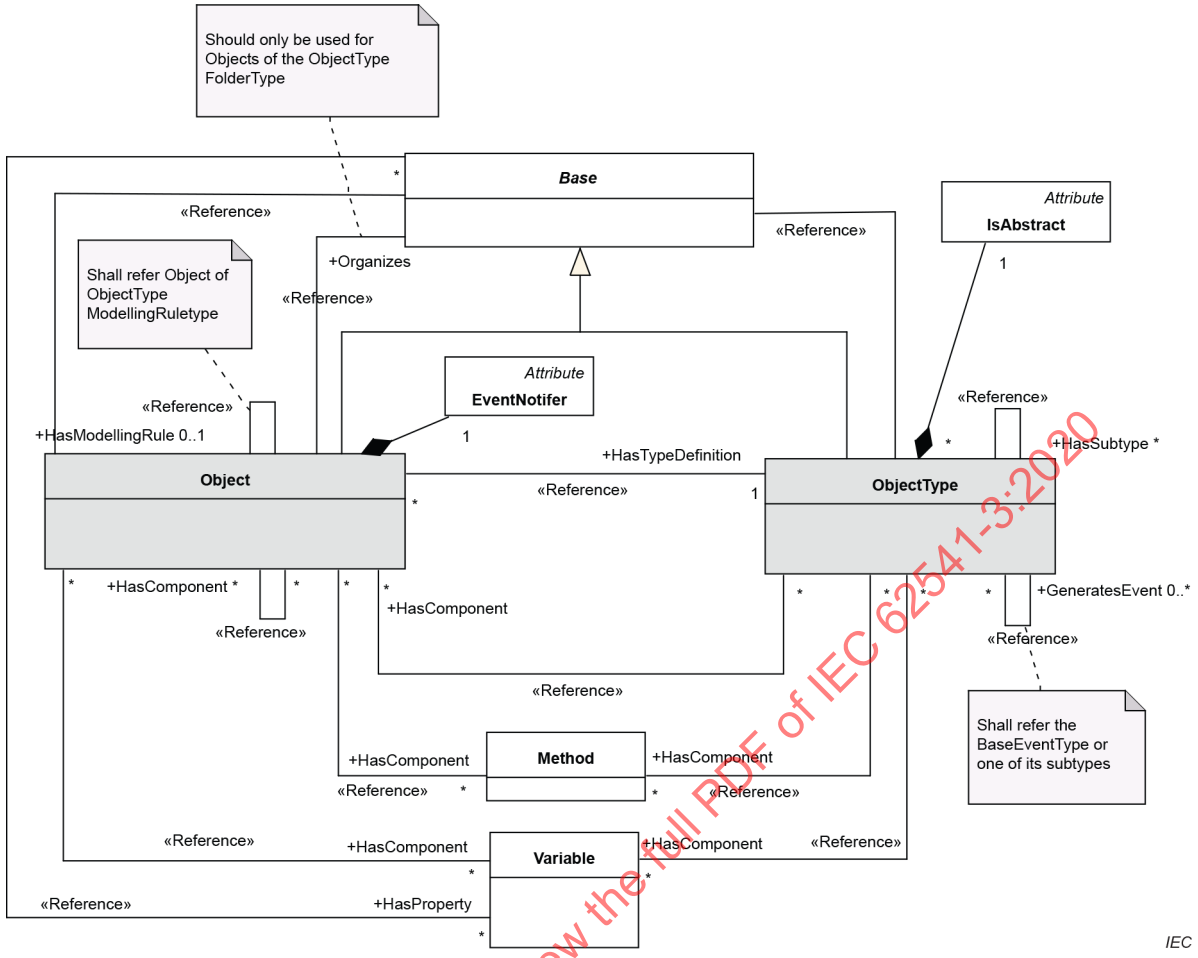


Figure B.8 – Object and ObjectType

B.3.6 EventNotifier

EventNotifier is shown in Figure B.9.

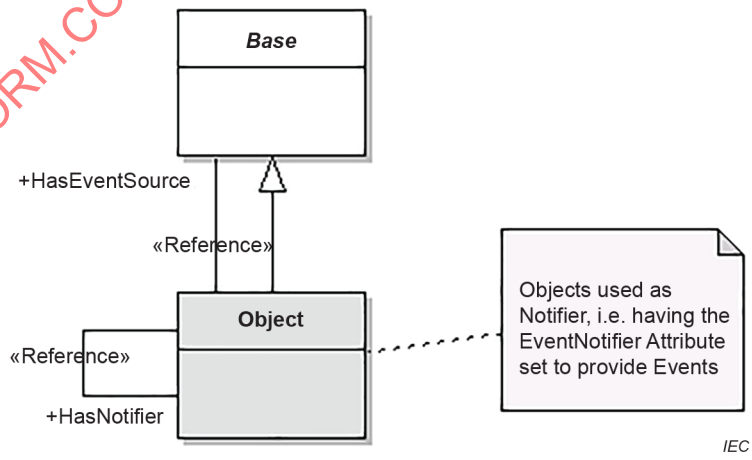


Figure B.9 – EventNotifier

B.3.7 Variable and VariableType

Variable and VariableType are shown in Figure B.10.

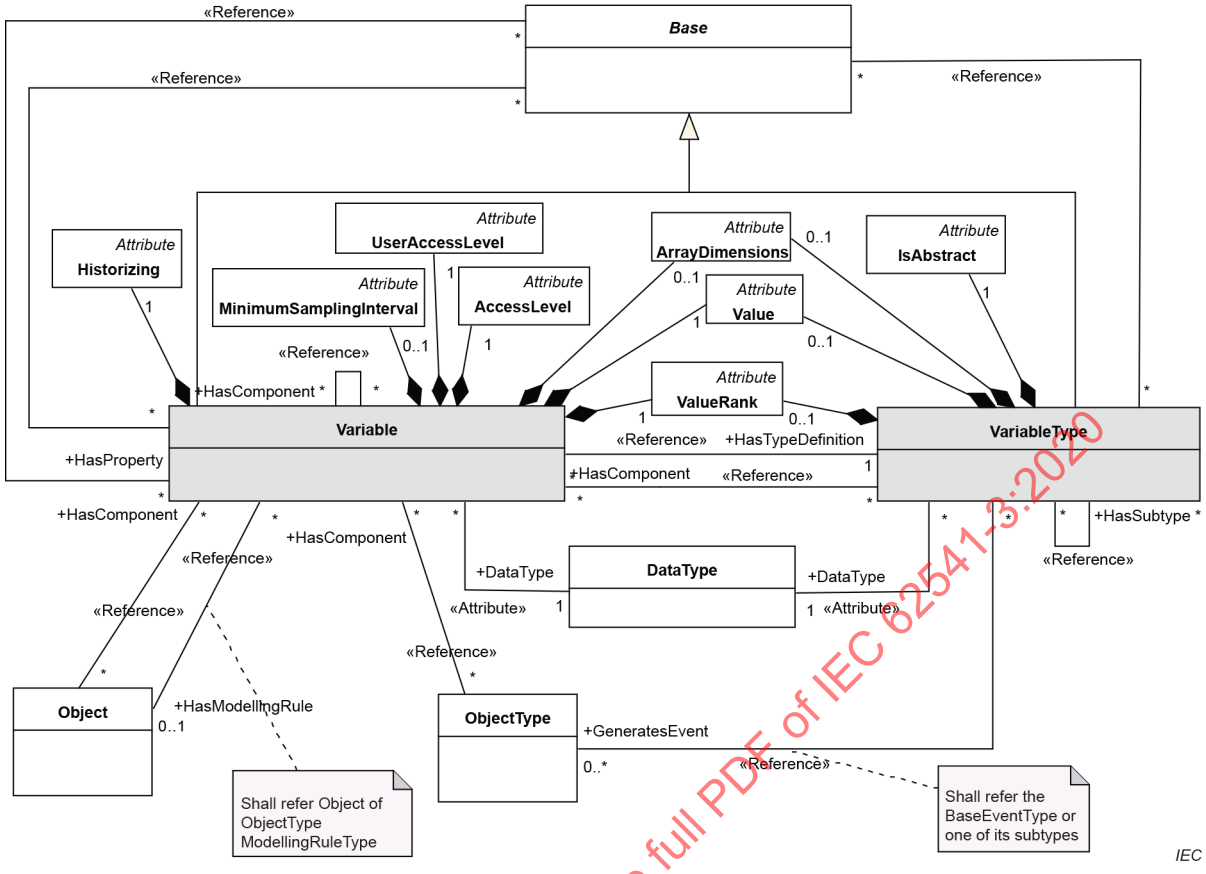


Figure B.10 – Variable and VariableType

The *DataType* of a *Variable* shall be the same as or a subtype of the *DataType* of its *VariableType* (referred with *HasTypeDefinition*).

If a *HasProperty* points to a *Variable* from a *Base* "A" then the following constraints apply:

- The Variable shall not be the SourceNode of a HasProperty or any other HierarchicalReferences Reference.
- All *Variables* having "A" as the *SourceNode* of a *HasProperty Reference* shall have a unique *BrowseName* in the context of "A".

B.3.8 Method

Method is shown in Figure B.11

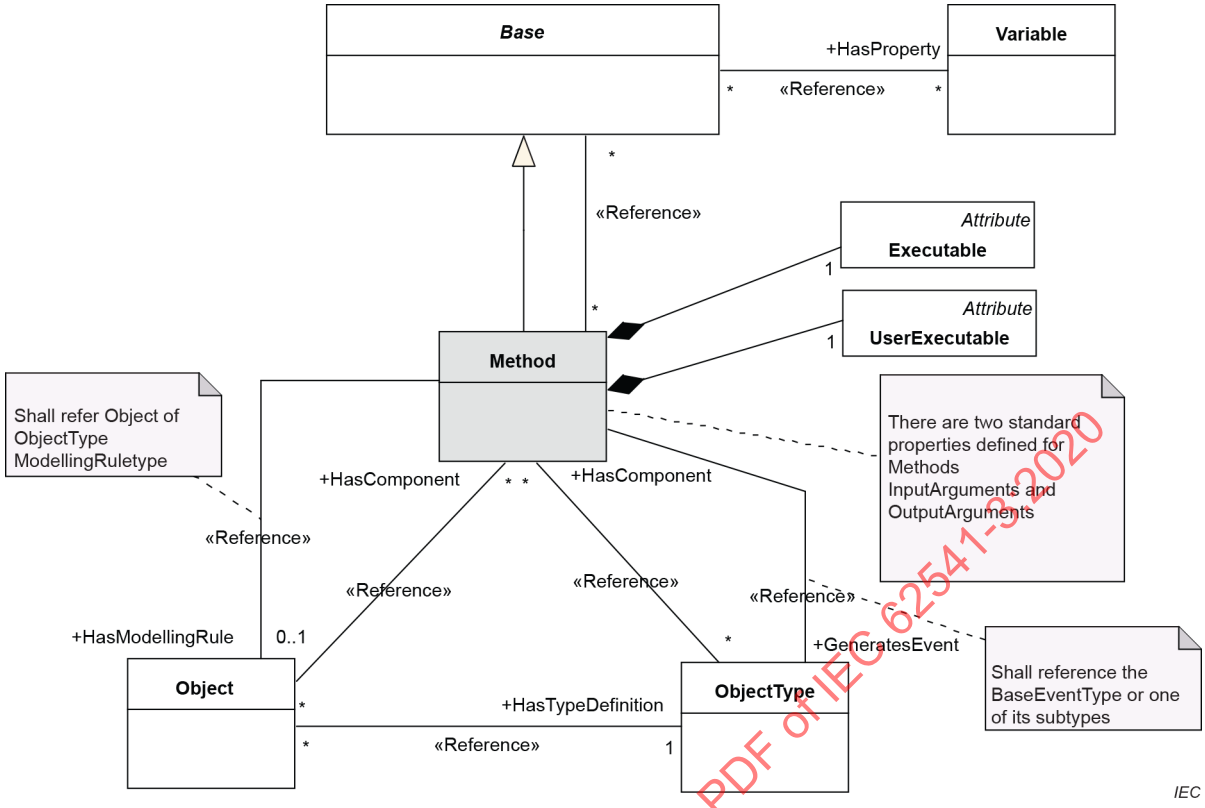


Figure B.11 – Method

B.3.9 DataType

DataType is shown in Figure B.12.

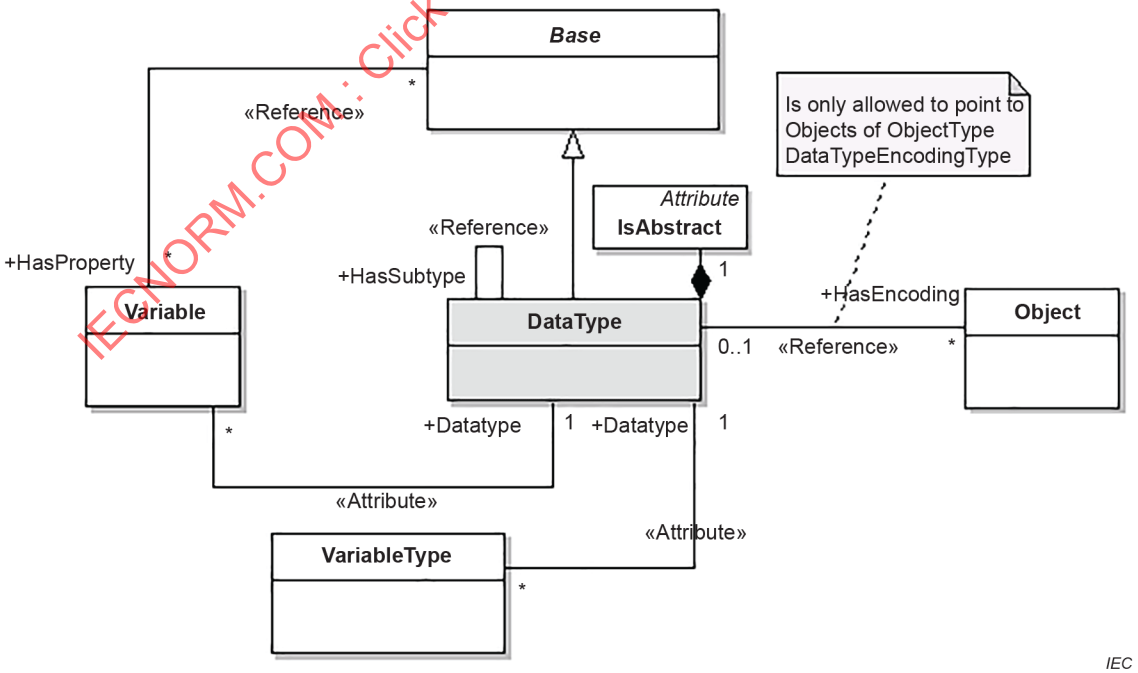


Figure B.12 – DataType

B.3.10 View

View is shown in Figure B.13.

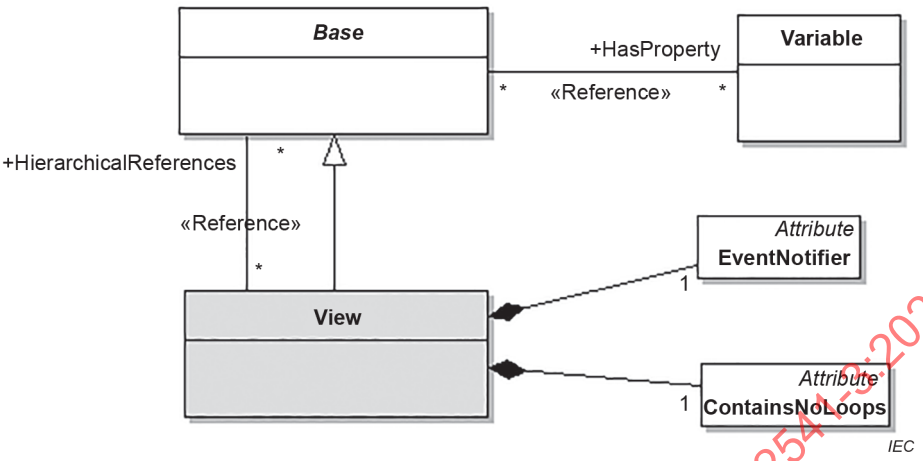


Figure B.13 – View

Annex C
(normative)

Graphical notation

C.1 General

Annex C defines a graphical notation for OPC UA data. Annex C is normative, that is, the notation is used in this document to expose examples of OPC UA data. However, it is not required to use this notation to expose OPC UA data.

The graphical notation is able to expose all structural data of OPC UA. *Nodes*, their *Attributes* including their current value and *References* between the *Nodes* including the *ReferenceType* can be exposed. The graphical notation provides no mechanism to expose events or historical data.

C.2 Notation

C.2.1 Overview

The notation is divided into two parts. The simple notation only provides a simplified view on the data hiding some details like *Attributes*. The extended notation allows exposing all structure information of OPC UA, including *Attribute* values. The simple and the extended notation can be combined to expose OPC UA data in one figure.

Common to both notations is that neither any colour nor the thickness or style of lines is relevant for the notation. Those effects can be used to highlight certain aspects of a figure.

C.2.2 Simple notation

Depending on their *NodeClass Nodes* are represented by different graphical forms as defined in Table C.1.

Table C.1 – Notation of Nodes depending on the NodeClass

NodeClass	Graphical Representation	Comment
Object	Object	Rectangle including text representing the string-part of the *DisplayName* of the *Object*. The font shall not be set to italic.
ObjectType	*ObjectType*	Shadowed rectangle including text representing the string-part of the *DisplayName* of the *ObjectType*. The font shall be set in italic.
Variable	Variable	Rectangle with rounded corners including text representing the string-part of the *DisplayName* of the *Variable*. The font shall not be set in italic.
VariableType	*VariableType*	Shadowed rectangle with rounded corners including text representing the string-part of the *DisplayName* of the *VariableType*. The font shall be set in italic.
DataType	*DataType*	Shadowed hexagon including text representing the string-part of the *DisplayName* of the *DataType*.
ReferenceType	*ReferenceType*	Shadowed six-sided polygon including text representing the string-part of the *DisplayName* of the *ReferenceType*.
Method	Method	Oval including text representing the string-part of the *DisplayName* of the *Method*.
View	View	Trapezium including text representing the string-part of the *DisplayName* of the *View*.

References are represented as lines between *Nodes* as exemplified in Figure C.1. Those lines can vary in their form. They do not have to connect the *Nodes* with a straight line; they can have angles, arches, etc.

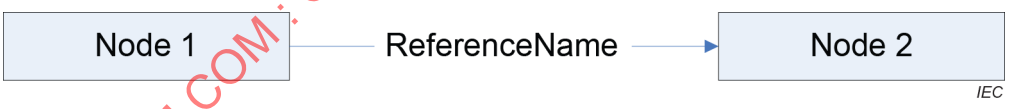
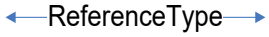
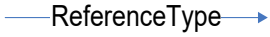
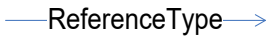


Figure C.1 – Example of a Reference connecting two Nodes

Table C.2 defines how symmetric and asymmetric *References* are represented in general, and also defines shortcuts for some *ReferenceTypes*. Although it is recommended to use those shortcuts, it is not required. Thus, instead of using the shortcut, the generic solution can also be used.

Table C.2 – Simple Notation of Nodes depending on the NodeClass

ReferenceType	Graphical Representation	Comment
Any symmetric ReferenceType		Symmetric *ReferenceTypes* are represented as lines between *Nodes* with closed and filled arrows on both sides pointing to the connected *Nodes*. Near the line has to be a text containing the string-part of the *BrowseName* of the *ReferenceType*.
Any asymmetric ReferenceType		Asymmetric *ReferenceTypes* are represented as lines between *Nodes* with a closed and filled arrow on the side pointing to the *TargetNode*. Near the line has to be a text containing the string-part of the *BrowseName* of the *ReferenceType*.
Any hierarchical ReferenceType		Asymmetric *ReferenceTypes* that are subtypes of *HierarchicalReferences* should be exposed the same way as asymmetric *ReferenceTypes* except that an open arrow is used.
HasComponent		The notation provides a shortcut for *HasComponent References* shown on the left. The single hashed line has to be near the *TargetNode*.
HasProperty		The notation provides a shortcut for *HasProperty References* shown on the left. The double hashed lines have to be near the *TargetNode*.
HasTypeDefinition		The notation provides a shortcut for *HasTypeDefinition References* shown on the left. The double closed and filled arrows have to point to the *TargetNode*.
HasSubtype		The notation provides a shortcut for *HasSubtype References* shown on the left. The double closed arrows have to point to the *SourceNode*.
HasEventSource		The notation provides a shortcut for *HasEventSource References* shown on the left. The closed arrow has to point to the *TargetNode*.

C.2.3 Extended notation

In the extended notation some additional concepts are introduced. It is allowed only to use some of those concepts on elements of a figure.

The following rules define some special handling of structures.

- In general, values of all *DataTypes* should be represented by an appropriate string representation. Whenever a *NamespaceIndex* or *LocaleId* is used in those structures they can be omitted.
- The *DisplayName* contains a *LocaleId* and a *String*. Such a structure can be exposed as [<LocaleId>:]<String> where the *LocaleId* is optional. For example, a *DisplayName* can be “en:MyName”. Instead of that, “MyName” can also be used. This rule applies whenever a *DisplayName* is shown, including the text used in the graphical representation of a *Node*.
- The *BrowseName* contains the *NamespaceIndex* and a *String*. Such a structure can be exposed as [<NamespaceIndex>:]<String> where the *NamespaceIndex* is optional. For example, a *BrowseName* can be “1:MyName”. Instead of that, “MyName” can also be used. This rule applies whenever a *BrowseName* is shown, including the text used in the graphical representation of a *Node*.

Instead of using the *HasTypeDefinition* reference to point from an *Object* or *Variable* to its *ObjectType* or *VariableType* the name of the *TypeDefinition* can be added to the text used in the *Node*. The *TypeDefinition* shall either be prefixed with “::” or it is put in italic as the top line. Figure C.2 gives an example, where “Node1” uses a *Reference* and “Node2” the shortcut in both notation variants. A figure can contain *HasTypeDefinition References* for some *Nodes* and the shortcut for other *Nodes*. It is not allowed that a *Node* uses the shortcut and additionally is the *SourceNode* of a *HasTypeDefinition*.

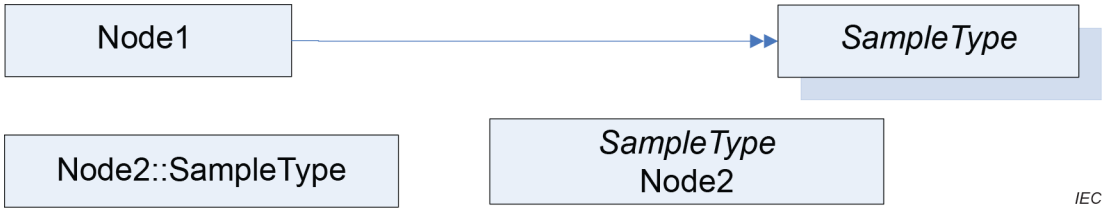


Figure C.2 – Example of using a TypeDefinition inside a Node

To display *Attributes* of a *Node* additional text can be put inside the form representing the *Node* under the text representing the *DisplayName*. The *DisplayName* and the text describing the *Attributes* have to be separated using a horizontal line. Each *Attribute* has to be set into a new text line. Each text line shall contain the *Attribute* name followed by an "=" and the value of the *Attribute*. On top of the first text line containing an *Attribute* shall be a text line containing the underlined text "Attribute". It is not required to expose all *Attributes* of a *Node*. It is allowed to show only a subset of *Attributes*. If an optional *Attribute* is not provided, the *Attribute* can be marked by a strike-through line, for example "~~Description~~". Examples of exposing *Attributes* are shown in Figure C.3.

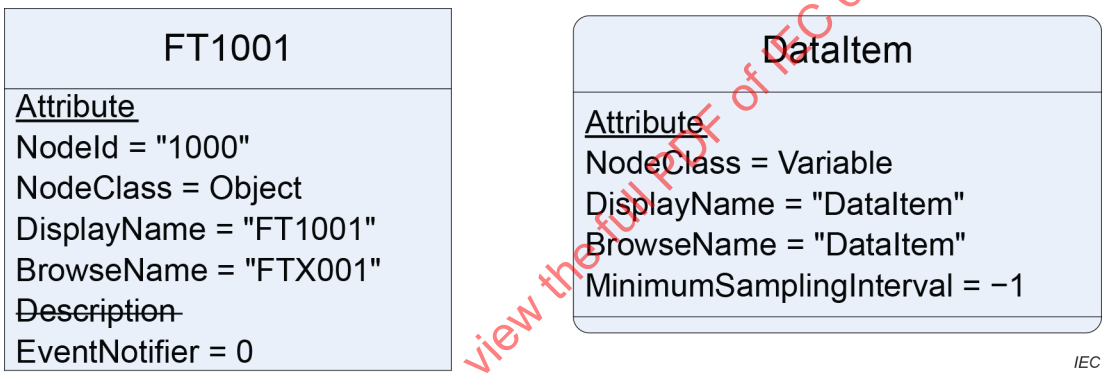


Figure C.3 – Example of exposing Attributes

To avoid too many *Nodes* in a figure it is allowed to expose *Properties* inside a *Node*, similar to *Attributes*. Therefore, the text field used for exposing *Attributes* is extended. Under the last text line containing an *Attribute* a new text line containing the underlined text "Property" has to be added. If no *Attribute* is provided, the text has to start with this text line. After this text line, each new text line shall contain a *Property*, starting with the *BrowseName* of the *Property* followed by "=" and the value of the *Value Attribute* of the *Property*. Figure C.4 shows some examples exposing *Properties* inline. It is allowed to expose some *Properties* of a *Node* inline, and other *Properties* as *Nodes*. It is not allowed to show a *Property* inline as well as an additional *Node*.

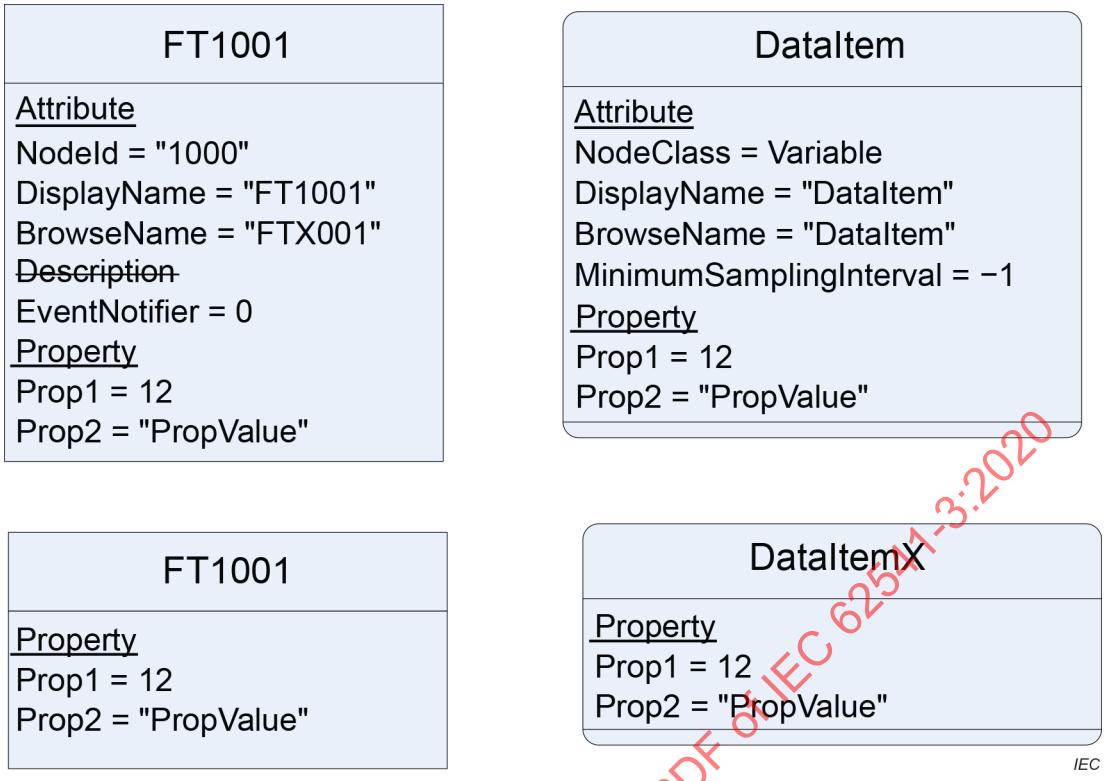


Figure C.4 – Example of exposing Properties inline

Adding additional information to a figure using the graphical representation, for example callouts, is permitted.

Bibliography

IEC 62541-11, *OPC Unified Architecture – Part 11: Historical Access*

_____

SOMMAIRE

COMMISSION ÉLECTROTECHNIQUE INTERNATIONALE

_____

ARCHITECTURE UNIFIÉE OPC –

Partie 3: Modèle d'espace d'adressage

AVANT-PROPOS

1) La Commission Electrotechnique 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. A 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 62541-3 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.

Cette troisième édition annule et remplace la deuxième édition parue en 2015.

Cette édition inclut les modifications techniques majeures suivantes par rapport à l'édition précédente:

a) ajout d'une nouvelle approche améliorée afin de présenter les définitions de la structure (un Attribut sur le Nœud du DataType contient désormais simplement une description binaire);
b) ajout de nouveaux fanions pour les Variables afin d'indiquer l'atomicité à la lecture ou à l'écriture;

c) ajout de Rôles et d'Autorisations afin de permettre la configuration d'une autorisation fondée sur les rôles;

d) ajout de nouveaux types de données: "Union", "Decimal", "OptionSet", "DateString", "TimeString", "DurationString", "NormalizedString", "DecimalString" et "AudioDataType";

e) ajout d'une définition expliquant comment utiliser les OptionalPlaceHolder et MandatoryPlaceHolder de ModellingRules pour les Méthodes;

f) ajout de Propriétés facultatives "MaxCharacters" et "MaxByteStringLength" aux Nœuds de variable.

Le texte de ce rapport technique est issu des documents suivants:

FDIS	Rapport de vote
65E/715/FDIS	65E/731/RVD

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

Ce document a été rédigé selon les Directives ISO/IEC, Partie 2.

Tout au long du présent document et des autres parties référencées de la série IEC 62541, certaines conventions documentaires sont utilisées:

Le format *italique* est utilisé pour mettre en évidence un terme défini ou une définition qui apparaît à l'Article 3 dans l'une des parties de la série.

Le format *italique* est également utilisé pour mettre en évidence le nom d'un paramètre d'entrée ou de sortie de service, ou le nom d'une structure ou d'un élément de structure habituellement défini dans les tableaux.

Par ailleurs, les *termes* et les *noms en italique* sont, à quelques exceptions près, écrits en camel-case (pratique qui consiste à joindre, sans espace, les éléments des mots ou expressions composés, la première lettre de chaque élément étant en majuscule). Par exemple, le terme défini est *AddressSpace* et non Espace d'adressage. Cela permet de mieux comprendre qu'il existe une définition unique pour *AddressSpace*, et non deux définitions distinctes pour Espace et pour Adressage.

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 ce document 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 au document recherché. A cette date, le document sera

- reconduit,
- supprimé,
- remplacé par une édition révisée, ou
- amendé.

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 3: Modèle d'espace d'adressage

1 Domaine d'application

La présente partie de l'IEC 62541 définit l'*AddressSpace* de l'Architecture Unifiée OPC (OPC UA) et ses *Objets*. Le présent document correspond au métamodèle OPC UA sur lequel se fondent les modèles d'information OPC UA.

2 Références normatives

Les documents suivants sont cités dans le texte de sorte qu'ils constituent, pour tout ou partie de leur contenu, des exigences du présent document. 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 62541-4, *Architecture unifiée OPC – Partie 4: Services*

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

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

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

ISO/IEC/IEEE 60559:2011, *Information technology – Microprocessor Systems – Floating-Point arithmetic* (disponible en anglais seulement)

ISO 639 (toutes les parties), *Codes pour la représentation des noms de langue*

ISO 3166 (toutes les parties), *Codes pour la représentation des noms de pays et de leurs subdivisions*

ISO 8601 (toutes les parties), *Date and time – Representations for information interchange* (disponible en anglais seulement)

IETF RFC 5646, Tags for Identifying Languages (disponible en anglais seulement)
http://tools.ietf.org/html/rfc5646

Unicode Standard Annex #15: Unicode Normalization Forms (disponible en anglais seulement) http://www.unicode.org/reports/tr15/

W3C XML Schema Definition Language (XSD) Part 2: DataTypes (disponible en anglais seulement) http://www.w3.org/TR/xmlschema-2/

TAI: Temps Atomique International
http://www.bipm.org/en/bipm-services/timescales/tai.html

3 Termes, définitions, termes abrégés et conventions

3.1 Termes et définitions

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

L'ISO et l'IEC tiennent à jour des bases de données terminologiques destinées à être utilisées en normalisation, consultables aux adresses suivantes:

- IEC Electropedia: disponible à l'adresse http://www.electropedia.org/
- ISO Online browsing platform: disponible à l'adresse http://www.iso.org/obp

3.1.1
DataType
instance d'un *Nœud DataType* qui est utilisée avec l'*Attribut ValueRank* pour définir le type de données d'une *Variable*

3.1.2
DataTypeId
NodeId d'un *Nœud DataType*

3.1.3
DataVariable
Variable qui représente la *valeur* d'un *Objet*, directement ou indirectement, pour des *Variables* complexes, les *Variables* étant toujours le *TargetNode* d'une *Référence HasComponent*

3.1.4
EventType
Nœud ObjectType représentant la définition de type d'un *Événement* donné

3.1.5
Référence hiérarchique
Référence utilisée pour construire des hiérarchies dans l'*AddressSpace*

Note 1 à l'article: Tous les ReferenceTypes hiérarchiques sont issus des HierarchicalReferences.

3.1.6
InstanceDeclaration
Nœud utilisé par un *TypeDefinitionNode* complexe pour présenter sa structure complexe

Note 1 à l'article: Cette instance est utilisée par une définition de type.

3.1.7
ModellingRule
métadonnée d'une *InstanceDeclaration* qui définit la manière dont l'*InstanceDeclaration* est utilisée pour l'instanciation et définit également les règles de sous-typage d'une *InstanceDeclaration*

3.1.8
Propriété
Variable qui constitue le *TargetNode* d'une *Référence HasProperty*

Note 1 à l'article: Les *Propriétés* décrivent les caractéristiques d'un *Nœud*.

3.1.9
SourceNode
Nœud qui sert de *Référence* à un autre *Nœud*

EXEMPLE: Dans la *Référence* "A contient B", "A" est le *SourceNode*.

3.1.10
TargetNode
Nœud qui est référencé par un autre *Nœud*

EXEMPLE: Dans la *Référence* "A contient B", "B" est le *TargetNode*.

3.1.11
TypeDefinitionNode
Nœud utilisé pour définir le type d'un autre *Nœud*

Note 1 à l'article: Les Nœuds ObjectType et VariableType sont des TypeDefinitionNodes.

3.1.12
VariableType
Nœud qui représente la définition de type d'une *Variable*

3.2 Termes abrégés

UA Unified Architecture (Architecture unifiée)

UML Unified Modeling Language (Langage de modélisation unifié)

URI Uniform Resource Identifier (Identifiant uniforme de ressource)

W3C World Wide Web Consortium (Consortium chargé de la normalisation du web mondial)

XML eXtensible Markup Language (Langage de balisage extensible)

3.3 Conventions

3.3.1 Conventions pour les figures d'AddressSpace

Les *Nœuds* et leurs *Références* respectives sont représentés par des figures. La Figure 1 présente les conventions utilisées dans ces figures.

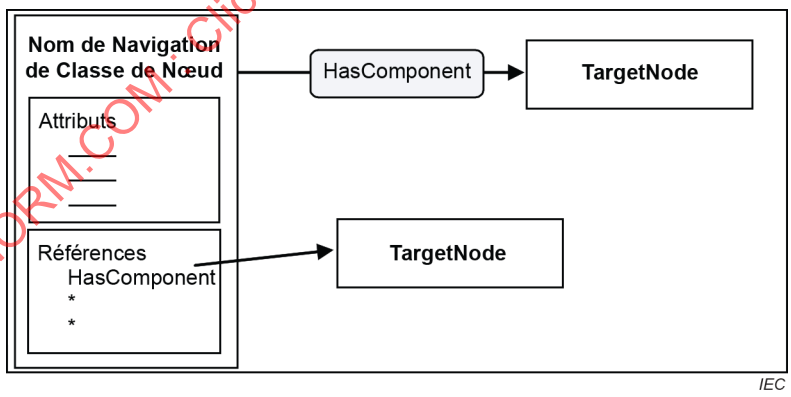


Figure 1 – Diagrammes des Nœuds de l'AddressSpace

Sur ces figures, les rectangles représentent les *Nœuds*. Le titre des rectangles des *Nœuds* peut être donné par une ou deux lignes de texte. Lorsque deux lignes sont utilisées, la première identifie la *NodeClass* et la seconde détermine le *BrowseName*. Lorsqu'une seule ligne est utilisée, elle contient le *BrowseName*.

Les rectangles des *Nœuds* peuvent inclure des cadres qui définissent leurs *Attributs* et *Références*. Des dénominations spécifiques dans ces cadres identifient des *Attributs* et des *Références* spécifiques.

Des rectangles grisés aux angles arrondis et traversés par des flèches représentent des *Références*. La flèche en question part du *SourceNode* et pointe vers le *TargetNode*. Les *Références* peuvent également être représentées en traçant une flèche qui part du nom de la *Référence* dans la case "Références" et aboutit au *TargetNode*.

3.3.2 Conventions pour la définition des NodeClasses

L'Article 5 définit les *NodeClasses de l'AddressSpace*. Le Tableau 1 décrit le format des tableaux utilisés pour définir les *NodeClasses*.

Tableau 1 – Conventions des tableaux de NodeClasses

Nom	Usage	Type de données	Description
Attributs			
"Nom de l'attribut"	"M" ou "O"	Type de données de l'*Attribut*	Définit l'*Attribut*
Références			
"Nom de la référence"	"1", "0..1" ou "0..*"	Non utilisé	Décrit l'utilisation de la *Référence* par la *NodeClass*
Propriétés normalisées			
"Nom de la Propriété"	"M" ou "O"	Type de données de la *Propriété*	Définit la *Propriété*

La colonne Nom donne le nom de l'*Attribut*, le nom du *ReferenceType* utilisé pour créer une *Référence* ou le nom d'une *Propriété* référencée par la *Référence HasProperty*.

La colonne Usage définit le caractère obligatoire (M pour Mandatory) ou facultatif (O pour Optional) de l'*Attribut* ou de la *Propriété*. Lorsqu'il est obligatoire, l'*Attribut* ou la *Propriété* doit exister pour tout *Nœud* de la *NodeClass*. Pour les *Références*, cette colonne précise la cardinalité. Les valeurs suivantes peuvent s'appliquer:

- "0..*" indique l'absence totale de restrictions; en d'autres termes, il n'est pas nécessaire de fournir la *Référence*, mais il n'existe par ailleurs aucune limite concernant le nombre de fois où elle peut être fournie;
- "0..1" signifie que la *Référence* est fournie une fois au plus;
- "1" signifie que la *Référence* doit être fournie une fois exactement.

La colonne Type de données indique le nom du *DataType* de l'*Attribut* ou de la *Propriété*. Elle n'est pas utilisée pour les *Références*.

La colonne Description décrit l'*Attribut*, la *Référence* ou la *Propriété*.

Seul le présent document peut définir des *Attributs*. De ce fait, tous les *Attributs* de la *NodeClass* sont spécifiés dans le tableau et peuvent uniquement être étendus par d'autres parties de la série de l'IEC 62541.

Le présent document définit également les *ReferenceTypes*; cependant, les *ReferenceTypes* peuvent également être spécifiés par un *Serveur* ou par un *Client* au moyen des *Services NodeManagement* décrits dans l'IEC 62541-4. Ainsi, les tableaux de *NodeClass* du présent document peuvent contenir les *ReferenceType* de base, appelés *Références*, qui indiquent que tout *ReferenceType* peut être utilisé pour la *NodeClass*, y compris les *ReferenceTypes* spécifiques au système. Les tableaux de *NodeClass* spécifient uniquement la manière dont les *NodeClasses* peuvent être utilisées comme *SourceNodes* de *Références* et non comme

TargetNodes. Si un tableau de *NodeClass* permet l'utilisation d'un *ReferenceType* de sa *NodeClass* comme *SourceNode*, ceci est également vrai pour les sous-types de *ReferenceType*. Les sous-types de *ReferenceType* peuvent toutefois restreindre ses *SourceNodes*.

Le présent document définit des *Propriétés*, mais d'autres organismes ou fournisseurs de normalisation peuvent aussi définir des *Propriétés*; les *Nœuds* peuvent également avoir des *Propriétés* qui ne sont pas normalisées. Les *Propriétés* établies dans le présent document sont définies par leur dénomination qui est mappée vers le *BrowseName* avec un *NamespaceIndex* 0, lequel représente le *Namespace* pour l'OPC UA.

La colonne Usage (facultatif ou obligatoire) n'implique pas de *ModellingRule* spécifique aux *Propriétés*. Chaque mise en œuvre de *Serveur* particulier choisit d'utiliser les *ModellingRules* qui lui conviennent.

4 Concepts de l'AddressSpace

4.1 Vue d'ensemble

L'Article 4 définit les concepts de l'*AddressSpace*. L'Article 5 définit les *NodeClasses* de l'*AddressSpace* qui représentent les concepts de l'*AddressSpace*. L'Article 6 décrit précisément le modèle type des *ObjectTypes* et des *VariableTypes*. Les Articles 7 à 9 définissent les *ReferenceTypes*, les *DataTypes* et les *EventTypes* normalisés.

L'Annexe A informative fournit des considérations d'ordre général concernant la manière d'utiliser le Modèle d'espace d'adressage et l'Annexe B informative fournit un modèle UML du Modèle d'espace d'adressage. L'Annexe C normative définit une notation graphique des données OPC UA.

4.2 Modèle d'objet

Le principal objectif de l'*AddressSpace* OPC UA est de fournir une méthode normalisée permettant aux *Serveurs* de représenter des *Objets* pour les *Clients*. C'est dans ce but qu'a été conçu le Modèle d'objet OPC UA. Il définit des *Objets* en ce qui concerne les *Variables* et les *Méthodes*. Il permet également l'expression de relations avec d'autres *Objets*. La Figure 2 représente ce modèle.

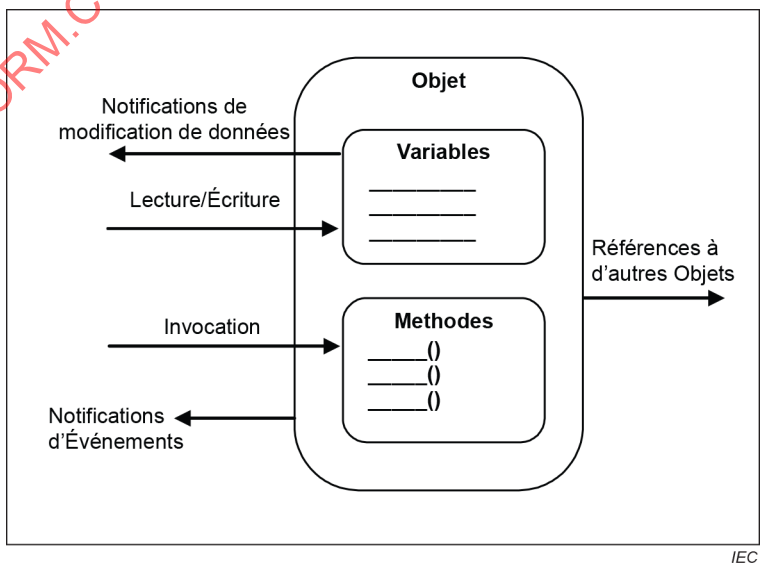


Figure 2 – Modèle d'objet OPC UA

Dans l'*AddressSpace*, les éléments de ce modèle sont représentés comme des *Nœuds*. Chaque *Nœud* est attribué à une *NodeClass* et chaque *NodeClass* représente un élément différent du Modèle d'objet. L'Article 5 définit les *NodeClasses* utilisées pour représenter ce modèle.

4.3 Modèle de nœud

4.3.1 Généralités

Le jeu d'*Objets* et les informations correspondantes que le *Serveur* OPC UA met à la disposition des *Clients* sont désignés comme son *AddressSpace*. Le modèle pour les *Objets* est défini par le Modèle d'objet OPC UA (voir 4.2).

Les Objets et leurs composants sont représentés dans l'*AddressSpace* comme un jeu de *Nœuds* décrits par des *Attributs* et interconnectés par des *Références*. La Figure 3 représente le modèle d'un *Nœud* et la suite du 4.3 décrit précisément le modèle de nœud.

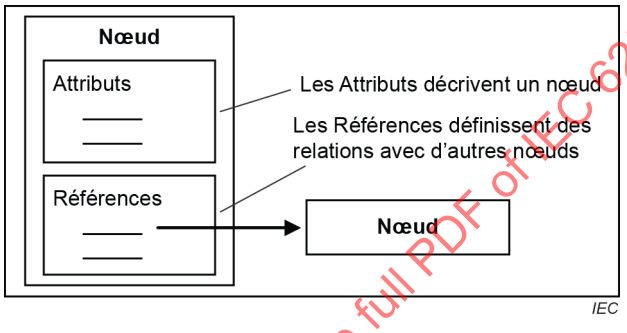


Figure 3 – Modèle de nœud de l'AddressSpace

4.3.2 NodeClasses

Les *NodeClasses* sont définies en ce qui concerne les *Attributs* et les *Références* qui doivent être instanciés (valeurs données) lorsqu'un *Nœud* est défini dans l'*AddressSpace*. Les *Attributs* sont traités en 4.3.3 et les *Références* en 4.3.4.

L'Article 5 définit les *NodeClasses* pour l'*AddressSpace* OPC UA. Ces *NodeClasses* sont collectivement appelées les métadonnées de l'*AddressSpace*. Chaque *Nœud* dans l'*AddressSpace* est une instance de l'une de ces *NodeClasses*. Aucune autre *NodeClass* ne doit être utilisée pour définir des *Nœuds* et, par conséquent, les *Clients* et les *Serveurs* ne peuvent pas définir des *NodeClasses* ni étendre les définitions de ces *NodeClasses*.

4.3.3 Attributs

Les *Attributs* sont des éléments de données qui décrivent des *Nœuds*. Les *Clients* peuvent accéder à des valeurs d'*Attribut* en utilisant les *Services* Read, Write, Query et Subscription/MonitoredItem. Ces *Services* sont définis dans l'IEC 62541-4.

Les *Attributs* sont des composants élémentaires des *NodeClasses*. Les définitions d'*Attribut* font partie intégrante de celles de *NodeClass* de l'Article 5 et, par conséquent, elles ne sont pas incluses dans l'*AddressSpace*.

Chaque définition d'*Attribut* comprend un identificateur d'attribut (pour les identificateurs d'*Attribut*, voir IEC 62541-6), une dénomination, une description, un type de données et un indicateur obligatoire/facultatif. L'ensemble d'*Attributs* défini pour chaque *NodeClass* ne doit pas être étendu par les *Clients* ou les *Serveurs*.

Lorsqu'un *Nœud* est instancié dans l'*AddressSpace*, les valeurs des *Attributs* de la *NodeClass* sont fournies. L'indicateur obligatoire/facultatif de l'*Attribut* indique si l'*Attribut* doit être instancié.

4.3.4 Références

Les *Références* sont utilisées pour corréler les *Nœuds* entre eux. Leur accès peut être obtenu en utilisant les *Services* de navigation et d'interrogation définis dans l'IEC 62541-4.

A l'instar des *Attributs*, elles sont définies comme des composants fondamentaux des *Nœuds*. Contrairement aux *Attributs*, les *Références* sont définies comme des instances des *Nœuds ReferenceType*. Les *Nœuds ReferenceType* sont visibles dans l'*AddressSpace* et sont définis en utilisant la *NodeClass ReferenceType* (voir 5.3).

Le *Nœud* qui contient la *Référence* est appelé *SourceNode* et le *Nœud* qui est référencé est appelé *TargetNode*. L'association du *SourceNode*, du *ReferenceType* et du *TargetNode* est utilisée par les *Services* OPC UA pour identifier les *Références* de manière unique. Ainsi, chaque *Nœud* peut référencer une seule fois un autre *Nœud* avec le même *ReferenceType*. Tous les sous-types de *ReferenceTypes* concrets sont considérés égaux aux *ReferenceTypes* concrets de base lorsqu'ils identifient des *Références* (voir 5.3 pour les sous-types de *ReferenceTypes*). La Figure 4 représente ce modèle de *Référence*.

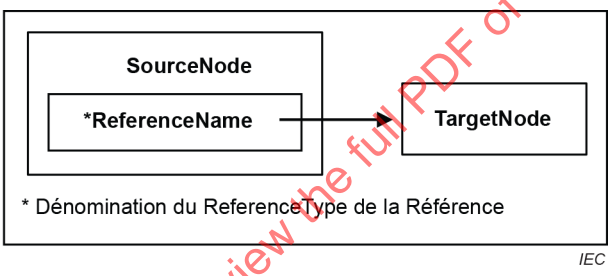


Figure 4 – Modèle de référence

Le *TargetNode* d'une *Référence* peut être situé dans le même *AddressSpace* ou dans l'*AddressSpace* d'un autre *Serveur* OPC UA. Les *TargetNodes* qui se trouvent dans d'autres *Serveurs* sont identifiés dans les *Services* OPC UA en utilisant une combinaison de la dénomination du *Serveur* distant et de l'identificateur attribué au *Nœud* par le *Serveur* distant.

L'OPC UA n'exige pas que *le TargetNode* existe; ainsi les *Références* peuvent désigner un *Nœud* qui n'existe pas.

4.4 Variables

4.4.1 Généralités

Les *Variables* sont utilisées pour représenter des *valeurs*. Deux types de *Variables* ont été définis: les *Propriétés* et les *DataVariables*. Elles diffèrent par le type de données qu'elles représentent et par le fait qu'elles peuvent contenir d'autres *Variables*.

4.4.2 Propriétés

Les *Propriétés* sont des caractéristiques d'*Objets*, de *DataVariables* et d'autres *Nœuds* définis par le *Serveur*. Les *Propriétés* diffèrent des *Attributs* du fait qu'elles caractérisent *ce que* représente le *Nœud*, par exemple un appareil ou un bon de commande. Les *Attributs* définissent des métadonnées supplémentaires qui sont instanciées pour tous les *Nœuds* à partir d'une *NodeClass*. Les *Attributs* sont communs à tous les *Nœuds* d'une *NodeClass* et sont uniquement définis par le présent document tandis que les *Propriétés* peuvent être définies par le *Serveur*.

Par exemple, un *Attribut* définit le *DataType* des *Variables* tandis qu'une *Propriété* peut être utilisée pour spécifier l'unité technique de certaines *Variables*.

Pour éviter les répétitions, les *Propriétés* ne peuvent pas avoir des *Propriétés* définies pour leur compte. Pour identifier facilement des *Propriétés*, le *BrowseName* d'une *Propriété* doit être unique dans le contexte du *Nœud* contenant les *Propriétés* (pour plus d'informations, voir 5.6.3).

Un *Nœud* et ses *Propriétés* doivent toujours résider dans le même *Serveur*.

4.4.3 DataVariables

Les *DataVariables* représentent le contenu d'un *Objet*. Il peut être par exemple défini par un *Objet* fichier qui contient un flux d'octets. Le flux d'octets peut être défini comme une *DataVariable* qui est un tableau d'octets. Les *Propriétés* peuvent être utilisées pour présenter la date de création et le propriétaire de l'*Objet* fichier.

Par exemple, si une *DataVariable* est définie par une structure de données qui contient deux champs "startTime" et "endTime", elle peut alors disposer d'une *Propriété* spécifique à cette structure de données comme "earliestStartTime".

Autre exemple, les blocs fonctionnels des systèmes de commande peuvent être représentés comme des *Objets*. Les paramètres du bloc fonctionnel, tels que ses points de consigne, peuvent être représentés comme des *DataVariables*. L'*Objet* bloc fonctionnel peut également disposer de *Propriétés* qui décrivent son temps d'exécution et son type.

Les *DataVariables* peuvent avoir des *DataVariables* supplémentaires, mais uniquement si elles sont complexes. Dans ce cas, leurs *DataVariables* doivent toujours être des éléments de définition complexe. En suivant l'exemple donné dans la description des *Propriétés* en 4.4.2, le *Serveur* peut présenter "startTime" et "endTime" comme des composants séparés de la structure de données.

Autre exemple, une *DataVariable* complexe peut définir un ensemble de valeurs de température générées par trois transmetteurs de température séparés également visibles dans l'*AddressSpace*. Dans ce cas, cette *DataVariable* complexe pourrait, à partir de là, définir des *Références HasComponent* pour les différentes valeurs de température dont elle est constituée.

4.5 TypeDefinitionNodes

4.5.1 Généralités

Les *Serveurs* OPC UA doivent fournir des définitions de type pour les *Objets* et les *Variables*. La *Référence HasTypeDefinition* doit être utilisée pour relier une instance à sa définition de type représentée par un *TypeDefinitionNode*. Les définitions de type sont exigées; pourtant, l'IEC 62541-5 définit un *BaseObjectType*, un *PropertyType* et un *BaseDataVariableType* de sorte qu'un *Serveur* puisse utiliser un type de base en l'absence d'informations de type plus spécifiques. Les *Objets* et *Variables* héritent des *Attributs* spécifiés par leur *TypeDefinitionNodes* (pour plus d'informations, voir 6.4).

Dans certains cas, le *NodeId* utilisé par la *Référence HasTypeDefinition* est notoirement connu des *Clients* et des *Serveurs*. Les organismes peuvent définir des *TypeDefinitionNodes* notoires dans l'industrie. Les *NodeIds* de *TypeDefinitionNodes* notoires assurent l'uniformité sur l'ensemble des *Serveurs* OPC UA et autorisent que les *Clients* interprètent le *TypeDefinitionNode* sans avoir à le lire à partir du *Serveur*. Par conséquent, les *Serveurs* peuvent utiliser des *NodeIds* notoires sans avoir à représenter les *TypeDefinitionNodes* correspondants dans leur *AddressSpace*. Cependant, pour les *Clients* génériques, les *TypeDefinitionNodes* doivent être fournis. Ces *TypeDefinitionNodes* peuvent exister dans un autre *Serveur*.

L'exemple suivant, représenté à la Figure 5, décrit l'utilisation d'une *Référence HasTypeDefinition*. Dans cet exemple, un paramètre de point de consigne "SP" est représenté par une *DataVariable* dans l'*AddressSpace*. Cette *DataVariable* fait partie d'un *Objet* qui n'est pas représenté sur la figure.

Pour fournir une définition de point de consigne commune qui peut être utilisée par d'autres *Objets*, un *VariableType* spécialisé est utilisé. Chaque *DataVariable* de point de consigne qui utilise cette définition commune aura une *Référence HasTypeDefinition* qui identifie le *VariableType* "SetPoint" commun.

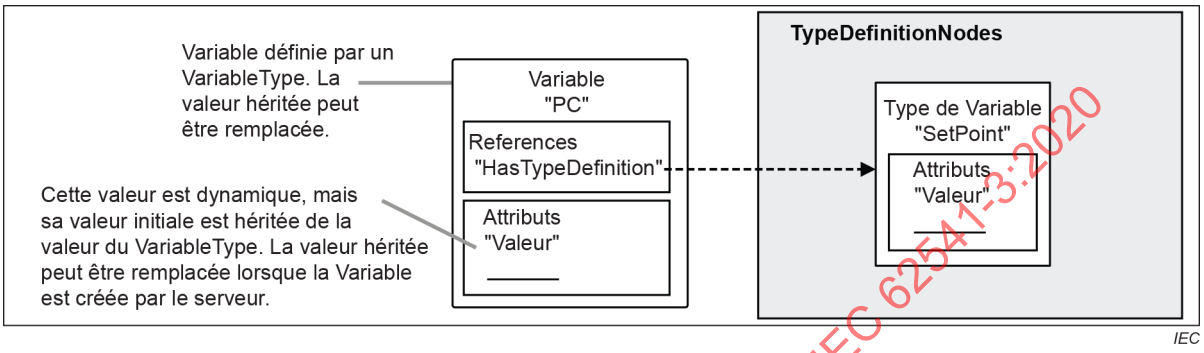


Figure 5 – Exemple d'une Variable définie par un VariableType

4.5.2 TypeDefinitionNodes complexes et leurs InstanceDeclarations

Les *TypeDefinitionNodes* peuvent être complexes. Un *TypeDefinitionNode* complexe définit également des *Références* à d'autres *Nœuds* dans le cadre de la définition de type. Les *ModellingRules* définies en 6.4.4 précisent la manière dont ces *Nœuds* sont traités pour la création d'une instance de la définition de type.

Un *TypeDefinitionNode* référence des instances plutôt que d'autres *TypeDefinitionNodes* afin de permettre l'utilisation de noms uniques pour plusieurs instances du même type, la définition de valeurs par défaut et l'ajout de *Références* à d'autres instances spécifiques à ce *TypeDefinitionNode* complexe et non au *TypeDefinitionNode* de l'instance. Par exemple, dans la Figure 6, l'*ObjectType* "AI_BLK_TYPE", qui représente un bloc fonctionnel, dispose d'une *Référence HasComponent* à une *Variable* "SP" du *VariableType* "SetPoint". "AI_BLK_TYPE" peut avoir une *Variable* de point de consigne supplémentaire du même type utilisant un nom différent. Il peut ajouter à la *Variable* une *Propriété* qui n'était pas définie par son *TypeDefinitionNode* "SetPoint". Et il peut également définir une valeur par défaut pour "SP", de sorte que chaque instance de "AI_BLK_TYPE" aurait une *Variable* "SP" initialement configurée sur cette valeur.

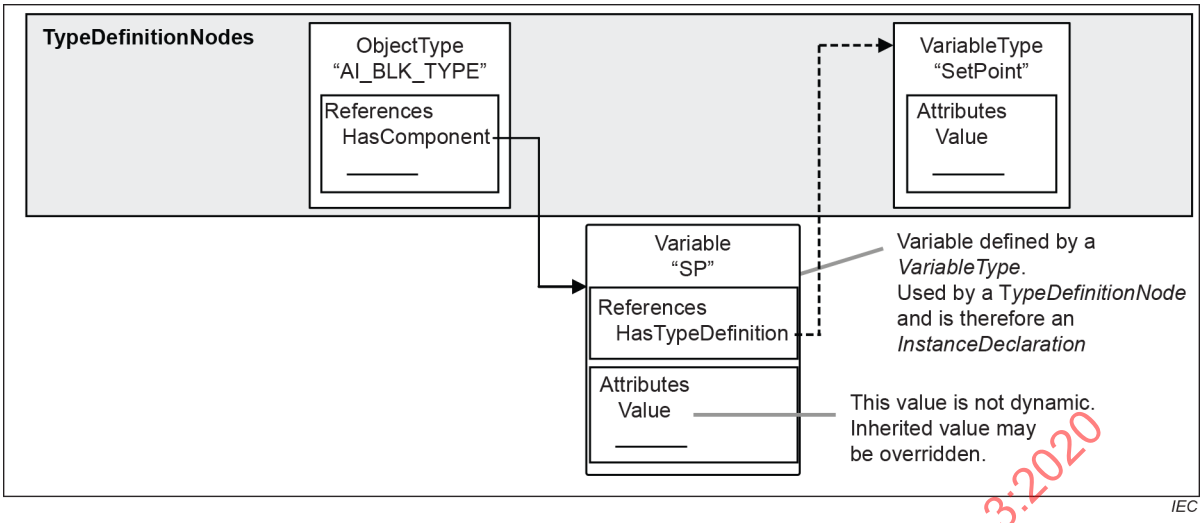


Figure 6 – Exemple d'une TypeDefinition complexe

Cette approche est souvent utilisée dans les langages de programmation orientés objet qui définissent les variables d'une classe comme des instances d'autres classes. Lorsque la classe est instanciée, chaque variable est également instanciée, mais en utilisant les valeurs par défaut (valeurs du constructeur) définies pour la classe conteneur. En d'autres termes, de manière générale, le constructeur de la classe composante vient en premier, suivi du constructeur de la classe conteneur. Un constructeur de classe conteneur peut prendre le pas sur les valeurs de composant définis par la classe composante.

Afin de faire la différence entre les instances utilisées pour les définitions de type et celles qui représentent des données réelles, ces instances sont appelées *InstanceDeclarations*. Ce terme est cependant utilisé pour simplifier la présente spécification; il est uniquement possible de déterminer si une instance est ou n'est pas une *InstanceDeclaration* dans l'*AddressSpace,* en suivant ses *Références*. Certaines instances peuvent être partagées et, par conséquent, référencées par des *TypeDefinitionNodes*, des *InstanceDeclarations* et des instances. Cette méthodologie est similaire aux variables de classe dans les langages de programmation orientés objet.

4.5.3 Sous-typage

Le présent document permet le sous-typage de définitions de type. Les règles de sous-typage sont définies à l'Article 6. Avec le sous-typage d'*ObjectTypes* et de *VariableTypes*, il est admis:

- aux *Clients* qui connaissent uniquement le super-type de traiter une instance du sous-type comme s'il s'agissait d'une instance du super-type;
- de remplacer des instances du super-type par des instances du sous-type;
- d'utiliser des types spécialisés qui héritent des caractéristiques communes du type de base.

En d'autres termes, les sous-types reflètent la structure définie par leur super-type, mais peuvent apporter des caractéristiques supplémentaires. Par exemple, un fournisseur peut souhaiter étendre un *VariableType* général "TemperatureSensor" en ajoutant une *Propriété* qui indique la périodicité de maintenance. Pour cela, le fournisseur crée un nouveau *VariableType* qui est un *TargetNode* pour une référence *HasSubtype* à partir du *VariableType* initial et y ajoute la nouvelle *Propriété*.

4.5.4 Instanciation de TypeDefinitionNodes complexes

L'instanciation de *TypeDefinitionNodes* complexes dépend des *ModellingRules* définies en 6.4.4. Il est cependant prévu que les instances d'une définition de type reflètent la

structure définie par le *TypeDefinitionNode*. La Figure 7 présente une instance du *TypeDefinitionNode* "AI_BLK_TYPE" auquel la *ModellingRule Mandatory*, définie en 6.4.4.5.2, a été appliquée à sa *Variable* conteneur. Ainsi, une instance de "AI_BLK_TYPE", appelée "AI_BLK_1", dispose d'une *Référence HasTypeDefinition* vers "AI_BLK_TYPE". Elle contient également une *Variable* "SP" ayant le même *BrowseName* que la *Variable* "SP" utilisée par le *TypeDefinitionNode* et, par conséquent, reflète la structure définie par le *TypeDefinitionNode*.

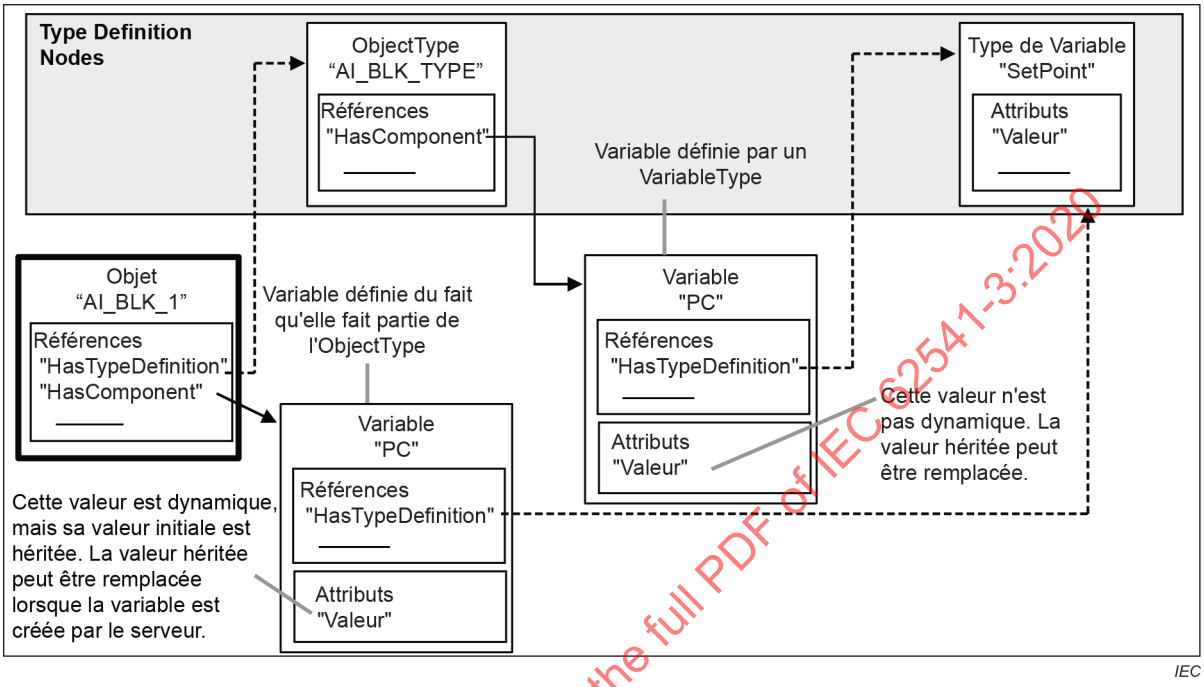


Figure 7 – Objet et ses composants définis par un ObjectType

Un *Client* connaissant l'*ObjectType* "AI_BLK_TYPE" peut utiliser cette information pour explorer directement les *Nœuds* contenants pour chaque instance de ce type. Ceci permet une programmation tenant compte du *TypeDefinitionNode*. Par exemple, un élément graphique peut être programmé dans le *Client* qui traite toutes les instances de "AI_BLK_TYPE" de la même manière en présentant la valeur de "SP".

Une programmation selon le *TypeDefinitionNode* présente plusieurs contraintes. Un *TypeDefinitionNode* ou une *InstanceDeclaration* ne doit jamais référencer deux *Nœuds* ayant un même *BrowseName* en utilisant des *Références hiérarchiques* directes. Les instances reposant sur des *InstanceDeclarations* doivent toujours conserver le même *BrowseName* que l'*InstanceDeclaration* dont elles sont issues. Un *Service* spécial défini dans l'IEC 62541-4, appelé TranslateBrowsePathsToNodeIds, peut être utilisé pour identifier les instances sur la base des *InstanceDeclarations*. La simple utilisation du *Service* Browse peut ne pas suffire, car l'unicité du *BrowseName* est uniquement exigée pour les *TypeDefinitionNodes* et les InstanceDeclarations, et non pour d'autres instances. Ainsi, "AI_BLK_1" peut disposer d'une autre *Variable* portant le *BrowseName* "SP", même si celle-ci ne résulte pas d'une *InstanceDeclaration* du *TypeDefinitionNode*.

Les instances obtenues à partir d'une *InstanceDeclaration* doivent avoir le même *TypeDefinitionNode* ou un sous-type de ce *TypeDefinitionNode*.

Un *TypeDefinitionNode* et ses *InstanceDeclarations* doivent toujours résider sur le même *Serveur*. Cependant, des instances peuvent pointer, avec leur *Référence HasTypeDefinition*, vers un *TypeDefinitionNode* situé dans un *Serveur* différent.

4.6 Modèle d'événement

4.6.1 Généralités

Le Modèle d'événement définit un système de traitement des événements d'usage général qui peut servir dans de nombreux marchés verticaux différents.

Les *Evénements* représentent des occurrences transitoires spécifiques. Les modifications de configuration système et les erreurs système sont des exemples d'*Evénements*. Les *Notifications d'événements* rendent compte de l'occurrence d'un *Événement* donné. Les *Evénements* définis dans le présent document ne sont pas directement visibles dans l'*AddressSpace* OPC UA. Des *Objets* et des *Vues* peuvent être utilisés pour s'abonner à des *Evénements*. L'*Attribut EventNotifier* indique si le *Nœud* peut s'abonner à des *Evénements*. Les *Clients* s'abonnent à ces *Nœuds* afin de recevoir des *Notifications* sur l'occurrence d'*Evénements*.

L'*Abonnement aux événements* utilise les *Services* Subscription/MonitoredItem définis dans l'IEC 62541-4 pour s'abonner aux *Notifications d'événements* d'un *Nœud*.

Tout *Serveur* OPC UA qui prend en charge le traitement des événements doit présenter au moins un *Nœud* comme *EventNotifier*. L'*Objet Serveur* défini dans l'IEC 62541-5 est utilisé à cet effet. Les *Evénements* générés par le *Serveur* sont disponibles par l'intermédiaire de cet *Objet Serveur*. Un *Serveur* n'est pas supposé produire des *Evénements* si, pour quelque raison que ce soit (c'est-à-dire si le système est hors ligne), la connexion à la source de l'événement est interrompue.

Des *Evénements* peuvent également être présentés par l'intermédiaire d'autres *Nœuds*, partout dans l'*AddressSpace*. Ces *Nœuds* (identifiés par l'*Attribut EventNotifier*) fournissent un sous-ensemble des *Evénements* générés par le *Serveur*. La position dans l'*AddressSpace* indique ce qu'est ce sous-ensemble. Par exemple, un *Objet* zone de processus, qui représente une zone fonctionnelle du processus, fournit des *Evénements* provenant uniquement de cette zone de processus. Il convient de noter qu'il s'agit seulement d'un exemple et qu'il incombe entièrement au *Serveur* de déterminer les *Evénements* qu'il convient que tel ou tel *Nœud* fournisse.

4.6.2 EventTypes

Chaque *Événement* correspond à un *EventType* spécifique. Un *Serveur* peut prendre en charge plusieurs types. Le présent document définit le *BaseEventType* dont sont issus tous les autres *EventTypes*. Il est prévu que d'autres spécifications associées définissent des *EventTypes* supplémentaires qui permettent d'obtenir les types de base définis dans le présent document.

Les *EventTypes* pris en charge par un *Serveur* sont présentés dans l'*AddressSpace* du *Serveur*. Les *EventTypes* sont représentés comme des *ObjectTypes* dans l'*AddressSpace* et il ne leur est pas associé de *NodeClass* spéciale. L'IEC 62541-5 définit précisément la manière dont un *Serveur* présente les *EventTypes*.

Les *EventTypes* définis dans le présent document sont spécifiés comme abstraits et, par conséquent, ils ne sont jamais instanciés dans l'*AddressSpace*. L'occurrence des Evénements de ces *EventTypes* est uniquement présentée par l'intermédiaire d'un *Abonnement*. Il existe dans l'*AddressSpace* des *EventTypes* qui autorisent que les *Clients* découvrent l'*EventType*. Cette information est utilisée par un *Client* lorsqu'il établit et utilise des *Abonnements* aux *Evénements*. Les *EventTypes* définis par d'autres parties de l'IEC 62541 ou par des spécifications associées, ainsi que les *EventTypes* spécifiques au *Serveur*, peuvent être définis comme non abstraits; par conséquent, les instances de ces *EventTypes* peuvent être visibles dans l'*AddressSpace* même si les *Evénements* de ces *EventTypes* sont également accessibles par l'intermédiaire des mécanismes de *Notification* d'*Événement*.

Les *EventTypes* normalisés sont décrits à l'Article 9. Leur représentation dans l'*AddressSpace* est spécifiée dans l'IEC 62541-5.

4.6.3 Catégorisation des Evénements

Les *Evénements* peuvent être classés par catégories en créant de nouveaux *EventTypes* qui sont des sous-types d'*EventTypes* existants, mais qui n'étendent pas un type existant. Ils sont uniquement utilisés pour identifier un événement comme appartenant au nouvel *EventType*. Par exemple, l'*EventType* DeviceFailureEventType peut avoir comme sous-type TransmitterFailureEventType et ComputerFailureEventType. Ces nouveaux sous-types ne sont pas intégrés comme de nouvelles *Propriétés* ni ne modifient la sémantique héritée du DeviceFailureEventType; il s'agit purement et simplement d'une catégorisation des *Evénements*.

Les sources d'*Evénement* peuvent également être organisées en groupes, en utilisant les *ReferenceTypes* d'*événement* décrits en 7.17 et 7.18. Par exemple, un *Serveur* peut définir des *Objets* dans l'*AddressSpace* représentant des *Evénements* liés aux appareils physiques ou des zones d'*Evénements* d'une installation ou une fonctionnalité contenue dans le *Serveur*. Les *Références d'événement* sont utilisées pour indiquer quelles sont les sources d'*Événement* qui représentent des appareils physiques et celles qui représentent une certaine fonctionnalité fondée sur le *Serveur*. En outre, des *Références* peuvent être utilisées pour grouper les appareils physiques ou les fonctionnalités fondées sur le *Serveur* en zones d'*Evénements* hiérarchisées. Dans certains cas, une source d'*Evénement* peut être catégorisée comme étant à la fois un appareil et une fonction du *Serveur*. Dans ce cas, deux relations sont établies. Pour des exemples supplémentaires, voir la description des *ReferenceTypes d'événements*.

Les *Clients* peuvent choisir une ou plusieurs catégories d'*Evénements* en définissant des filtres de contenu qui incluent des termes spécifiant l'*EventType* de l'*Événement* ou un groupement de sources d'*Événement*. Ces deux mécanismes autorisent la catégorisation d'un *Événement* particulier de plusieurs manières. Un *Client* peut obtenir tous les *Evénements* correspondant à un appareil physique ou toutes les défaillances d'un appareil particulier.

4.7 Méthodes

Les *Méthodes* sont des fonctions "légères" dont le champ d'application est limité par un *Objet* appartenance (voir Note); elles sont similaires aux méthodes d'une classe en programmation orientée objet ou d'un *ObjectType* appartenance, ainsi qu'aux méthodes statiques d'une classe. Les *Méthodes* sont invoquées par un *Client*; elles sont menées à bien sur le *Serveur* et les résultats sont renvoyés au *Client*. La durée de vie d'une instance d'invocation de *Méthode* commence lorsque le *Client* appelle la *Méthode* et se termine lorsque le résultat lui est renvoyé.

NOTE L'*Objet* ou l'*ObjectType* appartenance est spécifié dans l'appel de service, lorsque la *Méthode* est invoquée.

Même si les *Méthodes* peuvent affecter l'état de l'*Objet* appartenance, elles n'ont pas d'état explicite propre. De ce fait, elles sont sans état. Les *Méthodes* peuvent avoir un nombre variable d'arguments d'entrée et renvoyer des arguments résultants. Chaque *Méthode* est décrite par un *Nœud* de la *NodeClass Méthode*. Ce *Nœud* contient les métadonnées qui identifient les arguments de la *Méthode* et décrivent son comportement.

Les *Méthodes* sont invoquées par le *Service* Appel défini dans l'IEC 62541-4.

Les *Clients* trouvent les *Méthodes* prises en charge par un *Serveur* en explorant les *Références des Objets* appartenance qui identifient les *Méthodes* prises en charge.

4.8 Rôles

4.8.1 Vue d'ensemble

Un *Rôle* est une fonction assumée par un *Client* lorsqu'il accède à un *Serveur*. Les *Rôles* sont utilisés pour distinguer les authentifications (qui déterminent l'identité d'un *Client*) des autorisations (qui déterminent les actions que le *Client* peut réaliser). En séparant ces tâches, les *Serveurs* peuvent permettre la gestion, par les services centralisés, des identités et des informations d'accès des utilisateurs, tandis que le *Serveur* gère uniquement les *Autorisations* sur ses *Nœuds* attribués aux *Rôles*.

L'ensemble des *Rôles* pris en charge par un *Serveur* sont publiés en tant que composants de l'*Objet Rôles* défini dans l'IEC 62541-5. Il convient que les *Serveurs* définissent un ensemble de base de *Rôles* et autorisent les *Clients* de configuration à ajouter des *Rôles* spécifiques au système.

Lors de la création d'une *Session*, le *Serveur* doit déterminer quels *Rôles* sont attribués à cette *Session*. Le présent document définit les règles de mapping normalisées que les *Serveurs* peuvent prendre en charge. Les *Serveurs* peuvent aussi utiliser des règles de mapping spécifiques au fournisseur en plus ou à la place des règles normalisées.

Les règles de mapping normalisées autorisent l'attribution des *Rôles* sur la base des éléments suivants:

- l'identité de l'utilisateur;
- l'identité de l'application;
- le point d'extrémité.

Le mapping de l'identité des utilisateurs peut s'appuyer sur les noms, les certificats ou les groupes d'utilisateurs. Les groupes notoires comprennent "AuthenticatedUser" (tout utilisateur avec des authentifiants d'utilisateur valides) et "Anonymous" (sans authentifiants d'utilisateur fournis).

Le mapping de l'identité des applications s'appuie sur l'*ApplicationUri* spécifié dans le *Certificat Client*. L'identité de l'application ne peut être appliquée que si le *Client* prouve être en possession d'un *Certificat* de confiance en l'utilisant pour créer un *Canal sécurisé* ou en fournissant une signature dans *ActivateSession* (voir IEC 62541-4).

Le mapping de l'identité des points d'extrémité s'appuie sur l'URL utilisée pour se connecter au *Serveur*. L'identité du point d'extrémité peut être utilisée pour restreindre l'accès aux *Clients* sur certains réseaux.

L'IEC 62541-5 définit les *Objets*, les *Méthodes* et les *DataTypes* utilisés pour représenter et gérer ces règles de mapping dans l'*AddressSpace*.

4.8.2 Rôles notoires

Il convient que tous les *Serveurs* prennent en charge les *Rôles* notoires définis dans le Tableau 2. Les *NodeIds* pour les *Rôles* notoires sont définis dans l'IEC 62541-6.

Tableau 2 – Rôles notoires

BrowseName	Autorisations suggérées
Anonymous	Le *Rôle* a un accès très limité lorsqu'une *Session* possède des authentifiants d'utilisateur anonymes.
AuthenticatedUser	Le *Rôle* a un accès limité lorsqu'une *Session* possède des authentifiants d'utilisateur valides et non anonymes, mais ne bénéficie pas explicitement d'un accès à un *Rôle*.
Observer	Le *Rôle* peut naviguer, lire des données en direct, lire des données/événements historiques ou s'abonner à des données/événements.
Operator	Le *Rôle* peut naviguer, lire des données en direct, lire des données/événements historiques ou s'abonner à des données/événements. La *Session* peut en outre écrire des données en direct et appeler certaines *Méthodes*.
Engineer	Le *Rôle* peut naviguer, lire/écrire des données de configuration, lire des données/événements historiques, appeler des *Méthodes* ou s'abonner à des données/événements.
Supervisor	Le *Rôle* peut naviguer, lire des données en direct, lire des données/événements historiques, appeler des *Méthodes* ou s'abonner à des données/événements.
ConfigureAdmin	Le *Rôle* peut modifier les paramètres de configuration non liés à la sécurité.
SecurityAdmin	Le *Rôle* peut modifier les paramètres liés à la sécurité.

4.8.3 Evaluation des Autorisations avec les Rôles

Lorsqu'un *Client* essaie d'accéder à un *Nœud*, le *Serveur* analyse la liste des *Rôles* attribués à la *Session* et réunit logiquement les *Autorisations* pour le *Rôle* sur le *Nœud. En l'absence d'Autorisation spécifique à un Nœud, alors* les *Autorisations* par défaut pour le *Rôle* dans la *Propriété DefaultRolePermissions* de l'Objet *NamespaceMetadata* pour l'espace de noms auquel le *Nœud* appartient sont utilisées (voir IEC 62541-5). Le masque qui en résulte correspond aux *Autorisations* effectives. Si les bits qui correspondent à l'opération en cours sont définis, l'opération peut être exécutée. S'ils ne sont pas définis, le *Serveur* renvoie *Bad_UserAccessDenied*.

Les *Rôles* apparaissent sous l'*Objet Rôles* dans l'*AddressSpace* du *Serveur*. Chaque *Rôle* a des règles de mapping définies qui apparaissent en tant que *Propriétés* de l'*Objet Rôles* (voir IEC 62541-5). Les exemples fournis dans le Tableau 3 présentent la manière dont les règles de mapping normalisées peuvent être utilisées pour déterminer les *Rôles* auxquels une *Session* a accès et, par conséquent, les *Autorisations* qui sont attribuées à la *Session*.

Tableau 3 – Exemples de Rôles

Rôle	Règles de mapping	Description
Anonymous	Identités = Anonymous Applications = Points d'extrémité =	Règle de mapping de l'identité qui spécifie que le *Rôle* s'applique aux utilisateurs anonymes.
AuthenticatedUser	Identités = AuthenticatedUser Applications = Points d'extrémité =	Règle de mapping de l'identité qui spécifie que le *Rôle* s'applique aux utilisateurs authentifiés.
Operator1	Identités = Utilisateur dont le nom est "Joe" Applications = urn:OperatorStation1 Points d'extrémité =	Règle de mapping de l'identité qui spécifie des utilisateurs spécifiques qui ont accès au *Rôle* avec une règle d'application qui restreint l'accès à une seule application *Client*.
Operator2	Identités = Utilisateurs dont le nom est "Joe" ou "Ann" Applications = urn:OperatorStation2 Points d'extrémité =	Règle de mapping de l'identité qui spécifie des utilisateurs spécifiques qui ont accès au *Rôle* avec une règle d'application qui restreint l'accès à une seule application *Client*.
Supervisor	Identités = Utilisateur dont le nom est "Root" Applications = Points d'extrémité =	Règle de mapping de l'identité qui spécifie des utilisateurs spécifiques qui ont accès au *Rôle*
Administrateur	Identités = Utilisateur dont le nom est "Root" Applications = Points d'extrémité = opc.tcp://127.0.0.1:48000	Règle de mapping de l'identité qui spécifie des utilisateurs spécifiques qui ont accès au *Rôle* lorsqu'ils se connectent via un point d'extrémité spécifique.

Les exemples font également appel aux *Nœuds* définis dans le Tableau 4. Ce tableau spécifie la valeur de l'*Attribut RolePermissions* pour chaque *Nœud*.

Tableau 4 – Exemples de Nœuds

Nœud	Autorisations de Rôle
Unit1.Measurement	AuthenticatedUser = navigation Operator1 = navigation, lecture
Unit2.Measurement	AuthenticatedUser = navigation Operator2 = navigation, lecture
SetPoint	AuthenticatedUser = navigation Operator1 et Operator2 = navigation, lecture, écriture Supervisor = navigation, lecture
DisableDevice	AuthenticatedUser = navigation Operator1 et Operator2 = navigation, lecture Administrator = navigation, lecture, écriture

Lorsqu'un *Client* crée une *Session*, les *Rôles* attribués à la *Session* dépendent des règles définies pour chaque *Rôle*. Le Tableau 5 répertorie les *Rôles* attribués aux différentes *Sessions* créées avec différents *Utilisateurs*, applications *Client* et points d'extrémité.

Tableau 5 – Exemple d'attribution de Rôle

Utilisateur fourni par le Client	Rôles attribués à la Session
Anonymous	Anonymous
Sam	AuthenticatedUser
Joe utilisant l'application OperatorStation1.	AuthenticatedUser, Operator1
Joe utilisant l'application OperatorStation2.	AuthenticatedUser, Operator2
Joe utilisant l'application générique.	AuthenticatedUser
Root utilisant l'application OperatorStation1.	AuthenticatedUser, Supervisor
Root utilisant l'application générique et le point d'extrémité 127.0.0.1.	AuthenticatedUser, Supervisor, Administrator
Root utilisant l'application générique et un autre point d'extrémité.	AuthenticatedUser, Supervisor

Lorsqu'une application *Client* a accès à un *Nœud*, les *RolePermissions* pour ce *Nœud* sont comparées aux *Rôles* attribués à la *Session*. Toute *Autorisation* disponible pour au moins un *Rôle* est accordée au *Client*. Le Tableau 6 présente plusieurs scénarios et exemples, ainsi que la décision sur l'accès qui en découle.

Tableau 6 – Exemples d'évaluations d'accès

Cas d'utilisation	Autorisations de Rôle
Un utilisateur anonyme sur l'hôte local navigue sur le *Nœud* Unit1.Measurement.	Accès refusé, car aucune règle n'est définie pour les utilisateurs anonymes.
L'utilisateur "Sam" qui utilise l'application OperatorStation1 navigue sur le *Nœud* Unit1.Measurement.	Accès autorisé, car AuthenticatedUser a l'Autorisation de navigation.
L'utilisateur "Sam" qui utilise l'application OperatorStation2 lit la *Valeur* du *Nœud* Unit1.Measurement.	Accès refusé, car AuthenticatedUser n'a pas l'*Autorisation* de lecture.
L'utilisateur "Joe" qui utilise l'application OperatorStation1 lit la *Valeur* du *Nœud* Unit1.Measurement.	Accès autorisé, car Operator1 a l'*Autorisation* de lecture.
L'utilisateur "Joe" qui utilise l'application OperatorStation2 lit la *Valeur* du *Nœud* Unit1.Measurement.	Accès refusé, car AuthenticatedUser et Operator2 n'ont pas l'*Autorisation* de lecture.
L'utilisateur "Joe" qui utilise l'application générique OPC UA lit la *Valeur* du *Nœud* Measurement.	Accès refusé, car AuthenticatedUser n'a pas l'*Autorisation* de lecture.
L'utilisateur "Joe" qui utilise l'application OperatorStation1 écrit la *Valeur* du *Nœud* SetPoint.	Accès autorisé, car Operator1 a l'*Autorisation* d'écriture.
L'utilisateur "Root" qui utilise l'application OperatorStation1 écrit la *Valeur* du *Nœud* SetPoint.	Accès refusé, car AuthenticatedUser et Supervisor n'ont pas l'*Autorisation* d'écriture.
L'utilisateur "Joe" qui utilise l'application OperatorStation1 écrit la *Valeur* du *Nœud* DisableDevice.	Accès refusé, car AuthenticatedUser et Operator1 n'ont pas l'*Autorisation* d'écriture.
L'utilisateur "Root" qui utilise l'application OperatorStation1 écrit la *Valeur* du *Nœud* DisableDevice.	Accès refusé, car AuthenticatedUser et Supervisor n'ont pas l'*Autorisation* d'écriture.
L'utilisateur "Root" qui utilise le point d'extrémité 127.0.0.1 pour écrire la *Valeur* du *Nœud* DisableDevice.	Accès autorisé, car l'Administrateur a l'*Autorisation* d'écriture.

5 NodeClasses normalisées

5.1 Vue d'ensemble

L'Article 5 définit les *NodeClasses* utilisées pour définir les *Nœuds* dans l'*AddressSpace* OPC UA. Les *NodeClasses* sont issues d'une *NodeClass Base* commune. Cette *NodeClass* est définie en premier lieu, suivie de celles utilisées pour organiser l'*AddressSpace*, puis des *NodeClasses* utilisées pour représenter les *Objets*.

Les *NodeClasses* définies pour représenter des *Objets* s'inscrivent dans trois catégories: celles utilisées pour définir les instances, celles utilisées pour définir les types d'instances et celles utilisées pour définir les types de données. Le 6.3 décrit les règles de sous-typage et le 6.4 indique les règles d'instanciation des définitions de type.

5.2 NodeClass Base

5.2.1 Généralités

Le Modèle d'espace d'adressage OPC UA définit une *NodeClass Base* dont sont issues toutes les autres *NodeClasses*. Les *NodeClasses* dérivées représentent les différents composants du Modèle d'objet OPC UA (voir 4.2). Les *Attributs* de la *NodeClass Base* sont spécifiés dans le Tableau 7. Aucune *Référence* n'est spécifiée pour la *NodeClass Base*.

Tableau 7 – NodeClass Base

Nom	Usage	Type de données	Description
Attributs			
NodeId	M	NodeId	Voir 5.2.2
NodeClass	M	NodeClass	Voir 5.2.3
BrowseName	M	QualifiedName	Voir 5.2.4
DisplayName	M	LocalizedText	Voir 5.2.5
Description	O	LocalizedText	Voir 5.2.6
WriteMask	O	AttributeWriteMask	Voir 5.2.7
UserWriteMask	O	AttributeWriteMask	Voir 5.2.8
RolePermissions	O	RolePermissionType[]	Voir 5.2.9
UserRolePermissions	O	RolePermissionType[]	Voir 5.2.10
AccessRestrictions	O	AccessRestrictionsType	Voir 5.2.11
Références			Aucune *Référence* n'est spécifiée pour cette *NodeClass*

5.2.2 NodeId

Les *Nœuds* sont identifiés de manière univoque par un identificateur construit appelé *NodeId*. Certains *Serveurs* peuvent accepter d'autres *NodeIds* outre le *NodeId* canonique représenté dans cet *Attribut*. Un *Serveur* doit conserver le *NodeId* d'un *Nœud*; autrement dit, il ne doit pas générer de nouveaux *NodeIds* lors de la réinitialisation. La structure du *NodeId* est définie en 8.2.

5.2.3 NodeClass

L'*Attribut NodeClass* identifie la *NodeClass* d'un *Nœud*. Le type de données correspondant est défini en 8.30.

5.2.4 BrowseName

Les *Nœuds* disposent d'un *Attribut BrowseName* utilisé comme dénomination non localisée et lisible par l'homme lorsqu'il explore l'*AddressSpace* pour créer des chemins à partir des *BrowseNames*. Le *Service* TranslateBrowsePathsToNodeIds défini dans l'IEC 62541-4 peut être utilisé pour suivre un chemin constitué de *BrowseNames*.

Il convient de ne jamais utiliser un *BrowseName* pour afficher le nom d'un *Nœud*. Il convient d'utiliser pour cela le *DisplayName*.

Contrairement aux *NodeIds*, le *BrowseName* ne peut pas être utilisé pour identifier de manière univoque un *Nœud*. Différents *Nœuds* peuvent utiliser le même *BrowseName*.

Le 8.3 définit la structure du *BrowseName*. Il contient un espace de noms et une chaîne. L'espace de noms permet d'assurer l'unicité du *BrowseName* dans certains cas, dans le contexte d'un *Nœud* particulier (par exemple, les *Propriétés* d'un *Nœud*), même s'il n'est pas unique dans le contexte du *Serveur*. Si différents organismes définissent des *BrowseNames* pour les *Propriétés*, l'espace de noms du *BrowseName* fourni par l'organisme rend le *BrowseName* unique, même si différents organismes peuvent utiliser la même chaîne avec une signification légèrement différente.

Les *serveurs* peuvent souvent choisir d'utiliser le même espace de noms pour le *NodeId* et le *BrowseName*. Cependant, s'ils souhaitent fournir une *Propriété* normalisée, son *BrowseName* doit avoir l'espace de noms de l'organisme de normalisation, même si l'espace de noms du *NodeId* reflète quelque chose de différent, par exemple le *Serveur* local.

Il est recommandé que les organismes de normalisation qui établissent des définitions de type normalisées utilisent leur espace de noms pour le *NodeId* d'un *TypeDefinitionNode* ainsi que le *BrowseName* du *TypeDefinitionNode*.

La partie chaîne du *BrowseName* est sensible à la casse. C'est-à-dire que les *Clients* doivent les interpréter comme sensibles à la casse. Les *Serveurs* peuvent traiter les *BrowseNames* transmis dans les demandes de *Service* comme non sensibles à la casse. Le *Service* TranslateBrowsePathsToNodeIds ou le filtre *Événement* en sont des exemples.

5.2.5 DisplayName

L'*Attribut DisplayName* contient la dénomination localisée du *Nœud*. Il convient que les *Clients* utilisent cet *Attribut* s'ils souhaitent afficher le nom du *Nœud* pour l'utilisateur. Il convient qu'ils n'utilisent pas le *BrowseName* à cet effet. Le *Serveur* peut maintenir une ou plusieurs représentations localisées pour chaque *DisplayName*. Les *Clients* négocient les paramètres de lieu à renvoyer lorsqu'ils ouvrent une session sur le *Serveur*. Pour une description de l'établissement de session et des paramètres de lieu, voir IEC 62541-4. Le 8.5 définit la structure du *DisplayName*. La partie chaîne du *DisplayName* est limitée à 512 caractères.

5.2.6 Description

L'*Attribut* facultatif *Description* doit expliquer la signification du *Nœud* dans un texte localisé en utilisant les mêmes mécanismes de localisation que ceux décrits pour le *DisplayName* en 5.2.5.

5.2.7 WriteMask

L'*Attribut* facultatif *WriteMask* présente au *Client* les possibilités d'écriture des *Attributs* du *Nœud*. L'*Attribut WriteMask* ne tient compte d'aucun droit d'accès utilisateur, ce qui signifie que même si un *Attribut* est inscriptible, cette possibilité peut être limitée à certains utilisateurs/groupes d'utilisateurs.

Si le *Serveur* OPC UA n'a pas la possibilité d'obtenir les informations *WriteMask* d'un *Attribut* spécifique à partir du système sous-jacent, il convient de déclarer qu'il est inscriptible. Si une opération d'écriture est appelée pour l'*Attribut*, il convient que le *Serveur* transfère cette requête et renvoie le *StatusCode* correspondant si cette requête est rejetée. Les *StatusCodes* sont définis dans l'IEC 62541-4.

Le *DataType AttributeWriteMask* est défini en 8.60.

5.2.8 UserWriteMask

L'*Attribut* facultatif *UserWriteMask* présente à un *Client* les possibilités d'écriture des *Attributs* du *Nœud* en tenant compte des droits d'accès de l'utilisateur. Il utilise le *DataType AttributeWriteMask* défini en 8.60.

L'*attribut UserWriteMask* peut étendre la restriction de l'*attribut WriteMask* lorsqu'il est défini comme non inscriptible dans le cas général applicable à chaque utilisateur.

Les *Clients* ne peuvent pas prendre pour hypothèse qu'un *Attribut* peut être écrit sur la base de l'*Attribut UserWriteMask*. Le *Serveur* peut renvoyer une erreur d'accès refusé en raison d'une modification spécifique d'un *Serveur* qui n'a pas été reflétée dans l'état de cet *Attribut* au moment où le *Client* y a accédé.

5.2.9 RolePermissions

L'*Attribut* facultatif *RolePermissions* spécifie les *Autorisations* qui s'appliquent à un *Nœud* pour tous les *Rôles* qui ont accès à ce *Nœud*. La valeur de l'*Attribut* est une matrice des *Structures RolePermissionType* (voir Tableau 8).

Tableau 8 – RolePermissionType

Nom	Type	Description
RolePermissionType	Structure	Spécifie les *Autorisations* pour un *Rôle*.
roleId	NodeId	*NodeId* de l'*Objet* du *Rôle*.
permissions	PermissionType	Masque qui spécifie quelles *Autorisations* sont à la disposition du *Rôle*.

Les *Serveurs* peuvent autoriser l'écriture, par les administrateurs, sur l'*Attribut RolePermissions*.

Si cela n'est pas spécifié, la valeur de la *Propriété DefaultRolePermissions* de l'*Objet NamespaceMetadata* associé au *Nœud* doit être utilisée à la place. Si l'*Objet NamespaceMetadata* ne définit pas la *Propriété* ou n'existe pas, alors il convient que le *Serveur* ne publie aucune information sur la manière dont il gère les *Autorisations*.

Si un *Serveur* prend en charge des *Autorisations* pour un *Namespace* particulier, il doit ajouter la *Propriété DefaultRolePermissions* à l'*Objet NamespaceMetadata* pour cet *Espace de noms* (voir Figure 8). S'il est nécessaire qu'un *Nœud* particulier dans l'*Espace de noms* prenne le pas sur les valeurs par défaut, le *Serveur* ajoute l'*Attribut RolePermissions* au *Nœud.* La *Propriété DefaultRolePermissions* et l'*Attribut RolePermissions* doivent pouvoir être lus uniquement par les administrateurs. Si un *Serveur* autorise la modification des *Autorisations*, ces valeurs doivent être inscriptibles. Si le *Serveur* autorise le remplacement des *Autorisations* pour un *Nœud* particulier, mais n'a actuellement pas d'*Autorisation* de *Nœud* configurée, alors la valeur de l'*Attribut* doit être une matrice vide. Si l'administrateur souhaite retirer les *Autorisations* remplacées, une matrice vide doit être créée pour cet *Attribut*. Les *Serveurs* doivent empêcher les *Autorisations* d'être modifiées d'une manière qui rende le *Serveur* inopérant.

Si un *Serveur* publie des informations sur les *Rôles* pour un *Espace de noms* attribué à la *Session* en cours, il doit ajouter la *Propriété DefaultUserRolePermissions* à l'*Objet NamespaceMetadata* pour cet *Espace de noms.* La valeur de cette *Propriété* doit être une liste d'*Autorisations* en lecture seule pour chaque *Rôle* attribué à la *Session* en cours. S'il est nécessaire qu'un *Nœud* particulier dans l'*Espace de noms* prenne le pas sur les *RolePermissions* par défaut, le *Serveur* doit également prendre le pas sur les *DefaultUserRolePermissions* en ajoutant l'*Attribut UserRolePermissions* au *Nœud.* Si le *Serveur* autorise le remplacement des *Autorisations* pour un *Nœud* particulier, mais n'a pas pour l'heure actuelle d'*Autorisation* de *Nœud* configurée, alors le Serveur doit renvoyer la valeur de la *Propriété DefaultUserRolePermissions* pour l'*Espace de noms du Nœud.*

Si un *Serveur* met en œuvre un modèle d'*Autorisation* de *Rôle* spécifique à un fournisseur pour un *Espace de noms*, il ne doit pas ajouter les *Propriétés DefaultRolePermissions* ou *DefaultUserRolePermissions* à l'*Objet Namespace Metadata*.

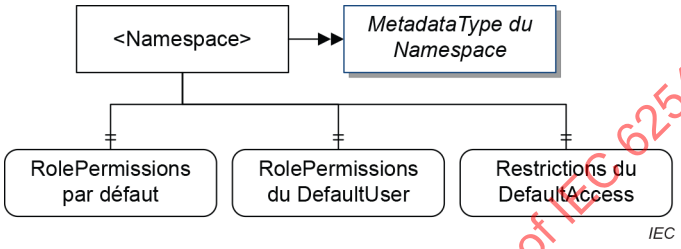


Figure 8 – Autorisations dans l'Espace d'adressage

5.2.10 UserRolePermissions

L'*Attribut* facultatif *UserRolePermissions* spécifie les *Autorisations* qui s'appliquent à un *Nœud* pour tous les *Rôles* attribués à la *Session* en cours. La valeur de l'*Attribut* est une matrice des *Structures RolePermissionType* (voir Tableau 8).

Les *Clients* peuvent déterminer leurs *Autorisations* effectives en réunissant logiquement les *Autorisations* pour chaque *Rôle* dans la matrice.

La valeur de cet *Attribut* est obtenue à partir des règles utilisées par le *Serveur* pour mapper les *Sessions* avec les *Rôles*. Ce mapping peut être spécifique à un fournisseur ou peut utiliser le modèle de *Rôle* normalisé défini en 4.8.

Cet *Attribut* ne doit pas être inscriptible.

Si cela n'est pas spécifié, la valeur de la *Propriété DefaultUserRolePermissions* de l'*Objet NamespaceMetadata* associé au *Nœud* est utilisée à la place. Si l'*Objet NamespaceMetadata* ne définit pas la *Propriété* ou n'existe pas, alors le *Serveur* ne publie aucune information sur les *Rôles* mappés avec la *Session* en cours.

5.2.11 AccessRestrictions

L'*Attribut* facultatif *AccessRestrictions* spécifie les *AccessRestrictions* qui s'appliquent à un *Nœud.* Le type de données correspondant est défini en 8.56. Si un *Serveur* prend en charge des *AccessRestrictions* pour un *Espace de noms* particulier, il ajoute la *Propriété DefaultAccessRestrictions* à l'*Objet NamespaceMetadata* pour cet *Espace de noms* (voir Figure 8). S'il est nécessaire qu'un *Nœud* particulier dans l'*Espace de noms* prenne le pas sur les valeurs par défaut, le *Serveur* ajoute l'*Attribut AccessRestrictions* au *Nœud.*

Si un *Serveur* met en œuvre un modèle de restriction d'accès spécifique à un fournisseur pour un *Espace de noms*, il n'ajoute pas la *Propriété DefaultAccessRestrictions* à l'*Objet NamespaceMetadata*.

5.3 NodeClass ReferenceType

5.3.1 Généralités

Les *Références* sont définies comme des instances de *Nœuds ReferenceType*. Les *Nœuds ReferenceType* sont visibles dans l'*AddressSpace* et sont définis en utilisant la *NodeClass ReferenceType* comme spécifié dans le Tableau 9. En revanche, une *Référence* est une partie inhérente d'un *Nœud* et aucune *NodeClass* n'est utilisée pour représenter les *Références*.

Le présent document définit un ensemble de *ReferenceTypes* fournis comme parties inhérentes du Modèle d'espace d'adressage OPC UA. Ces *ReferenceTypes* sont définis dans l'Article 7 et leur représentation dans l'*AddressSpace* est définie dans l'IEC 62541-5. Les *Serveurs* peuvent également définir des *ReferenceTypes*. Par ailleurs, l'IEC 62541-4 définit les *Services NodeManagement* qui autorisent que les *Clients* ajoutent des *ReferenceTypes* à l'*AddressSpace*.

Tableau 9 – NodeClass ReferenceType

Nom	Usage	Type de données	Description
Attributs			
Attributs de la NodeClass Base	M	--	Hérités de la *NodeClass Base*. Voir 5.2.
IsAbstract	M	Boolean	*Attribut* booléen prenant les valeurs suivantes: TRUE Il s'agit d'un *ReferenceType* abstrait, c'est-à-dire qu'aucune *Référence* de ce type ne doit exister, mais uniquement ses sous-types. FALSE Il s'agit d'un *ReferenceType* qui n'est pas abstrait, c'est-à-dire qu'il peut exister des *Références* de ce type.
Symmetric	M	Boolean	*Attribut* booléen prenant les valeurs suivantes: TRUE La signification du *ReferenceType* est la même que celle qui est perçue à la fois du point de vue du *SourceNode* et du *TargetNode.* FALSE La signification du *ReferenceType* telle que perçue du point de vue du *TargetNode* est l'inverse de celle qui est perçue à partir du *SourceNode*.
InverseName	O	LocalizedText	Dénomination inverse de la *Référence*, c'est-à-dire la signification du *ReferenceType* telle qu'elle est perçue à partir du *TargetNode*.
Références			
HasProperty	0..*		Utilisé pour identifier les propriétés (voir 5.3.3.2).
HasSubtype	0..*		Utilisé pour identifier les sous-types (voir 5.3.3.3).
Propriétés normalisées			
NodeVersion	O	String	La *Propriété NodeVersion* est utilisée pour indiquer la version d'un *Nœud*. La *Propriété NodeVersion* est mise à jour à chaque fois qu'une *Référence* est ajoutée au ou retirée du *Nœud* auquel la *Propriété* appartient. Les modifications de la valeur de l'*Attribut* n'entraînent pas une modification de la *NodeVersion*. Les *Clients* peuvent lire la *Propriété NodeVersion* ou s'y abonner pour déterminer le moment où la structure d'un *Nœud* a varié.

5.3.2 Attributs

La *NodeClass ReferenceTypes* hérite des *Attributs* de base de la *NodeClass Base* définie en 5.2. L'*Attribut BrowseName* hérité est utilisé pour préciser la signification du *ReferenceType* telle que perçue à partir du *SourceNode*. Par exemple, le *ReferenceType* portant le *BrowseName* "Contains" est utilisé dans les *Références* pour indiquer que le *SourceNode* contient le *TargetNode*. L'*Attribut DisplayName* hérité contient une traduction du *BrowseName*.

Le *BrowseName* d'un *ReferenceType* doit être unique dans un *Serveur*. Il n'est pas admis que deux *ReferenceTypes* différents portent le même *BrowseName*.

L'*Attribut IsAbstract* indique que ce *ReferenceType* est abstrait. Les *ReferenceTypes* abstraits ne peuvent être instanciés et sont utilisés uniquement pour des raisons d'organisation, par exemple pour spécifier certaines caractéristiques sémantiques ou contraintes d'ordre général dont ont hérité ses sous-types.

L'*Attribut Symmetric* est utilisé pour indiquer si la signification du *ReferenceType* est (ou n'est pas) identique à la fois pour le *SourceNode* et le *TargetNode*.

Si un *ReferenceType* est symétrique, l'*Attribut InverseName* doit être omis. Les exemples de *ReferenceTypes* symétriques sont notamment "Connects To" et "Communicates With". Les deux impliquent la même sémantique en provenance du *SourceNode* ou du *TargetNode*. Par conséquent, les deux directions sont considérées comme des *Références* directes.

Si le *ReferenceType* est non symétrique et non abstrait, l'*Attribut InverseName* doit être configuré. L'*Attribut InverseName* spécifie la signification du *ReferenceType* telle que perçue à partir du *TargetNode*. Les exemples de *ReferenceTypes* non symétriques sont notamment "Contains" et "Contained In", et "Receives From" et "Sends To".

Les *Références* qui utilisent l'*InverseName*, par exemple les *Références* "Contained In", sont appelées *Références* inverses.

La Figure 9 fournit des exemples de *Références* symétriques et non symétriques, ainsi que d'utilisation du *BrowseName* et de l'*InverseName*.

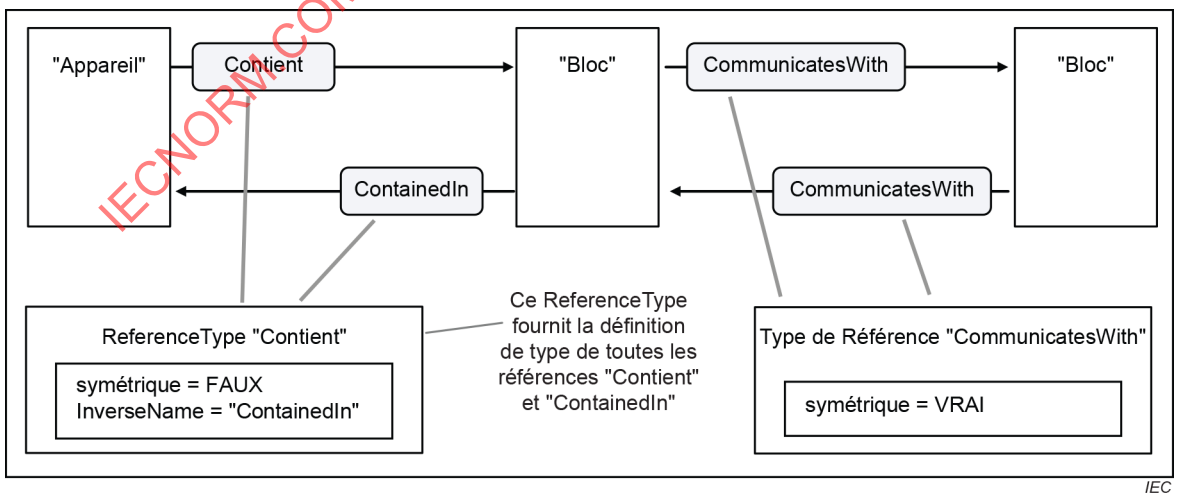


Figure 9 – Références symétriques et non symétriques

Il est admis qu'il ne soit pas toujours possible pour les *Serveurs* d'instancier à la fois des *Références* directes et inverses pour des *ReferenceTypes* non symétriques, comme représenté à la Figure 9. Lorsque c'est le cas, les *Références* sont appelées *bidirectionnelles*. Bien que cela ne soit pas exigé, il est recommandé que toutes les *Références hiérarchiques*

soient instanciées de manière bidirectionnelle afin d'assurer la connectivité de la navigation. Une *Référence* bidirectionnelle est modélisée comme deux *Références* séparées.

Exemple de *Référence unidirectionnelle*: il arrive souvent qu'un collecteur de signaux connaisse la source de signaux, sans que cette source de signaux connaisse le collecteur de signaux. Le collecteur de signaux aurait la *Référence* "Sourced By" pour la source des signaux, la source des signaux ayant pour sa part les *Références* inverses correspondantes "Sourced To" pour ses collecteurs de signaux.

Le *DisplayName* et l'*InverseName* sont les seuls emplacements normalisés utilisés pour indiquer la sémantique d'un *ReferenceType*. Il peut exister plusieurs règles sémantiques complexes associées à un *ReferenceType* qui peuvent être exprimées dans ces *Attributs* (par exemple, la sémantique de *HasSubtype*). Le présent document ne spécifie pas la manière dont il convient d'exprimer cette sémantique. L'*Attribut Description* peut cependant être utilisé à cet effet. Le présent document fournit une règle sémantique pour les *ReferenceTypes* spécifiés à l'Article 7.

Un *ReferenceType* peut avoir des contraintes qui limitent son utilisation. Il peut spécifier par exemple qu'en partant du *Nœud* A et uniquement en suivant les *Références* de ce *ReferenceType* ou de l'un de ses sous-types, il ne doit jamais être possible de revenir à A; c'est-à-dire qu'il s'agit d'une contrainte "No Loop".

Le présent document ne spécifie pas la manière dont il est possible ou dont il convient de mettre à disposition ces contraintes dans l'*AddressSpace*. Cependant, pour les *ReferenceTypes* normalisés, certaines contraintes sont spécifiées à l'Article 7. Le présent document ne limite pas le type de contraintes utilisables pour un *ReferenceType*. La contrainte peut, par exemple, affecter également un *ObjectType*. La restriction selon laquelle un *ReferenceType* peut uniquement être utilisé par des *Nœuds* de certaines *NodeClasses* ayant une cardinalité définie est une contrainte particulière d'un *ReferenceType*.

5.3.3 Références

5.3.3.1 Généralités

Les *Références HasSubtype* et *HasProperty* sont les seuls *ReferenceTypes* qui peuvent être utilisés avec les *Nœuds ReferenceType* tels que le *SourceNode*. Les *Nœuds ReferenceType* ne doivent pas être le *SourceNode* d'autres types de *Références*.

5.3.3.2 Références HasProperty

Les *Références HasProperty* sont utilisées pour identifier les *Propriétés* d'un *ReferenceType* et ne doivent se référer qu'aux *Nœuds* de la *NodeClass Variables.*

La *Propriété NodeVersion* est utilisée pour indiquer la version du *ReferenceType*.

Il n'existe pas d'autres *Propriétés* définies pour les *ReferenceTypes* dans le présent document. D'autres parties de l'IEC 62541 peuvent définir des *Propriétés* supplémentaires pour les *ReferenceTypes*.

5.3.3.3 Références HasSubtype

Les *Références HasSubtype* sont utilisées pour définir des sous-types de *ReferenceTypes*. Il n'est pas exigé de fournir la *Référence HasSubtype* pour le super-type, mais ce sous-type doit fournir la *Référence* inverse à son super-type. Les règles suivantes s'appliquent au sous-typage.

a) La sémantique d'un *ReferenceType* (par exemple "couvre une hiérarchie") est héritée par ses sous-types et peut être affinée (par exemple "couvre une hiérarchie spéciale"). Le *DisplayName*, ainsi que l'*InverseName* pour les *ReferenceTypes* non symétriques, reflètent cette spécialisation.

b) Si un *ReferenceType* spécifie certaines contraintes (par exemple "n'autorise aucune boucle"), cette restriction est héritée et ne peut qu'être rendue plus sévère (par exemple la restriction héritée "aucune boucle" peut être affinée en "doit être une arborescence – un parent unique"), mais non réduite (par exemple "autoriser les boucles").

c) Les contraintes concernant les *NodeClasses* qui peuvent être référencées sont également héritées et peuvent uniquement être plus strictes. En d'autres termes, si un *ReferenceType* A n'est pas autorisé à relier un *Objet* à un *ObjectType*, ceci est également vrai pour tous ses sous-types.

d) Un *ReferenceType* doit avoir exactement un super-type, sauf en ce qui concerne le *ReferenceType Références* défini en 7.2 comme étant le type racine de la hiérarchie du *ReferenceType*. La hiérarchie du *ReferenceType* ne prend pas en charge plusieurs héritages multiples.

5.4 NodeClass Vue

Les systèmes sous-jacents sont souvent de grande taille et les *Clients* n'ont souvent besoin que d'un sous-ensemble spécifique de données. Il n'est pas nécessaire de les surcharger avec des *Nœuds* de visualisation dans un *AddressSpace* qui ne les intéresse aucunement.

Pour résoudre ce problème, le présent document définit le concept de *Vue*. Chaque *Vue* définit un sous-ensemble de *Nœuds* dans *l'AddressSpace*. L'*AddressSpace* dans son ensemble est la *Vue* par défaut. Chaque *Nœud* dans une *Vue* peut contenir uniquement un sous-ensemble de ses *Références*, comme défini par le créateur de la *Vue*. Le *Nœud Vue* agit comme la racine des *Nœuds* dans la *Vue*. Les *Vues* sont définies au moyen de la *NodeClass Vues* spécifiée dans le Tableau 10.

Tous les *Nœuds* contenus dans une *Vue* doivent être accessibles à partir du *Nœud Vue* lorsqu'une navigation est effectuée dans le contexte de la *Vue*. Tous les *Nœuds* contenants ne sont pas supposés pourvoir être explorés directement à partir du *Nœud Vue*, mais plutôt d'autres *Nœuds* contenus dans la *Vue*.

Un *Nœud Vue* peut ne pas être uniquement utilisé comme entrée supplémentaire dans l'*AddressSpace*, mais comme une construction permettant d'organiser l'*AddressSpace* et ainsi comme le seul point d'entrée dans un sous-ensemble de l'*AddressSpace*. Par conséquent, les *Clients* ne doivent pas ignorer les *Nœuds Vues* lorsqu'ils présentent l'*AddressSpace*. Les *Clients* simples qui n'utilisent pas les *Vues* à des fins de filtrage peuvent, par exemple, traiter un *Nœud Vue* comme un type d'*Objet* de type *FolderType* (voir 5.5.3).

Tableau 10 – NodeClass Vue

<table>
<tr><th>Nom</th><th>Usage</th><th>Type de données</th><th>Description</th></tr>
<tr><td>Attributs</td><td></td><td></td><td></td></tr>
<tr><td>Attributs de la NodeClass Base</td><td>M</td><td>--</td><td>Hérités de la NodeClass Base. Voir 5.2.</td></tr>
<tr><td>ContainsNoLoops</td><td>M</td><td>Boolean</td><td>S'il est configuré sur True, cet Attribut indique que les Références suivantes dans le contexte de la Vue ne contiennent aucune boucle; en d'autres termes, à partir d'un Nœud A contenu dans la Vue et suivant les Références directes dans le contexte du Nœud Vue, A n'est jamais atteint de nouveau. Ceci ne signifie pas qu'il n'existe qu'un seul chemin à partir du Nœud Vue pour atteindre un Nœud contenu dans la Vue.

S'il est configuré sur False, cet Attribut indique que les Références suivantes dans le contexte de la Vue peuvent donner lieu à des boucles.</td></tr>
<tr><td>EventNotifier</td><td>M</td><td>Byte</td><td>L'Attribut EventNotifier est utilisé pour indiquer si le Nœud peut être utilisé pour s'abonner à des Evénements ou pour lire/écrire des Evénements historiques.

L'EventNotifier est un entier de 8 bits non signé dont la structure est définie dans le tableau suivant.
<table>
<tr><th>Champ</th><th>Bit</th><th>Description</th></tr>
<tr><td>SubscribeTo Events</td><td>0</td><td>Indique s'il peut être utilisé pour s'abonner à des Evénements
(0 signifie qu'il ne peut pas être utilisé pour s'abonner à des Evénements, 1 signifie qu'il peut être utilisé pour s'abonner à des Evénements)</td></tr>
<tr><td>Reserved</td><td>1</td><td>Réservé pour une utilisation ultérieure. Doit toujours être zéro.</td></tr>
<tr><td>HistoryRead</td><td>2</td><td>Indique si l'historique des Evénements est lisible (0 signifie non lisible, 1 signifie lisible).</td></tr>
<tr><td>HistoryWrite</td><td>3</td><td>Indique si l'historique des Evénements est inscriptible
(0 signifie non inscriptible, 1 signifie inscriptible).</td></tr>
<tr><td>Reserved</td><td>4:7</td><td>Réservé pour une utilisation ultérieure. Doit toujours être zéro.</td></tr>
</table>
Les deux bits suivants indiquent également si l'historique des Evénements est disponible via le Serveur OPC UA.</td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td>Références</td><td></td><td></td><td></td></tr>
<tr><td>HierarchicalReferences</td><td>0..*</td><td></td><td>Les Nœuds de niveau supérieur dans une Vue sont référencés par des Références hiérarchiques (voir 7.3).</td></tr>
<tr><td>HasProperty</td><td>0..*</td><td></td><td>Les Références HasProperty identifient les Propriétés de la Vue.</td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td>Propriétés normalisées</td><td></td><td></td><td></td></tr>
<tr><td>NodeVersion</td><td>O</td><td>String</td><td>La Propriété NodeVersion est utilisée pour indiquer la version d'un Nœud.

La Propriété NodeVersion est mise à jour à chaque fois qu'une Référence est ajoutée au ou retirée du Nœud auquel la Propriété appartient. Les modifications de la valeur de l'Attribut n'entraînent pas une modification de la NodeVersion. Les Clients peuvent lire la Propriété NodeVersion ou s'y abonner pour déterminer le moment où la structure d'un Nœud a varié.</td></tr>
<tr><td>ViewVersion</td><td>O</td><td>UInt32</td><td>Le numéro de version de la Vue. Lorsque des Nœuds sont ajoutés ou retirés d'une Vue, la valeur de la Propriété ViewVersion est mise à jour. Les Clients peuvent détecter d'éventuelles modifications dans la composition d'une Vue en utilisant cette Propriété. La valeur de la ViewVersion doit toujours être supérieure à 0.</td></tr>
</table>

La *NodeClass Vue* hérite des *Attributs* de base de la *NodeClass Base* définie en 5.2. Deux *Attributs* supplémentaires sont également spécifiés.

L'*Attribut* obligatoire *ContainsNoLoops* est configuré sur False si le *Serveur* est incapable de savoir si la *Vue* contient ou non des boucles.

L'*Attribut* obligatoire *EventNotifier* indique si la *Vue* peut être utilisée pour s'abonner à des *Evénements* qui ont lieu soit dans le contenu de la *Vue*, soit en tant que *ModelChangeEvents* (voir 9.32) du contenu de la *Vue*; il est également utilisé pour lire/écrire l'historique des *Evénements*. Une *Vue* prenant en charge des *Evénements* doit prévoir tous les *Evénements* qui ont lieu dans un quelconque *Objet* utilisé comme *EventNotifier* faisant partie du contenu de la *Vue*. En outre, il doit tenir compte de tous les *ModelChangeEvents* qui ont lieu dans le contexte de la *Vue*.

Pour éviter les répétitions, c'est-à-dire obtenir tous les *Evénements* du *Serveur*, l'*Objet Serveur* défini dans l'IEC 62541-5 ne doit jamais faire partie d'une quelconque *Vue* étant donné qu'il fournit tous les *Evénements* du *Serveur*.

Les *Vues* sont définies par le *Serveur*. Les *Services* de navigation et d'interrogation définis dans l'IEC 62541-4 attendent de recevoir le *NodeId* d'un *Nœud Vue* pour fournir ces *Services* dans le contexte de la *Vue*.

Les *Références HasProperty* sont utilisées pour identifier les *Propriétés* d'une *Vue*. La *Propriété NodeVersion* est utilisée pour indiquer la version du *Nœud Vue*. La *Propriété ViewVersion* est utilisée pour indiquer la version du contenu de la *Vue*. Contrairement à la *NodeVersion,* la *Propriété ViewVersion* est mise à jour même si les *Nœuds* qui ne sont pas directement référencés par le *Nœud Vue* sont ajoutés ou retirés de la *Vue*. Cette *Propriété* est facultative, car il est possible que certains *Serveurs* ne puissent pas détecter les modifications dans le contenu de la *Vue*. Les *Serveurs* peuvent également générer un *ModelChangeEvent* (décrit en 9.32) si des *Nœuds* sont ajoutés ou supprimés de la *Vue*. Il n'existe pas d'autres *Propriétés* définies pour les *Vues* dans le présent document. D'autres parties de l'IEC 62541 peuvent définir des *Propriétés* supplémentaires pour les *Vues*.

Les *Vues* peuvent être le *SourceNode* d'une quelconque *Référence hiérarchique*. Elles ne doivent pas être le *SourceNode* d'une quelconque *Référence* non hiérarchique.

5.5 Objets

5.5.1 NodeClass Objet

Les *Objets* sont utilisés pour représenter des systèmes, des composants système, des objets concrets et des objets logiciels. Les *Objets* sont définis en utilisant la *NodeClass Objet* spécifiée dans le Tableau 11.

Tableau 11 – NodeClass Objet

Nom	Usage	Type de données	Description
Attributs			
Attributs de la NodeClass Base	M	--	Hérités de la *NodeClass Base*. Voir 5.2.
EventNotifier	M	EventNotifierType	L'*Attribut EventNotifier* est utilisé pour indiquer si le *Nœud* peut être utilisé pour s'abonner à des *Evénements* ou pour lire/écrire des *Evénements* historiques. L'*EventNotifierType* est défini en 8.59.
Références			
HasComponent	0..*		Les *Références HasComponent* identifient les *DataVariables*, les *Méthodes* et les *Objets* contenus dans l'*Objet*.
HasProperty	0..*		Les *Références HasProperty* identifient les *Propriétés* de l'*Objet*.
HasModellingRule	0..1		Les *Objets* peuvent pointer vers, au plus, un *Objet ModellingRule* en utilisant une *Référence HasModellingRule* (pour plus d'informations concernant les *ModellingRules*, voir 6.4.4).
HasTypeDefinition	1		La *Référence HasTypeDefinition* pointe vers la définition de type de l'*Objet*. Chaque *Objet* doit avoir précisément une définition de type et, par conséquent, être le *SourceNode* d'une seule et unique *Référence HasTypeDefinition* pointant vers un *ObjectType*. voir 4.5 pour une description des définitions de types.
HasEventSource	0..*		Les *Références HasEventSource* pointent vers les sources d'événement de l'*Objet*. Les *Références* de ce type peuvent uniquement être utilisées pour des *Objets* dont le bit "SubscribeToEvents" est défini dans l'*Attribut EventNotifier*. voir 7.17 pour plus d'informations.
HasNotifier	0..*		La *Référence HasNotifier* pointe vers les notificateurs de l'*Objet*. Les *Références* de ce type peuvent uniquement être utilisées pour des *Objets* dont le bit "SubscribeToEvents" est défini dans l'*Attribut EventNotifier*. voir 7.18 pour plus d'informations.
Organise	0..*		Il convient d'utiliser cette *Référence* uniquement pour des *Objets* de l'*ObjectType FolderType* (voir 5.5.3).
<autres références>	0..*		Les *Objets* peuvent contenir d'autres *Références*.
Propriétés normalisées			
NodeVersion	O	String	La *Propriété NodeVersion* est utilisée pour indiquer la version d'un *Nœud*. La *Propriété NodeVersion* est mise à jour à chaque fois qu'une *Référence* est ajoutée au ou retirée du *Nœud* auquel la *Propriété* appartient. Les modifications de la valeur de l'*Attribut* n'entraînent pas une modification de la *NodeVersion*. Les *Clients* peuvent lire la *Propriété NodeVersion* ou s'y abonner pour déterminer le moment où la structure d'un *Nœud* a varié.
Icon	O	Image	La *Propriété Icon* fournit une image qui peut être utilisée par les *Clients* lorsqu'ils affichent le *Nœud*. La *propriété Icon* est supposée contenir une image relativement petite.
NamingRule	O	NamingRuleType	La Propriété *NamingRule* définit la *NamingRule* d'une *ModellingRule* (voir 6.4.4.2 pour plus d'informations). Cette *Propriété* doit uniquement être utilisée pour des *Objets* de type *ModellingRuleType* défini en 6.4.4.

La *NodeClass Objet* hérite des *Attributs* de base de la *NodeClass Base* définie en 5.2.

L'*Attribut* obligatoire *EventNotifier* est utilisé pour indiquer si l'*Objet* peut être utilisé pour s'abonner à des *Evénements* ou pour lire et écrire l'historique des Evénements.

La *NodeClass Objet* utilise la *Référence HasComponent* pour définir les *DataVariables*, les *Objets* et les *Méthodes* d'un *Objet*.

Elle utilise la *Référence HasProperty* pour définir les *Propriétés* d'un *Objet*. La *Propriété NodeVersion* est utilisée pour indiquer la version de l'*Objet*. La *Propriété Icon* fournit une icône de l'*Objet*. La *Propriété NamingRule* définit la *NamingRule* d'une *ModellingRule* et doit uniquement être appliquée à des *Objets* de type *ModellingRuleType*. Il n'existe pas d'autres *Propriétés* définies pour les *Objets* dans le présent document. D'autres parties de l'IEC 62541 peuvent définir des *Propriétés* supplémentaires pour les *Objets*.

Pour spécifier sa *ModellingRule*, un *Objet* peut utiliser au plus une *Référence HasModellingRule* pointant vers un *Objet ModellingRule*. Les *ModellingRules* sont définies en 6.4.4.

Les *Références HasNotifier* et *HasEventSource* sont utilisées pour fournir des informations concernant les événements et ne peuvent être appliquées qu'aux *Objets* utilisés comme notificateurs d'événements. Une description précise est fournie en 7.17 et 7.18.

La *Référence HasTypeDefinition* pointe vers l'*ObjectType* utilisé comme définition de type de l'*Objet*.

Les *Objets* peuvent utiliser des *Références* supplémentaires pour définir des relations avec d'autres *Nœuds*. Aucune contrainte n'est appliquée aux types de *Références* utilisés ou aux *NodeClasses* des *Nœuds* pouvant être référencés. Cependant, le *ReferenceType* peut définir des restrictions qui excluent son utilisation pour des *Objets*. Les *ReferenceTypes* normalisés sont décrits à l'Article 7.

Si l'*Objet* est utilisé comme une *InstanceDeclaration* (voir 4.5), tous les *Nœuds* référencés avec des *Références hiérarchiques* directes doivent alors avoir des *BrowseNames* uniques dans le contexte de cet *Objet*.

Si l'*Objet* est créé sur la base d'une *InstanceDeclaration*, il doit alors avoir le même *BrowseName* que son *InstanceDeclaration*.

5.5.2 NodeClass ObjectType

Les *ObjectTypes* fournissent des définitions pour les *Objets*. Les *ObjectTypes* sont définis en utilisant la *NodeClass ObjectType* qui est spécifiée dans le Tableau 12.

Tableau 12 – NodeClass ObjectType

Nom	Usage	Type de données	Description
Attributs			
Attributs de la NodeClass Base	M	--	Hérités de la *NodeClass Base*. Voir 5.2.
IsAbstract	M	Boolean	*Attribut* booléen prenant les valeurs suivantes: TRUE Il s'agit d'un *ObjectType* abstrait, ce qui signifie qu'aucun *Objet* de ce type ne doit exister, mais uniquement des *Objets* de ses sous-types. FALSE Il s'agit d'un *ObjectType* non abstrait, c'est-à-dire qu'il peut exister des *Objets* de ce type.
Références			
HasComponent	0..*		Les *Références HasComponent* identifient les *DataVariables*, les *Méthodes* et les *Objets* contenus dans l'*ObjectType*. Le 6.4 spécifie l'éventuelle instanciation des *Nœuds* référencés et la manière dont cette instanciation est réalisée lorsqu'un *Objet* de ce type est instancié.
HasProperty	0..*		Les *Références HasProperty* identifient les *Propriétés* de l'*ObjectType*. Le 6.4 spécifie l'éventuelle instanciation des *Propriétés* et la manière dont cette instanciation est réalisée lorsqu'un *Objet* de ce type est instancié.
HasSubtype	0..*		Les *Références HasSubtype* indiquent les *ObjectTypes* qui sont des sous-types de ce type. La *Référence* inverse *SubtypeOf* identifie le type parent de ce type.
GeneratesEvent	0..*		Les *Références GeneratesEvent* identifient le type d'instance d'*Evénements* que ce type peut générer.
<autres références>	0..*		Les *ObjectTypes* peuvent contenir d'autres *Références* qui peuvent être instanciées par des *Objets* définis par cet *ObjectType*.
Propriétés normalisées			
NodeVersion	O	String	La *Propriété NodeVersion* est utilisée pour indiquer la version d'un *Nœud*. La *Propriété NodeVersion* est mise à jour à chaque fois qu'une *Référence* est ajoutée au ou retirée du *Nœud* auquel la *Propriété* appartient. Les modifications de la valeur de l'*Attribut* n'entraînent pas une modification de la *NodeVersion*. Les *Clients* peuvent lire la *Propriété NodeVersion* ou s'y abonner pour déterminer le moment où la structure d'un *Nœud* a varié.
Icon	O	Image	La *Propriété Icon* fournit une image qui peut être utilisée par les *Clients* lorsqu'ils affichent le *Nœud*. La *propriété Icon* est supposée contenir une image relativement petite.

La *NodeClass ObjectType* hérite des *Attributs* de base de la *NodeClass Base* définie en 5.2. L'*Attribut* supplémentaire *IsAbstract* indique si l'*ObjectType* est ou non abstrait.

La *NodeClass ObjectType* utilise les *Références HasComponent* pour définir les *DataVariables*, les *Objets* et les *Méthodes* d'un *ObjectType*.

La *Référence HasProperty* est utilisée pour identifier les *Propriétés*. La *Propriété NodeVersion* est utilisée pour indiquer la version de l'*ObjectType*. La *Propriété Icon* fournit une icône de l'*ObjectType*. Il n'existe pas d'autres *Propriétés* définies pour les *ObjectTypes* dans le présent document. D'autres parties de l'IEC 62541 peuvent définir des *Propriétés* supplémentaires pour les *ObjectTypes*.

Les *Références HasSubtype* sont utilisées pour définir les sous-types des *ObjectTypes*. Les sous-types de l'*ObjectType* héritent des caractéristiques sémantiques générales du type parent. Les règles générales de sous-typage s'appliquent comme défini à l'Article 6. Il n'est

pas exigé de fournir la *Référence HasSubtype* pour le super-type, mais ce sous-type doit fournir la *Référence* inverse à son super-type.

Les *Références GeneratesEvent* identifient le type d*Evénements* que les instances de cet *ObjectType* peuvent générer. Ces *Objets* peuvent être la source d'un *Événement* du type spécifié ou d'un de ses sous-types. Il convient que les *Serveurs* fassent en sorte que les *Références GeneratesEvent* soient bidirectionnelles. Il est cependant admis qu'elles soient unidirectionnelles lorsque le *Serveur* n'est pas capable de présenter le sens inverse pointant de l'*EventType* vers chaque *ObjectType* prenant en charge l'*EventType.* L'*Attribut EventNotifier* d'un *Objet* et les *Références GeneratesEvent* de cet *ObjectType* n'ont aucune relation. Les *Objets* qui peuvent générer des *Evénements* peuvent ne pas être utilisés comme des *Objets* auxquels s'abonnent les *Clients* pour obtenir les notifications d'*Evénements* correspondantes.

Les *Références GeneratesEvent* sont facultatives, c'est-à-dire que les *Objets* peuvent générer les *Evénements* d'un *EventType* qui n'est pas présenté par cet *ObjectType*.

Les *ObjectTypes* peuvent utiliser des *Références* supplémentaires pour définir des relations avec d'autres *Nœuds*. Aucune contrainte n'est appliquée aux types de *Références* utilisés ou aux *NodeClasses* des *Nœuds* pouvant être référencés. Cependant, le *ReferenceType* peut définir des restrictions qui excluent son utilisation pour des *ObjectTypes*. Les *ReferenceTypes* normalisés sont décrits à l'Article 7.

Tous les *Nœuds* référencés avec des *Références hiérarchiques* directes doivent avoir des *BrowseNames* uniques dans le contexte d'un *ObjectType* (voir 4.5).

5.5.3 ObjectType FolderType normalisé

L'*ObjectType FolderType* est défini de manière formelle dans l'IEC 62541-5. Son objectif est de fournir des *Objets* qui n'ont aucune valeur sémantique autre que l'organisation de l'*AddressSpace*. Un *ReferenceType* particulier est introduit pour ces *Objets Folder*: le *ReferenceType Organizes*. Il convient toujours que le *SourceNode* d'une telle *Référence* soit une *Vue* ou un *Objet* d'*ObjectType FolderType*; le *TargetNode* peut être de n'importe quelle *NodeClass*. Les *Références Organizes* peuvent être utilisées en toute combinaison avec des *Références HasChild* (*HasComponent*, *HasProperty*, etc.; voir 7.5) et n'empêchent pas les boucles. Elles peuvent ainsi être utilisées pour couvrir plusieurs hiérarchies.

5.5.4 Création du côté client d'Objets d'un ObjectType donné

Les *Objets* sont toujours fondés sur un *ObjectType*; ceci signifie qu'ils ont une *Référence HasTypeDefinition* pointant vers son *ObjectType*.

Les *Clients* peuvent créer des *Objets* en utilisant le *Service AddNodes* défini dans l'IEC 62541-4. Le *Service* exige que le *TypeDefinitionNode* de l'*Objet* soit spécifié. Un *Objet* créé par le *Service AddNodes* contient tous les composants définis par son *ObjectType* en fonction des *ModellingRules* spécifiées pour les composants. Le *Serveur* peut toutefois ajouter des composants supplémentaires et des *Références* à l'*Objet* et à ses composants qui ne sont pas définis par l'*ObjectType*. Ce comportement dépend du *Serveur*. L'*ObjectType* spécifie uniquement l'ensemble minimal de composants qui doit exister pour chaque *Objet* d'un *ObjectType* donné.

Outre le *Service AddNodes*, les *ObjectTypes* peuvent disposer d'une *Méthode* spéciale avec le *BrowseName* "*Create*". Cette *Méthode* est utilisée pour créer un *Objet* de cet *ObjectType*. Cette *Méthode* peut être utile pour créer des *Objets* lorsqu'il convient que les règles sémantiques de la création soient différentes du comportement par défaut attendu dans le contexte du *Service AddNodes*. Il convient par exemple que les valeurs soient totalement différentes des valeurs par défaut ou il convient que des *Objets* supplémentaires soient ajoutés, etc. Les arguments d'entrée et de sortie de cette *Méthode* dépendent de l'*ObjectType*; la seule caractéristique commune est le *BrowseName* qui indique que cette

Méthode crée un *Objet* sur la base de l'*ObjectType*. Il convient que les *Serveurs* ne fournissent pas une *Méthode* portant sur un *ObjectType* avec le *BrowseName* "*Create*" pour des raisons autres que la création d'*Objets* de l'*ObjectType*.

5.6 Variables

5.6.1 Généralités

Deux types de *Variables* ont été définis: les *Propriétés* et les *DataVariables*. Bien qu'elles soient utilisées de manière différente (décrite en 4.4) et qu'elles aient des contraintes différentes (décrites dans la suite du 5.6), elles utilisent la même *NodeClass* (décrite en 5.6.2). Les contraintes de *Propriétés* fondées sur cette *NodeClass* sont définies en 5.6.3, les contraintes des *DataVariables* en 5.6.4.

5.6.2 NodeClass Variables

Les *Variables* sont utilisées pour représenter des valeurs qui peuvent être simples ou complexes. Les *Variables* sont définies par des *VariableTypes*, spécifiés en 5.6.5.

Les *Variables* sont toujours définies comme des *Propriétés* ou des *DataVariables* d'autres *Nœuds* de l'*AddressSpace*. Elles ne sont jamais définies par elles-mêmes. Une *Variable* fait toujours partie d'au moins un autre *Nœud*, mais elle peut être liée à n'importe quel nombre d'autres *Nœuds*. Les *Variables* sont définies en utilisant la *NodeClass Variables* spécifiée dans le Tableau 13.

Tableau 13 – NodeClass Variables

Nom	Usage	Type de données	Description
Attributs			
Attributs de la NodeClass Base	M	--	Hérités de la *NodeClass Base*. Voir 5.2.
Value	M	Défini par l'*Attribut DataType*	Valeur la plus récente de la *Variable* dont dispose le *Serveur*. Son type de données est défini par l'*Attribut DataType*. Il s'agit du seul *Attribut* auquel n'est associé aucun type de données. Ceci permet à toutes les *Variables* d'avoir une valeur définie par le même *Attribut Value*.
DataType	M	NodeId	*NodeId* de la définition du *DataType* pour l'*Attribut Value*. Les *DataTypes* normalisés sont décrits à l'Article 8.
ValueRank	M	Int32	Cet *Attribut* indique si l'*Attribut Value* de la *Variable* est une matrice et le nombre de dimensions de cette matrice. Il peut prendre les valeurs suivantes: n > 1: la valeur est une matrice dont le nombre de dimensions est spécifié. OneDimension (1): la valeur est une matrice unidimensionnelle. OneOrMoreDimensions (0): la valeur est une matrice à une ou plusieurs dimensions. Scalar (−1): la valeur n'est pas une matrice. Any (−2): la valeur peut être scalaire ou une matrice avec n'importe quel nombre de dimensions. ScalarOrOneDimension (−3): la valeur peut être scalaire ou une matrice unidimensionnelle. Tous les DataTypes sont considérés comme des valeurs scalaires, même s'ils ont une sémantique de type matrice comme ByteString et String.

Nom	Usage	Type de données	Description
ArrayDimensions	O	UInt32[]	Cet *Attribut* spécifie la longueur maximale de chaque dimension prise en charge. Si la valeur maximale n'est pas connue, la valeur doit être 0. Le nombre d'éléments doit être égal à la valeur de l'*Attribut ValueRank*. Cet *Attribut* doit être nul si le *ValueRank* ≤ 0. Par exemple, si une *Variable* est définie par la matrice C suivante: Int32 myArray[346]; ainsi, ce *DataType* de la *Variable* pointe vers une Int32, le *ValueRank* de la *Variable* prend la valeur 1 et l'*Attribut ArrayDimensions* indique une matrice dont l'une des entrées a la valeur 346. Le nombre maximal d'éléments d'une matrice transférés à la volée est 2147483647 (max Int32).
AccessLevel	M	AccessLevelType	L'*attribut AccessLevel* indique la manière dont la *Valeur* d'une *Variable* peut être accédée (lecture/écriture) et si celle-ci contient des données actuelles et/ou historiques. L'*AccessLevel* ne tient compte d'aucun droit d'accès utilisateur; ceci signifie que même si la *Variable* est inscriptible, cette possibilité peut être limitée à certains utilisateurs/groupes d'utilisateurs. L'AccessLevelType est défini en 8.57.
UserAccessLevel	M	AccessLevelType	L'*attribut AccessLevelType* indique la manière dont la *Valeur* d'une *Variable* peut être accédée (lecture/écriture) et si celle-ci contient des données actuelles ou historiques en tenant compte des droits d'accès de l'utilisateur. L'AccessLevelType est défini en 8.57.
MinimumSamplingInterval	O	Duration	L'*Attribut MinimumSamplingInterval* indique la fréquence d'actualisation de la *Valeur* de la *Variable*. Il spécifie (en millisecondes) la fréquence à laquelle le *Serveur* peut réaliser un échantillonnage raisonnable des modifications de la valeur (pour une description précise de l'intervalle d'échantillonnage, voir IEC 62541-4). Un *MinimumSamplingInterval* de 0 indique que le *Serveur* doit surveiller l'élément en continu. Un *MinimumSamplingInterval* de -1 signifie qu'il est indéterminé.
Historizing	M	Boolean	L'*Attribut Historizing* indique si le *Serveur* recueille activement les données historiques de la *Variable*. Il diffère de l'*Attribut AccessLevel* qui indique si la *Variable* dispose d'éventuelles données historiques. Une valeur TRUE indique que le *Serveur* recueille activement des données. Une valeur FALSE indique que le *Serveur* ne recueille pas activement des données. La valeur par défaut est FALSE.
AccessLevelEx	O	AccessLevelExType	L'*attribut AccessLevelEx* indique la manière dont la *Valeur* d'une *Variable* peut être accédée (lecture/écriture) et si celle-ci contient des données actuelles et/ou historiques, et son atomicité. L'*AccessLevelEx* ne tient compte d'aucun droit d'accès utilisateur, ce qui signifie que même si la *Variable* est inscriptible, cette possibilité peut être limitée à certains utilisateurs/groupes d'utilisateurs. L'*AccessLevelEx* est une version étendue de l'attribut AccessLevel et, en tant que tel, ses 8 premiers bits correspondent aux 8 bits de l'attribut AccessLevel. L'*AccessLevelEx* est un entier de 32 bits non signé dont la structure est définie en 8.58. Si cet Attribut n'est pas fourni, les informations fournies par ces Champs supplémentaires ne sont pas connues.

Nom	Usage	Type de données	Description
Références			
HasModellingRule	0..1		Les *Variables* peuvent pointer vers, au plus, un *Objet ModellingRule* en utilisant une *Référence HasModellingRule* (pour plus d'informations concernant les *ModellingRules*, voir 6.4.4).
HasProperty	0..*		Les *Références HasProperty* sont utilisées pour identifier les *Propriétés* d'une *DataVariable*. Les *Propriétés* ne peuvent pas être le *SourceNode* des *Références HasProperty*.
HasComponent	0..*		Les *Références HasComponent* sont utilisées par des *DataVariables* complexes pour identifier les *DataVariables* qui les composent. Les *Propriétés* ne peuvent pas utiliser cette *Référence*.
HasTypeDefinition	1		La *Référence HasTypeDefinition* pointe vers la définition de type de la *Variable*. Chaque *Variable* doit avoir précisément une définition de type et, par conséquent, être le *SourceNode* d'une seule et unique *Référence HasTypeDefinition* pointant vers un *VariableType*. voir 4.5 pour une description des définitions de types.
<autres références>	0..*		Les *DataVariables* peuvent être le *SourceNode* de toute autre *Référence*. Les *Propriétés* peuvent uniquement être le *SourceNode* d'une quelconque *Référence* non hiérarchique.
Propriétés normalisées			
NodeVersion	O	String	La *Propriété NodeVersion* est utilisée pour indiquer la version d'une *DataVariable*. Elle ne s'applique pas aux *Propriétés*. La *Propriété NodeVersion* est mise à jour à chaque fois qu'une *Référence* est ajoutée au ou retirée du *Nœud* auquel la *Propriété* appartient. Sauf pour ce qui concerne l'*Attribut DataType*, les modifications de la valeur de l'*Attribut* n'entraînent pas la modification de la *NodeVersion*. Les *Clients* peuvent lire la *Propriété NodeVersion* ou s'y abonner pour déterminer le moment où la structure d'un *Nœud* a varié. Bien que la relation entre une *Variable* et son *DataType* ne soit pas modélisée en utilisant des *Références*, les modifications apportées à l'*Attribut DataType* d'une *Variable* entraînent une mise à jour de la *Propriété NodeVersion*.
LocalTime	O	TimeZone DataType	La *Propriété LocalTime* est uniquement utilisée pour les *DataVariables*. Elle ne s'applique pas aux *Propriétés*. Cette *Propriété* est une structure contenant les fanions Offset et DaylightSavingInOffset. Offset spécifie la différence temporelle (en minutes) entre le SourceTimestamp (UTC) associé à la valeur et l'heure du lieu où la valeur a été obtenue. Le SourceTimestamp est défini dans l'IEC 62541-4. Si DaylightSavingInOffset est configuré sur TRUE, l'heure d'été/normale du lieu d'origine est en vigueur et Offset inclut la correction de l'heure d'été. S'il est configuré sur FALSE, Offset n'inclut pas la correction de l'heure d'été et celle-ci peut ne pas être en vigueur.
AllowNulls	O	Boolean	La *Propriété AllowNulls* est uniquement utilisée pour les *DataVariables*. Elle ne s'applique pas aux *Propriétés*. Cette *Propriété* indique si une valeur nulle est admise pour l'*Attribut Value* de la *DataVariable*. Si elle est configurée sur True, le *Serveur* peut renvoyer des valeurs nulles et accepter l'écriture de valeurs nulles. Si elle est configurée sur False, le *Serveur* ne doit jamais renvoyer de valeur nulle et doit refuser toute requête d'écriture d'une valeur nulle. Si cette *Propriété* n'est pas fournie, l'autorisation ou non des valeurs nulles est spécifique au *Serveur*.

Nom	Usage	Type de données	Description
ValueAsText	O	Localized Text	Ce paramètre est utilisé pour les *DataVariables* avec un ensemble fini de *LocalizedTexts* associés à sa valeur. Par exemple, toute *DataVariable* dont le *DataType* est *Enumeration*. Cette *Propriété* facultative fournit la représentation du texte localisé de la valeur. Elle peut être utilisée par des *Clients* qui s'intéressent seulement à l'affichage du texte pour s'abonner à la *Propriété* au lieu de l'attribut Value.
MaxStringLength	O	UInt32	Uniquement utilisé pour des *DataVariables* dont le *DataType* est *String*. Cette *Propriété* facultative indique le nombre maximal d'octets pris en charge par la *DataVariable*.
MaxCharacters	O	UInt32	Uniquement utilisé pour des *DataVariables* dont le *DataType* est *String*. Cette *Propriété* facultative indique le nombre maximal de caractères Unicode pris en charge par la *DataVariable*.
MaxByteStringLength	O	UInt32	Uniquement utilisé pour des *DataVariables* dont le *DataType* est *ByteString*. Cette *Propriété* facultative indique le nombre maximal d'octets pris en charge par la *DataVariable*.
MaxArrayLength	O	UInt32	Uniquement utilisé pour des *DataVariables* dont l'*Attribut ValueRank* n'est pas défini comme scalaire. Cette *Propriété* facultative indique la longueur maximale d'une matrice prise en charge par la *DataVariable*. Dans une matrice multidimensionnelle, elle indique la longueur totale. Par exemple une matrice tridimensionnelle de 2 x 3 x 10 a une longueur de matrice de 60. NOTE Afin de présenter la longueur d'une suite d'octets, ne pas utiliser le *DataType ByteString*, mais une matrice d'*Octets* du *DataType*. Dans ce cas, la *MaxArrayLength* s'applique.
EngineeringUnits	O	EU Information	Uniquement utilisé pour des *DataVariables* dont le *DataType* est *Number*. Cette *Propriété* facultative indique les unités techniques pour la valeur de la *DataVariable* (par exemple hertz ou secondes). L'IEC 62541-8 fournit des informations sur la *Propriété* et les unités techniques qu'il convient d'utiliser. Le *DataType EUInformation* est aussi défini dans l'IEC 62541-8.

La *NodeClass Variables* hérite des *Attributs* de base de la *NodeClass Base* définie en 5.2.

La *NodeClass Variables* définit également un ensemble d'*Attributs* qui décrit la valeur du temps d'exécution de la *Variable*. L'*Attribut Value* représente la valeur de la *Variable*. Les *Attributs DataType*, *ValueRank* et *ArrayDimensions* permettent de décrire des valeurs simples et complexes.

L'*Attribut AccessLevel* indique l'accessibilité de la *Valeur* d'une *Variable* sans tenir compte des droits d'accès de l'utilisateur. Si le *Serveur* OPC UA n'a pas la possibilité d'obtenir les informations sur l'*AccessLevel* à partir du système sous-jacent, il convient alors de déclarer qu'il est lisible et inscriptible. Si une opération de lecture ou d'écriture est invoquée pour la *Variable*, il convient alors que le *Serveur* transfère cette requête et renvoie le *StatusCode* correspondant même si une telle requête est rejetée. Les *StatusCodes* sont définis dans l'IEC 62541-4.

Le fanion *SemanticChange* de l'*Attribut AccessLevel* est utilisé pour les *Propriétés* qui peuvent varier et définir les aspects sémantiques du *Nœud* parent. Par exemple, la *Propriété* EngineeringUnits décrit la sémantique d'une *DataVariable*, au contraire de la *Propriété* Icon. Dans cet exemple, si la *Propriété* EngineeringUnits peut varier alors que le Serveur est en cours d'exécution, le fanion *SemanticChange* doit être défini en ce sens.

Les *Serveurs* qui prennent en charge les abonnements aux *Evénements* doivent générer un *SemanticChangeEvent* dès lors qu'une *Propriété*, dont le fanion *SemanticChange* est défini, évolue.

Si une *Variable*, qui a une *Propriété* dont le fanion *SemanticChange* est défini, est utilisée dans un *Abonnement* et que la valeur de la *Propriété* évolue, alors le bit *SemanticsChanged* du *StatusCode* doit être défini conformément à l'IEC 62541-4. Il convient que les *Clients* qui s'abonnent à une *Variable* examinent le *StatusCode* afin d'identifier une éventuelle modification de la sémantique et récupèrent les *Propriétés* pertinentes avant de traiter la valeur renvoyée par l'*Abonnement*.

L'*Attribut UserAccessLevel* indique l'accessibilité de la *Valeur* d'une *Variable* en tenant compte des droits d'accès de l'utilisateur. Si le *Serveur* OPC UA n'a pas la possibilité d'obtenir des informations concernant les droits d'accès de l'utilisateur à partir du système sous-jacent, il convient alors d'utiliser le même masque de bits que dans l'*Attribut AccessLevel*. L'*attribut UserAccessLevel* peut limiter l'accessibilité indiquée par l'*attribut AccessLevel*, mais ne peut pas la dépasser. Il convient que les *Clients* ne considèrent pas que les droits d'accès sont fondés sur l'*Attribut UserAccessLevel*. Par exemple, il est possible que le *Serveur* renvoie une erreur en raison d'une modification spécifique à un serveur qui n'a pas été reflétée dans l'état de cet *Attribut* au moment où le *Client* a accédé à la *Variable*.

L'*attribut MinimumSamplingInterval* spécifie la rapidité à laquelle le *Serveur* peut réaliser un échantillonnage raisonnable des modifications de la valeur. L'exactitude de cette valeur (l'aptitude du *Serveur* à atteindre ses performances "optimales") peut être fortement affectée par la charge du système et d'autres facteurs.

L'*Attribut Historizing* indique si le *Serveur* recueille activement les données historiques de la *Variable*. Voir IEC 62541-11 pour plus d'informations sur les *Variables* d'historisation.

Les *Clients* peuvent lire ou écrire des valeurs de *Variables*, ou surveiller la modification des valeurs, comme spécifié dans l'IEC 62541-4. L'IEC 62541-8 définit des règles supplémentaires d'utilisation des *Services* pour l'automatisation des données.

Pour spécifier sa *ModellingRule*, une *Variable* peut utiliser au plus une *Référence HasModellingRule* pointant vers un *Objet ModellingRule*. Les *ModellingRules* sont définies en 6.4.4.

Si la *Variable* est créée sur la base d'une *InstanceDeclaration* (voir 4.5), elle doit avoir le même *BrowseName* que son *InstanceDeclaration*.

Les autres *Références* sont décrites séparément pour les *Propriétés* et *DataVariables* dans la suite du 5.6.

5.6.3 Propriétés

Les *Propriétés* sont utilisées pour définir les caractéristiques des *Nœuds*. Les *Propriétés* sont définies en utilisant la *NodeClass Variables* spécifiée dans le Tableau 13. Cependant, elles limitent leur utilisation.

Les *Propriétés* sont les feuilles de toute arborescence hiérarchique; par conséquent, elles ne doivent pas être le *SourceNode* d'éventuelles *Références hiérarchiques*. Elles comprennent la *Référence HasComponent* ou *HasProperty*; ceci signifie que des *Propriétés* ne contiennent pas elles-mêmes de *Propriétés* et ne peuvent présenter leur structure complexe. Cependant, elles peuvent être le *SourceNode* de toute *Référence* non hiérarchique.

La *Référence HasTypeDefinition* pointe vers le *VariableType* de la *Propriété*. Sachant que les *Propriétés* sont identifiées de manière unique par leur *BrowseName*, toutes les *Propriétés* doivent pointer vers le *PropertyType* défini dans l'IEC 62541-5.

Les *Propriétés* doivent toujours être définies dans le contexte d'un autre *Nœud* et doivent être le *TargetNode* d'au moins une *Référence HasProperty*. Pour les différencier des *DataVariables*, elles ne doivent pas être le *TargetNode* de toute *Référence* "*HasComponent*". Ainsi, une *Référence HasProperty* pointant vers un *Nœud Variable* définit ce *Nœud* comme une *Propriété*.

Le *BrowseName* d'une *Propriété* est toujours unique dans le contexte d'un *Nœud*. Il n'est pas admis qu'un *Nœud* fasse référence à deux *Variables* en utilisant des *Références HasProperty* dont le *BrowseName* est identique.

5.6.4 DataVariable

Les *DataVariables* représentent le contenu d'un *Objet*. Les *DataVariables* sont définies en utilisant la *NodeClass Variables* spécifiée dans le Tableau 13.

Les *DataVariables* identifient leurs *Propriétés* en utilisant des *Références HasProperty*. Les *DataVariables* complexes utilisent les *Références HasComponent* pour présenter les DataVariables qui les composent.

La *Propriété NodeVersion* indique la version de la *DataVariable*.

La *Propriété LocalTime* indique la différence entre le SourceTimestamp de la valeur et l'heure normale du lieu où la valeur a été obtenue.

La *Propriété AllowNulls* indique si les valeurs nulles sont admises pour l'*Attribut Value*.

La *Propriété ValueAsText* fournit une représentation du texte localisé pour les valeurs d'énumération.

La *Propriété MaxStringLength* indique le nombre maximal d'octets d'une valeur *String*. Si un *Serveur* n'impose pas un nombre maximal d'octets ou n'est pas en mesure de déterminer le nombre maximal d'octets, cette *Propriété* ne doit pas être fournie. Si cette *Propriété* est fournie, alors la *Propriété MaxCharacters* ne doit pas être fournie.

La *Propriété MaxCharacters* indique le nombre maximal de caractères Unicode d'une valeur de chaîne. Si un *Serveur* n'impose pas un nombre maximal de caractères Unicode ou n'est pas en mesure de déterminer le nombre maximal de caractères Unicode, cette *Propriété* ne doit pas être fournie. Si cette *Propriété* est fournie, alors la *Propriété MaxStringLength* ne doit pas être fournie.

La *Propriété MaxByteStringLength* indique le nombre maximal d'octets d'une valeur *ByteString*. Si un *Serveur* n'impose pas un nombre maximal d'octets ou n'est pas en mesure de déterminer le nombre maximal d'octets, cette *Propriété* ne doit pas être fournie.

La *Propriété MaxArrayLength* indique la longueur maximale de matrice admise de la valeur.

La *Propriété EngineeringUnits* indique les unités techniques de la valeur. Il n'existe pas d'autres *Propriétés* définies pour les *DataVariables* dans le présent document. D'autres parties de l'IEC 62541 peuvent définir des *Propriétés* supplémentaires pour les *DataVariables*. L'IEC 62541-8 définit un ensemble de *Propriétés* qui peuvent être utilisées pour les *DataVariables*.

Les *DataVariables* peuvent utiliser des *Références* supplémentaires pour définir des relations avec d'autres *Nœuds*. Aucune contrainte n'est appliquée aux types de *Références* utilisés ou aux *NodeClasses* des *Nœuds* pouvant être référencés. Cependant, le *ReferenceType* peut définir des restrictions qui excluent son utilisation pour des *DataVariables*. Les *ReferenceTypes* normalisés sont décrits à l'Article 7.

Il est prévu qu'une *DataVariable* soit définie dans le contexte d'un *Objet*. Cependant, des *DataVariables* complexes peuvent présenter d'autres *DataVariables*, et les *ObjectTypes* et *VariableTypes* complexes peuvent également contenir des *DataVariables*. De ce fait, chaque *DataVariable* doit être le *TargetNode* d'au moins une *Référence HasComponent* provenant d'un *Objet*, d'un *ObjectType*, d'une *DataVariable* ou d'un *VariableType*. Les *DataVariables* ne doivent pas être le *TargetNode* d'une quelconque *Référence HasProperty.* Par conséquent, une *Référence HasComponent* pointant vers un *Nœud Variable* l'identifie comme étant une *DataVariable*.

La *Référence HasTypeDefinition* pointe vers le *VariableType* utilisé comme définition de type de la *DataVariable*.

Si la *DataVariable* est utilisée comme *InstanceDeclaration* (voir 4.5), tous les *Nœuds* référencés avec des *Références hiérarchiques* directes doivent avoir des *BrowseNames* uniques dans le contexte de cette *DataVariable*.

5.6.5 NodeClass VariableType

Les *VariableTypes* sont utilisés pour fournir des définitions de type pour les *Variables*. Les *VariableTypes* sont définis en utilisant la *NodeClass VariableType* spécifiée dans le Tableau 14.

Tableau 14 – NodeClass VariableType

Nom	Usage	Type de données	Description
Attributs			
Attributs de la NodeClass Base	M	--	Hérités de la *NodeClass Base*. Voir 5.2.
Value	O	Défini par l'*Attribut DataType*	*Valeur* par défaut des instances de ce type.
DataType	M	NodeId	*NodeId* de la définition de type de données pour les instances de ce type.
ValueRank	M	Int32	Cet *Attribut* indique si l'*Attribut Value* du *VariableType* est une matrice et le nombre de dimensions de cette matrice. Il peut prendre les valeurs suivantes: n > 1: la valeur est une matrice dont le nombre de dimensions est spécifié. OneDimension (1): la valeur est une matrice unidimensionnelle. OneOrMoreDimensions (0): la valeur est une matrice à une ou plusieurs dimensions. Scalar (−1): la valeur n'est pas une matrice. Any (−2): la valeur peut être scalaire ou une matrice avec n'importe quel nombre de dimensions. ScalarOrOneDimension (−3): la valeur peut être scalaire ou une matrice unidimensionnelle. NOTE Tous les DataTypes sont considérés comme des valeurs scalaires, même s'ils ont une sémantique de type matrice comme ByteString et String.
ArrayDimensions	O	UInt32[]	Cet *Attribut* spécifie la longueur de chaque dimension d'une valeur matricielle. L'Attribut spécifie la longueur maximale de chaque dimension prise en charge. Si la longueur maximale n'est pas connue, la valeur est 0. Le nombre d'éléments doit être égal à la valeur de l'*Attribut ValueRank*. Cet *Attribut* doit être nul si le *ValueRank* ≤ 0. Par exemple, si un *VariableType* est défini par la matrice C suivante: Int32 myArray[346]; ainsi, ce *DataType* de *VariableType* pointe vers un Int32, le *ValueRank* du *VariableType* prend la valeur 1 et l'*Attribut ArrayDimensions* indique une matrice dont l'une des entrées a la valeur 346.
IsAbstract	M	Boolean	*Attribut* booléen prenant les valeurs suivantes: TRUE Il s'agit d'un *VariableType* abstrait, c'est-à-dire qu'aucune *Variable* de ce type ne doit exister, mais uniquement ses sous-types. FALSE Il s'agit d'un *VariableType* non abstrait, c'est-à-dire qu'il peut exister des *Variables* de ce type.
Références			
HasProperty	0..*		Les *Références HasProperty* sont utilisées pour identifier les *Propriétés* d'un *VariableType*. Les *Nœuds* référencés peuvent être instanciés par des instances de ce type, en fonction des *ModellingRules* définies en 6.4.4.
HasComponent	0..*		Les *Références HasComponent* sont utilisées pour des *VariableTypes* complexes afin d'identifier les DataVariables qui les composent. Les *VariableTypes* complexes peuvent uniquement être utilisés pour des *DataVariables*. Les *Nœuds* référencés peuvent être instanciés par des instances de ce type, en fonction des *ModellingRules* définies en 6.4.4.
HasSubtype	0..*		Les *Références HasSubtype* indiquent les *VariableTypes* qui sont des sous-types de ce type. La *Référence* inverse *SubtypeOf* identifie le type parent de ce type.

Nom	Usage	Type de données	Description
GeneratesEvent	0..*		Les *Références GeneratesEvent* identifient le type d'instance d'*Evénements* que ce type peut générer.
<autres références>	0..*		Les *VariableTypes* peuvent contenir d'autres *Références* qui peuvent être instanciées par des *Variables* définies par ce *VariableType*. Les *ModellingRules* sont définies en 6.4.4.
Propriétés normalisées			
NodeVersion	O	String	La *Propriété NodeVersion* est utilisée pour indiquer la version d'un *Nœud*. La *Propriété NodeVersion* est mise à jour à chaque fois qu'une *Référence* est ajoutée au ou retirée du *Nœud* auquel la *Propriété* appartient. Sauf pour ce qui concerne l'*Attribut DataType*, les modifications de la valeur de l'*Attribut* n'entraînent pas la modification de la *NodeVersion*. Les *Clients* peuvent lire la *Propriété NodeVersion* ou s'y abonner pour déterminer le moment où la structure d'un *Nœud* a varié. Bien que la relation entre un *VariableType* et son *DataType* ne soit pas modélisée en utilisant des *Références*, les modifications apportées à l'*Attribut DataType* d'un *VariableType* entraînent une mise à jour de la *Propriété NodeVersion*.

La *NodeClass VariableType* hérite des *Attributs* de la *NodeClass Base* définie en 5.2. La *NodeClass VariableType* définit également un ensemble d'*Attributs* qui décrit la valeur par défaut ou la valeur initiale de son instance *Variables*. L'*Attribut Value* représente la valeur par défaut. Les *Attributs DataType*, *ValueRank* et *ArrayDimensions* permettent de décrire des valeurs simples et complexes. L'*Attribut IsAbstract* indique si le type peut être directement instancié.

La *NodeClass VariableType* utilise les *Références HasProperty* pour définir les *Propriétés* et les *Références HasComponent* pour définir les *DataVariables*. Leur éventuelle instanciation dépend des *ModellingRules* définies en 6.4.4.

La *Propriété NodeVersion* indique la version du *VariableType*. Il n'existe pas d'autres *Propriétés* définies pour les *VariableTypes* dans le présent document. D'autres parties de l'IEC 62541 peuvent définir des *Propriétés* supplémentaires pour les *VariableTypes*. L'IEC 62541-8 définit un ensemble de *Propriétés* qui peut être utilisé pour les *VariableTypes*.

Les *Références HasSubtype* sont utilisées pour définir des sous-types des *VariableTypes*. Les sous-types des *VariableTypes* héritent des caractéristiques sémantiques générales du type parent. Les règles générales de sous-typage sont définies à l'Article 6. Il n'est pas exigé de fournir la *Référence HasSubtype* pour le super-type, mais ce sous-type doit fournir la *Référence* inverse à son super-type.

Les *Références GeneratesEvent* indiquent que les *Variables* du *VariableType* peuvent être la source d'un *Événement* de l'*EventType* spécifié ou d'un de ses sous-types. Il convient que les *Serveurs* fassent en sorte que les *Références GeneratesEvent* soient bidirectionnelles. Il est cependant admis qu'elles soient unidirectionnelles lorsque le *Serveur* n'est pas capable de présenter le sens inverse pointant de l'*EventType* vers chaque *VariableType* prenant en charge l'*EventType.*

Les *Références GeneratesEvent* sont facultatives, c'est-à-dire que les *Variables* peuvent générer les *Evénements* d'un *EventType* qui n'est pas présenté par ce *VariableType*.

Les *VariableTypes* peuvent utiliser des *Références* supplémentaires pour définir des relations avec d'autres *Nœuds*. Aucune contrainte n'est appliquée aux types de *Références* utilisés ou aux *NodeClasses* des *Nœuds* pouvant être référencés. Cependant, le *ReferenceType* peut

définir des restrictions qui excluent son utilisation pour des *VariableTypes*. Les *ReferenceTypes* normalisés sont décrits à l'Article 7.

Tous les *Nœuds* référencés avec des *Références hiérarchiques* directes doivent avoir des *BrowseNames* uniques dans le contexte du *VariableType* (voir 4.5).

5.6.6 Création du côté client de Variables VariableType

Les *Variables* sont toujours fondées sur un *VariableType*; ceci signifie qu'elles ont une *Référence HasTypeDefinition* pointant vers son *VariableType*.

Les *Clients* peuvent créer des *Variables* en utilisant le *Service AddNodes* défini dans l'IEC 62541-4. Le *Service* exige que le *TypeDefinitionNode* de la *Variable* soit spécifié. Une *Variable* créée par le *Service AddNodes* contient tous les composants définis par son *VariableType* en fonction des *ModellingRules* spécifiées pour les composants. Le *Serveur* peut cependant ajouter des composants supplémentaires et des *Références* à la *Variable* et à ses composants qui ne sont pas définis par le *VariableType*. Ce comportement dépend du *Serveur*. Le *VariableType* spécifie uniquement l'ensemble minimal de composants qui doit exister pour chaque *Variable* d'un *VariableType* donné.

5.7 NodeClass Méthode

Les *Méthodes* définissent des fonctions invocables. Les *Méthodes* sont invoquées par le *Service Call* défini dans l'IEC 62541-4. Les invocations de Méthodes ne sont pas représentées dans l'*AddressSpace*. Les invocations de Méthodes vont toujours jusqu'à achèvement et renvoient toujours des réponses après leur achèvement. Les *Méthodes* sont définies en utilisant la *NodeClass Méthode* spécifiée dans le Tableau 15.

Tableau 15 – NodeClass Méthode

Nom	Usage	Type de données	Description
Attributs			
Attributs de la NodeClass Base	M	--	Hérités de la *NodeClass Base*. Voir 5.2.
Executable	M	Boolean	*L'Attribut Executable* indique si la *Méthode* est actuellement exécutable (False signifie non exécutable; True signifie exécutable). *L'Attribut Executable* ne tient compte d'aucun droit d'accès utilisateur, ce qui signifie que même si la *Méthode* est exécutable, cette possibilité peut être limitée à certains utilisateurs/groupes d'utilisateurs.
UserExecutable	M	Boolean	L'*Attribut UserExecutable* indique si la *Méthode* est actuellement exécutable en tenant compte des droits d'accès de l'utilisateur (False signifie non exécutable; True signifie exécutable).
Références			
HasProperty	0..*		Les *Références "HasProperty"* identifient les *Propriétés* de la *Méthode*.
HasModellingRule	0..1		Les *Méthodes* peuvent pointer vers, au plus, un *Objet ModellingRule* en utilisant une *Référence HasModellingRule* (pour plus d'informations concernant les *ModellingRules*, voir 6.4.4).
GeneratesEvent	0..*		Les *Références GeneratesEvent* identifient le type d'*Evénements* qui sont générés à chaque fois que la *Méthode* est invoquée.
AlwaysGeneratesEvent	0..*		Les *Références AlwaysGeneratesEvent* identifient le type d'*Evénements* qui doit être généré chaque fois que la *Méthode* est invoquée.
<autres références>	0..*		Les *Méthodes* peuvent contenir d'autres *Références*.
Propriétés normalisées			
NodeVersion	O	String	La *Propriété NodeVersion* est utilisée pour indiquer la version d'un *Nœud*. La *Propriété NodeVersion* est mise à jour à chaque fois qu'une *Référence* est ajoutée au ou retirée du *Nœud* auquel la *Propriété* appartient. Les modifications de la valeur de l'*Attribut* n'entraînent pas une modification de la *NodeVersion*. Les *Clients* peuvent lire la *Propriété NodeVersion* ou s'y abonner pour déterminer le moment où la structure d'un *Nœud* a varié.
InputArguments	O	Argument[]	La *Propriété InputArguments* est utilisée pour spécifier les arguments qui doivent être utilisés par un *Client* lorsqu'il invoque la *Méthode*.
OutputArguments	O	Argument[]	La *Propriété OutputArguments* spécifie le résultat renvoyé par l'invocation de la *Méthode*.

La *NodeClass Méthode* hérite des *Attributs* de base de la *NodeClass Base* définie en 5.2. La *NodeClass Méthode* ne définit pas d'*Attributs* supplémentaires.

L'*Attribut Executable* indique si une *Méthode* est exécutable, sans tenir compte des droits d'accès de l'utilisateur. Si le *Serveur* OPC UA ne peut obtenir du système sous-jacent les informations de caractère *Exécutable*, il convient de déclarer qu'il est exécutable. Si une *Méthode* est invoquée, il convient alors que le *Serveur* transfère cette requête et renvoie le *StatusCode* correspondant même si une telle requête est rejetée. Les *StatusCodes* sont définis dans l'IEC 62541-4.

L'*Attribut UserExecutable* indique si la *Méthode* est exécutable, en tenant compte des droits d'accès de l'utilisateur. Si le *Serveur* OPC UA ne peut pas obtenir du système sous-jacent les informations concernant les droits d'accès de l'utilisateur, il convient d'utiliser la même valeur que celle de l'*Attribut Executable*. L'Attribut *UserExecutable* peut être configuré sur False, même si l'Attribut *Executable* est configuré sur True; il doit cependant être configuré sur False si l'Attribut *Executable* est configuré sur False. Les *Clients* ne peuvent pas prendre pour hypothèse qu'une *Méthode* peut être exécutée sur la base de l'*Attribut UserExecutable*. Le *Serveur* peut renvoyer une erreur d'accès refusé en raison d'une modification spécifique à un *Serveur* qui n'a pas été reflétée dans l'état de cet *Attribut* au moment où le *Client* y a accédé.

Les *Propriétés* peuvent être définies pour les *Méthodes* qui utilisent les *Références HasProperty*. Les *Propriétés InputArguments* et *OutputArguments* spécifient les arguments d'entrée et les arguments de sortie de la *Méthode*. Les deux contiennent une matrice de l'*Argument DataType* comme spécifié en 8.6. Une matrice vide ou une *Propriété* qui n'est pas fournie indique qu'il n'existe pas d'arguments d'entrée ou d'arguments de sortie pour la *Méthode*.

La *Propriété NodeVersion* indique la version de la *Méthode*. Il n'existe pas d'autres *Propriétés* définies pour les *Méthodes* dans le présent document. D'autres parties de l'IEC 62541 peuvent définir des *Propriétés* supplémentaires pour les *Méthodes*.

Pour spécifier sa *ModellingRule*, une *Méthode* peut utiliser au plus une *Référence HasModellingRule* pointant vers un *Objet ModellingRule*. Les *ModellingRules* sont définies en 6.4.4.

Les *Références GeneratesEvent* indiquent que les *Méthodes* peuvent générer un *Événement* de l'*EventType* spécifié ou d'un de ses sous-types pour chaque invocation de la *Méthode*. Un *Serveur* peut générer un *Événement* pour chaque *EventType* référencé lorsqu'une *Méthode* est invoquée avec succès.

Les *Références AlwaysGeneratesEvent* indiquent que les *Méthodes* génèrent un *Événement* de l'*EventType* spécifié ou de l'un de ses sous-types pour chaque invocation de la *Méthode*. Un *Serveur* doit toujours générer un *Événement* pour chaque *EventType* référencé lorsqu'une *Méthode* est invoquée avec succès.

Il convient que les *Serveurs* fassent en sorte que les *Références GeneratesEvent* soient bidirectionnelles. Il est cependant admis qu'elles soient unidirectionnelles lorsque le *Serveur* n'est pas capable de présenter le sens inverse, pointant de l'*EventType* vers chaque *Méthode* générant l'*EventType*.

Les *Références GeneratesEvent* sont facultatives, ce qui signifie que l'invocation d'une *Méthode* peut produire des *Evénements* d'un *EventType* qui n'est pas référencé avec une *Référence GeneratesEvent* par la *Méthode*.

Les *Méthodes* peuvent utiliser des *Références* supplémentaires pour définir des relations avec d'autres *Nœuds*. Aucune contrainte n'est appliquée aux types de *Références* utilisés ou aux *NodeClasses* des *Nœuds* pouvant être référencés. Cependant, le *ReferenceType* peut définir des restrictions qui excluent son utilisation pour des *Méthodes*. Les *ReferenceTypes* normalisés sont décrits à l'Article 7.

Une *Méthode* doit toujours être le *TargetNode* d'au moins une *Référence HasComponent*. Le *SourceNode* de ces *Références HasComponent* doit être un *Objet* ou un *ObjectType*. Si une *Méthode* est invoquée, alors le *NodeId* de l'un de ces *Nœuds* doit être placé dans le *Service* Call défini dans l'IEC 62541-4, en tant que paramètre, afin de détecter le contexte d'exécution de la *Méthode*.

Si la *Méthode* est utilisée comme *InstanceDeclaration* (voir 4.5), tous les *Nœuds* référencés avec des *Références hiérarchiques* directes doivent avoir des *BrowseNames* uniques dans le contexte de cette *Méthode*.

5.8 DataTypes

5.8.1 Modèle de DataType

Le Modèle de DataType est utilisé pour définir des types de données simples et structurés. Les types de données sont utilisés pour décrire la structure de l'*Attribut Value* des *Variables* et leurs *VariableTypes*. Par conséquent, chaque *Variable* et *VariableType* pointe, au moyen de son *Attribut DataType*, vers un *Nœud* de la *NodeClass DataType,* comme représenté à la Figure 10.

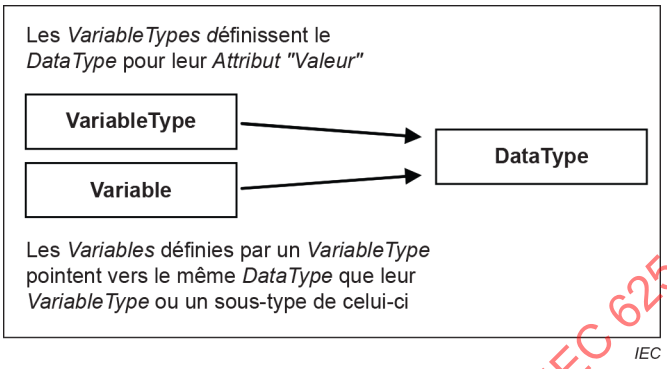


Figure 10 – Variables, VariableTypes et DataTypes correspondants

Dans de nombreuses situations, le *NodeId* du *Nœud DataType* (*DataTypeId*) est bien connu des *Clients* et des *Serveurs*. L'Article 8 définit les *DataTypes* et l'IEC 62541-6 définit leurs *DataTypeIds*. Par ailleurs, d'autres organismes peuvent définir des *DataTypes* notoires dans l'industrie. Les *DataTypeIds* notoires permettent une généralisation sur les *Serveurs* OPC UA et les *Clients* peuvent de ce fait interpréter des valeurs sans avoir à lire la description de type à partir du *Serveur*. Par conséquent, les *Serveurs* peuvent utiliser des *DataTypeIds* notoires sans avoir à représenter les *Nœuds DataTypes* correspondants dans leurs *AddressSpaces*.

Dans d'autres situations, les *DataTypes* et leurs *DataTypeIds* correspondants peuvent être définis par le fournisseur. Il convient que les *Serveurs* tentent de présenter les *Nœuds DataType* et les informations concernant la structure de ces *DataTypes* pour que les *Clients* puissent les lire, même si ces informations peuvent ne pas toujours être disponibles pour le *Serveur*.

La Figure 11 représente les *Nœuds* utilisés dans l'*AddressSpace* pour décrire la structure d'un *DataType*. Le *DataType* pointe vers un *Objet* du type *DataTypeEncodingType*. Chaque *DataType* peut avoir plusieurs *DataTypeEncoding*, par exemple encodage "Default", "UA Binary" et "XML". Les services définis dans l'IEC 62541-4 autorisent que les *Clients* demandent un encodage ou choisissent un encodage "Default". Chaque *DataTypeEncoding* est utilisé précisément par un *DataType*, ce qui signifie qu'il n'est pas admis que deux *DataTypes* pointent vers le même *DataTypeEncoding*.

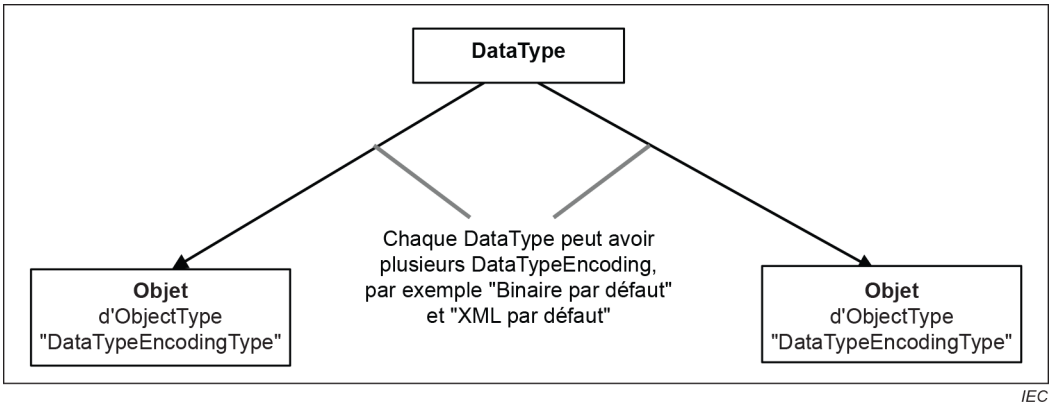


Figure 11 – Modèle de DataType

Sachant que le *NodeId* du *DataTypeEncoding* est utilisé dans certains mappings pour identifier le *DataType* et son encodage (comme défini dans l'IEC 62541-6), ces *NodeId* peuvent également être notoires lorsqu'ils correspondent à des *DataTypeId* notoires.

5.8.2 Règles d'encodage pour différents types de DataTypes

Les différents types de *DataTypes* sont traités de manière différente en ce qui concerne l'encodage et selon que cet encodage est ou non représenté dans l'*AddressSpace*.

Les *DataTypes Built-in* sont un jeu fixe de *DataTypes* (voir IEC 62541-6 pour une liste complète des *DataTypes Built-in*). Ces types n'ont pas d'encodages visibles dans l'*AddressSpace*, car il convient que l'encodage soit connu de l'ensemble des produits OPC UA. Des exemples de *DataTypes Built-in* sont *Int32* (voir 8.26) et *Double* (voir 8.12).

Les *DataTypes Simples* sont des sous-types de *DataTypes Built-in.* Ils sont traités à la volée comme les *DataTypes Built-in*, ce qui signifie qu'ils ne peuvent pas être distingués de leurs super-types *Built-in* en cours d'utilisation. Etant traités, en ce qui concerne l'encodage, comme des *DataTypes Built-in*, leurs encodages ne peuvent pas être définis dans l'*AddressSpace*. Les *Clients* peuvent lire l'*Attribut DataType* d'une *Variable* ou d'un *VariableType* pour identifier le *DataType simple* de l'*Attribut Value*. Un exemple de *DataType simple* est la *Duration*. Il est traité à la volée comme un *Double*, mais le *Client* peut lire l'*Attribut DataType* et ainsi interpréter la valeur, comme définie par la *Duration* (voir 8.13).

Les *DataTypes Structured* sont des *DataTypes* qui représentent des données structurées et ne sont pas définis comme des *DataTypes Built-in. Les DataTypes Structured* héritent directement ou indirectement de la *Structure* de *DataType* définie en 8.33. Les *DataTypes Structured* peuvent avoir plusieurs encodages et ces derniers sont présentés dans l'*AddressSpace*. La manière dont l'encodage des *DataTypes Structured* est traité à la volée est définie dans l'IEC 62541-6. L'encodage du *DataType structuré* est transmis avec chaque valeur; ainsi, les *Clients* sont informés du *DataType* sans avoir à lire l'*Attribut DataType*. L'encodage doit être transmis, de façon à ce que le *Client* soit en mesure d'interpréter les données. Un exemple de *DataType structuré* est l'*Argument* (voir 8.6).

Les *DataTypes Enumeration* sont des *DataTypes* qui représentent des jeux discrets de valeurs définies. Les énumérations sont toujours codées en Int32 à la volée, comme défini dans l'IEC 62541-6. *Les DataTypes* d'énumération héritent directement ou indirectement du *DataType Enumeration* défini en 8.14. Les énumérations n'ont pas d'encodages présentés dans l'*AddressSpace*. Pour exposer la représentation lisible par l'homme d'une valeur énumérée, le *Nœud DataType* peut disposer d'une *Propriété EnumStrings* contenant une matrice de *LocalizedText*. La représentation Integer de la valeur d'énumération pointe vers une position à l'intérieur de cette matrice. La *Propriété EnumValues* peut être utilisée au lieu des *EnumStrings* pour prendre en charge la représentation Integer des énumérations qui ne sont pas fondées sur zéro ou qui présentent des espaces. Elle contient une matrice d'un

DataType Structured contenant la représentation Integer ainsi que la représentation lisible par l'homme. La *NodeClass* qui est définie en 8.30 est un exemple de *DataType* d'énumération contenant une liste d'Integers éparse.

Outre les *DataTypes* décrits ci-dessus, les *DataTypes* abstraits sont également pris en charge; ces derniers n'ont aucun encodage et ne peuvent pas être échangés à la volée. Les *Variables* et *VariableTypes* utilisent des *DataTypes* abstraits pour indiquer que leur *Valeur* peut être n'importe lequel des sous-types du *DataType* abstrait. Un exemple de *DataType* abstrait est l'Integer défini en 8.24.

5.8.3 NodeClass DataType

La *NodeClass Datatype* décrit la syntaxe d'une *Variable Value*. Les *DataTypes* sont définis en utilisant la *NodeClass Datatype*, spécifiée dans le Tableau 16.

Tableau 16 – NodeClass DataType

Nom	Usage	Type de données	Description
Attributs			
Attributs de la NodeClass Base	M	--	Hérités de la *NodeClass Base*. Voir 5.2.
IsAbstract	M	Boolean	*Attribut* booléen prenant les valeurs suivantes: TRUE Il s'agit d'un *DataType* abstrait. FALSE Il s'agit d'un *DataType* non abstrait.
DataTypeDefinition	O	DataTypeDefinition	L'*Attribut DataTypeDefinition* est utilisé pour fournir les métadonnées et les informations d'encodage pour les *DataTypes* personnalisés. Le *DataType* abstrait de la *DataTypeDefinition* est défini en 8.48. DataTypes Structure et Union L'*Attribut* est obligatoire pour les *DataTypes* issus de la *Structure* et de l'*Union*. Pour ces *DataTypes*, l'*Attribut* contient une structure du *DataType StructureDefinition*. Le *DataType StructureDefinition* est défini en 8.49. Il s'agit d'un sous-type de la *DataTypeDefinition*. DataTypes Enumeration et OptionSet L'*Attribut* est obligatoire pour les *DataTypes* issus de l'*Enumeration et de l'OptionSet* et des sous-types de l'*UInteger* représentant un *OptionSet*. Pour ces *DataTypes*, l'*Attribut* contient une structure du *DataType EnumDefinition*. Le *DataType EnumDefinition* est défini en 8.50. Il s'agit d'un sous-type de la *DataTypeDefinition*.
Références			
HasProperty	0..*		Les *Références HasProperty* identifient les *Propriétés* du *DataType*.
HasSubtype	0..*		Les *Références HasSubtype* peuvent être utilisées pour couvrir une hiérarchie de type de données.
HasEncoding	0..*		Les *Références HasEncoding* indiquent les encodages du *DataType* représentés comme des *Objets* de type *DataTypeEncodingType*. Seuls des *DataTypes Structured* concrets peuvent utiliser les *Références HasEncoding*. Les *DataTypes* abstraits *Built-in, Enumeration* et *Simples* ne peuvent pas être le *SourceNode* d'une *Référence HasEncoding*. Chaque *DataType Structured* concret doit pointer vers au moins un *Objet DataTypeEncoding,* avec le *BrowseName* "Default Binary" ou "Default XML" dont le *NamespaceIndex* est 0. Le *BrowseName* des *Objets DataTypeEncoding* doit être unique dans le contexte d'un *DataType*, ce qui signifie qu'un *DataType* ne doit pas pointer vers deux *DataTypeEncodings* dont le *BrowseName* est identique.
Propriétés normalisées			
NodeVersion	O	String	La *Propriété NodeVersion* est utilisée pour indiquer la version d'un *Nœud*. La *Propriété NodeVersion* est mise à jour à chaque fois qu'une *Référence* est ajoutée au ou retirée du *Nœud* auquel la *Propriété* appartient. Les modifications de la valeur de l'*Attribut* n'entraînent pas une modification de la *NodeVersion*. Les *Clients* peuvent lire la *Propriété NodeVersion* ou s'y abonner pour déterminer le moment où la structure d'un *Nœud* a varié. Les *Clients* ne doivent pas utiliser le contenu à des fins programmatiques, sauf pour les comparaisons d'égalités.

Nom	Usage	Type de données	Description
EnumStrings	O	LocalizedText[]	La *Propriété EnumStrings* s'applique uniquement aux *DataTypes Enumeration.* Elle ne doit pas être appliquée à d'autres *DataTypes.* Si la *Propriété EnumValues* est fournie, la *Propriété EnumStrings* ne doit pas être fournie. Chaque entrée de la matrice de *LocalizedText* dans cette *Propriété* constitue la représentation lisible par l'homme d'une valeur énumérée. La représentation Integer de la valeur d'énumération pointe vers une position de la matrice.
EnumValues	O	EnumValueType[]	La *Propriété EnumValues* s'applique uniquement aux *DataTypes Enumeration.* Elle ne doit pas être appliquée à d'autres *DataTypes.* Si la *Propriété EnumStrings* est fournie, la *Propriété EnumValues* ne doit pas être fournie. A l'aide de la *Propriété EnumValues*, il est possible de représenter des énumérations avec des entiers qui ne sont pas fondées sur zéro ou qui ont des espaces (par exemple, 1, 2, 4, 8 et 16). Chaque entrée de la matrice de *EnumValueType* dans cette *Propriété* représente une valeur d'énumération avec sa notation d'entier, une représentation lisible par l'homme et des informations d'aide.
OptionSetValues	O	LocalizedText[]	La *Propriété OptionSetValues* s'applique uniquement aux *DataTypes OptionSet* et aux *DataTypes UInteger*. Un *DataType OptionSet* est utilisé pour représenter un masque de bits et la *Propriété OptionSetValues* contient la représentation lisible par l'homme pour chaque bit du masque de bits. La *Propriété OptionSetValues* fournit une matrice du *LocalizedText* qui contient la représentation lisible par l'homme pour chaque bit.

La *NodeClass DataType* hérite des *Attributs* de base de la *NodeClass Base* définie en 5.2. L'*Attribut IsAbstract* spécifie si le *DataType* est ou non abstrait. Des *DataTypes* abstraits peuvent être utilisés dans l'*AddressSpace*, c'est-à-dire que les *Variables* et les *VariableTypes* peuvent pointer, au moyen de leur *Attribut DataType*, vers un *DataType* abstrait. Cependant, les valeurs concrètes ne peuvent jamais être un *DataType* abstrait et doivent toujours être d'un sous-type concret du *DataType* abstrait.

Les *Références HasProperty* sont utilisées pour identifier les *Propriétés* d'un *DataType*. La *Propriété NodeVersion* est utilisée pour indiquer la version du *DataType*. La *Propriété EnumStrings* contient des représentations interprétables par l'utilisateur des valeurs d'énumération et est uniquement appliquée aux *DataTypes Enumeration.* Au lieu de la *Propriété EnumStrings*, un *DataType Enumeration* peut aussi utiliser la *Propriété EnumValues* pour représenter des *énumérations* avec des valeurs d'entier qui ne sont pas fondées sur zéro ou contenant des espaces. Il n'existe pas d'autres *Propriétés* définies pour les *DataTypes* dans le présent document. D'autres parties de l'IEC 62541 peuvent définir des *Propriétés* supplémentaires pour les *DataTypes*.

Les *Références HasSubtype* peuvent être utilisées pour présenter une hiérarchie de type de données dans l'*AddressSpace*. La sémantique du sous-typage est uniquement définie au point qu'un Serveur peut fournir des instances du sous-type plutôt que du DataType. Il convient que les *Clients* ne fassent aucune hypothèse relative à une éventuelle sémantique utilisant ces informations. Par exemple, il peut s'avérer impossible de relier la valeur d'un type de données à son type de données de base. Les *Serveurs* peuvent ne pas fournir les *références HasSubtype*, même si leurs DataTypes couvrent une hiérarchie de types. Certaines restrictions s'appliquent pour le sous-typage des DataTypes d'énumération définis en 8.14.

Les *Références HasEncoding* pointent du *DataType* vers ses *DataTypeEncodings*. Chaque *DataType Structured* concret peut pointer vers plusieurs *DataTypeEncodings*, mais chaque *DataTypeEncoding* doit appartenir à un *DataType*; en d'autres termes, il n'est pas admis que

deux *Nœuds DataType* pointent vers le même *Objet DataTypeEncoding* au moyen de *Références HasEncoding*.

Un *DataType* abstrait n'est pas le *SourceNode* d'une *Référence HasEncoding*. Le *DataTypeEncoding* d'un *DataType* abstrait est fourni par ses sous-types concrets.

Les *Nœuds DataType* ne doivent pas être le *SourceNode* d'autres types de *Références*. Ils peuvent cependant être le *TargetNode* d'autres *Références*.

5.8.4 DataTypeEncoding et information sur l'encodage

Si un *Nœud DataType* est présenté dans l'*AddressSpace*, il doit fournir ses *DataTypeEncodings* en utilisant les *Références HasEncoding*. Ces *Références* doivent être bidirectionnelles. La Figure 12 fournit un exemple de modélisation de *DataTypes* dans l'*AddressSpace*.

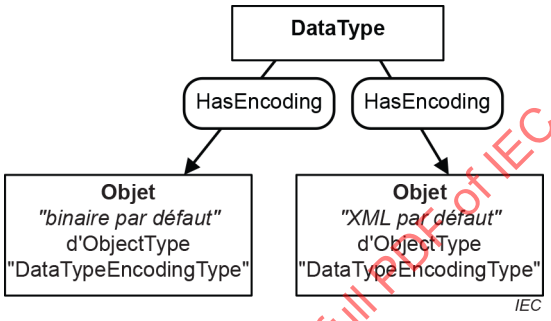


Figure 12 – Exemple de modélisation de DataType

Les informations relatives à la manière d'encoder le *DataType* sont fournies dans l'*Attribut DataTypeDefinition* du *Nœud DataType*. Le contenu de cet *Attribut* ne doit pas être modifié une fois qu'il a été fourni aux *Clients* étant donné que les *Clients* peuvent constamment mettre ces informations en cache. S'il est nécessaire d'apporter des modifications conceptuelles à l'encodage d'un *DataType*, il est nécessaire de fournir un nouveau *DataType*. Autrement dit, un nouveau *NodeId* doit être utilisé pour le *DataType*. Etant donné que les *Clients* identifient le *DataType* via les *DataTypeEncodings*, les *NodeId* pour les *DataTypeEncodings* du *DataType* doivent également être modifiés lorsque l'encodage est modifié.

5.9 Récapitulatif des Attributs des NodeClasses

Le Tableau 17 répertorie tous les *Attributs* définis dans le présent document et précise les *NodeClasses* qui les utilisent, que ce soit de manière facultative (O pour Optional) ou obligatoire (M pour Mandatory).

Tableau 17 – Vue d'ensemble des Attributs

Attribut	Variable	Type de variable	Objet	Type d'objet	Type de référence	DataType	Method	View
AccessLevel	M							
AccessLevelEx	O							
ArrayDimensions	O	O						
AccessRestrictions	O	O	O	O	O	O	O	O
BrowseName	M	M	M	M	M	M	M	M
ContainsNoLoops								M
DataType	M	M						
DataTypeDefinition						O		
Description	O	O	O	O	O	O	O	O
DisplayName	M	M	M	M	M	M	M	M
EventNotifier			M					M
Executable							M	
Historizing	M							
InverseName					O			
IsAbstract		M		M	M	M		
MinimumSamplingInterval	O							
NodeClass	M	M	M	M	M	M	M	M
NodeId	M	M	M	M	M	M	M	M
RolePermissions	O	O	O	O	O	O	O	O
Symmetric					M			
UserAccessLevel	M							
UserExecutable							M	
UserRolePermissions	O	O	O	O	O	O	O	O
UserWriteMask	O	O	O	O	O	O	O	O
Value	M	O						
ValueRank	M	M						
WriteMask	O	O	O	O	O	O	O	O

6 Modèle type d'ObjectTypes et de VariableTypes

6.1 Vue d'ensemble

Dans la suite de l'Article 6, le modèle type d'*ObjectTypes* et de *VariableTypes* est défini en ce qui concerne le sous-typage et l'instanciation.

6.2 Définitions

6.2.1 InstanceDeclaration

Une *InstanceDeclaration* est un *Objet*, une *Variable* ou une *Méthode* qui référence une *ModellingRule* au moyen d'une *Référence HasModellingRule* et qui est le *TargetNode* d'une *Référence hiérarchique* à partir d'une *TypeDefinitionNode* ou d'une autre *InstanceDeclaration*. Le type d'une *InstanceDeclaration* peut être abstrait; cependant, l'instance doit être de type concret.

6.2.2 Instances sans ModellingRules

S'il n'existe aucune *ModellingRule*, le *Nœud* n'est pas pris en compte pour l'instanciation de type ni pour le sous-typage.

Si un *Nœud* référencé par un *TypeDefinitionNode* ne référence pas une *ModellingRule*, cela signifie que ce *Nœud* appartient uniquement au *TypeDefinitionNode* et non aux instances. Par exemple, un *Nœud ObjectType* peut contenir une *Propriété* qui décrit des scénarios qui peuvent utiliser le type. Cette *Propriété* n'est pas prise en compte lors de la création d'instances de ce type. Ceci est également vrai pour le sous-typage, ce qui signifie que les sous-types de la définition de type n'héritent pas du *Nœud* référencé.

6.2.3 InstanceDeclarationHierarchy

L'*InstanceDeclarationHierarchy* d'un *TypeDefinitionNode* contient le *TypeDefinitionNode* ainsi que toutes les *InstanceDeclarations* directement ou indirectement référencées à partir du *TypeDefinitionNode* au moyen de *Références hiérarchiques* directes.

6.2.4 Nœud similaire à l'InstanceDeclaration

Un *Nœud* similaire à une *InstanceDeclaration* est un *Nœud* qui a le même *BrowseName* et la même *NodeClass* que l'*InstanceDeclaration* et, dans le cas de *Variables* et d'*Objets*, le même *TypeDefinitionNode* ou un sous-type de celui-ci.

6.2.5 BrowsePath

Toutes les cibles de *Références hiérarchiques* directes à partir d'un *TypeDefinitionNode* doivent avoir un *BrowseName* qui est unique au sein du *TypeDefinitionNode*. Cette même restriction s'applique aux cibles de *Références hiérarchiques* directes à partir d'une quelconque *InstanceDeclaration*. Ceci signifie que toute *InstanceDeclaration* dans l'*InstanceDeclarationHierarchy* peut être identifiée de manière unique par une séquence de *BrowseNames*. Cette séquence de *BrowseNames* est appelée *BrowsePath*.

6.2.6 Traitement des Attributs des InstanceDeclarations

Il existe certaines restrictions concernant les *Attributs* des *InstanceDeclarations* lorsque l'*InstanceDeclaration* est remplacée ou instanciée. Le *BrowseName* et la *NodeClass* ne doivent jamais varier et toujours demeurer les mêmes que ceux de l'*InstanceDeclaration* initiale.

En outre, les règles définies en 6.2.7 s'appliquent aux *InstanceDeclarations* de la *NodeClass Variables*.

6.2.7 Traitement des Attributs de Variables et de VariableTypes

Il existe certaines restrictions concernant les *Attributs* d'un *VariableType* ou d'une *Variable* utilisée comme *InstanceDeclaration* pour ce qui concerne le type de données de l'*Attribut Value*.

Lorsqu'une *Variable* utilisée comme *InstanceDeclaration* ou qu'un *VariableType* est remplacé ou instancié, les règles suivantes s'appliquent:

a) L'*Attribut DataType* ne peut être modifié qu'en un nouveau *DataType* si le nouveau *DataType* est un sous-type du *DataType* initialement utilisé.

b) L'*Attribut ValueRank* peut seulement être plus strict:

   1) "Any": peut être configuré sur toute autre valeur;

   2) "ScalarOrOneDimension": peut être configuré sur "Scalar" ou sur "OneDimension"';

3) "OneOrMoreDimensions": peut être configuré sur un nombre concret de dimensions (valeur > 0);
4) Toutes les autres valeurs de cet *Attribut* doivent demeurer inchangées.

c) L'*Attribut ArrayDimensions* peut être ajouté s'il n'a pas été fourni lors de la modification de la valeur 0 d'une entrée de la matrice en une valeur différente. Toutes les autres valeurs de la matrice doivent demeurer inchangées.

6.2.8 NodeIds des InstanceDeclarations

Les *InstanceDeclarations* sont identifiées par leur *BrowsePath*. Différents *Serveurs* peuvent utiliser différents *NodeIds* pour les *InstanceDeclarations* des *TypeDefinitionNodes* communs, à moins que le *TypeDefinitionNode* définisse déjà un *NodeId* pour l'*InstanceDeclaration*. Tous les *TypeDefinitionNodes* définis dans l'IEC 62541-5 définissent déjà les *NodeIds* pour leurs *InstanceDeclarations* et doivent donc être utilisés dans tous les *Serveurs.*

6.3 Sous-typage d'ObjectTypes et de VariableTypes

6.3.1 Vue d'ensemble

Le *ReferenceType HasSubtype* définit des sous-types de types. Le sous-typage ne peut avoir lieu qu'entre *Nœuds* de la même *NodeClass.* Les règles de sous-typage des *ReferenceTypes* sont décrites en 5.3.3.3. Il n'y a aucune définition commune pour le sous-typage des *DataTypes*, comme décrit en 5.8.3. La suite du 6.3 spécifie les règles de sous-typage pour un héritage simple portant sur les *ObjectTypes* et les *VariableTypes*.

6.3.2 Attributs

Les sous-types héritent des valeurs d'*Attribut* du type parent, sauf le *NodeId.* Les valeurs d'*Attribut* héritées peuvent être remplacées par le sous-type; il convient de remplacer les valeurs du *BrowseName* et du *DisplayName*. Des règles particulières s'appliquent à certains *Attributs* de *VariableTypes* comme défini en 6.2.7. Des *Attributs* facultatifs, non prévus par le type parent, peuvent être ajoutés au sous-type.

6.3.3 InstanceDeclarations

6.3.3.1 Vue d'ensemble

Les sous-types héritent intégralement des *InstanceDeclarations* de type parent héritées.

Tant que ces *InstanceDeclarations* ne sont pas remplacées, elles ne sont pas référencées par le sous-type. Des *InstanceDeclarations* peuvent être remplacées en ajoutant des *Références*, en modifiant les *Références* portant sur les différents *Nœuds*, en modifiant les *Références* en sous-types du *ReferenceType* initial, en modifiant les valeurs des *Attributs* ou en ajoutant des *Attributs* facultatifs. Pour obtenir l'ensemble des informations concernant un sous-type, les *InstanceDeclarations* héritées doivent être recueillies à partir de tous les types qui peuvent être obtenus en suivant de manière récursive les *Références* inverses *HasSubtype* à partir du sous-type. Ce recueil d'*InstanceDeclarations* est appelé l'*InstanceDeclarationHierarchy* intégralement héritée d'un sous-type.

La suite du 6.3.3 définit la méthode de construction de l'*InstanceDeclarationHierarchy* intégralement héritée et la manière dont les *InstanceDeclarations* peuvent être remplacées.

6.3.3.2 InstanceDeclarationHierarchy intégralement héritée

Une instance de *TypeDefinitionNode* est décrite par l'*InstanceDeclarationHierarchy* intégralement héritée du *TypeDefinitionNode*. L'*InstanceDeclarationHierarchy* intégralement héritée peut être créée en partant de l'*InstanceDeclarationHierarchy* du *TypeDefinitionNode* et en fusionnant l'*InstanceDeclarationHierarchy* intégralement héritée de son type parent.

Le processus de fusion des *InstanceDeclarationHierarchies* est direct. Il peut être représenté par l'exemple de la Figure 13, qui spécifie un *TypeDefinitionNode* "BetaType" étant un sous-type de "AlphaType". Dans chaque case, la lettre est le *BrowseName* et le chiffre est le *NodeId.*

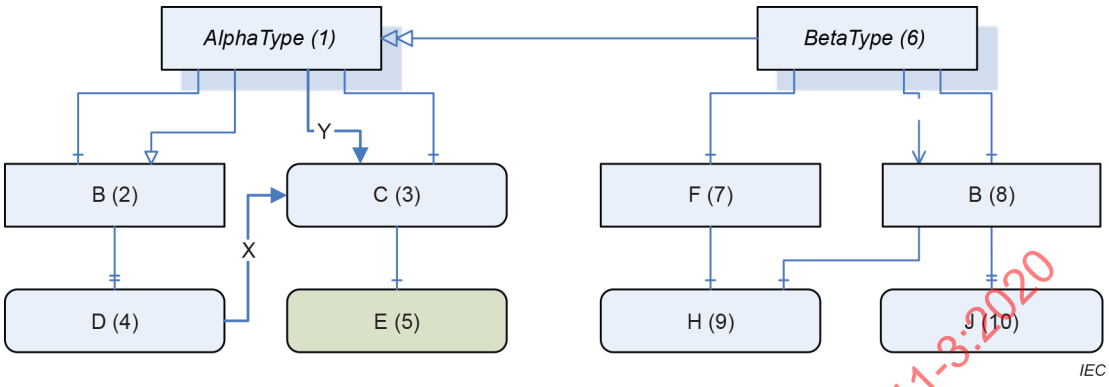


Figure 13 – Sous-typage des TypeDefinitionNodes

Une *InstanceDeclarationHierarchy* peut être pleinement décrite comme un tableau de *Nœuds* identifiés par leurs *BrowsePaths* avec un tableau de *Références* correspondant. L'*InstanceDeclarationHierarchy* pour "BetaType" est décrite dans le Tableau 18; la moitié supérieure de celui-ci correspond au tableau des *Nœuds* et la moitié inférieure au tableau des *Références* (les références *HasModellingRule* ont été omises du tableau pour plus de clarté; tous les Nœuds à l'exception des Nœuds 1, 6 et 5 ont des *ModellingRules*). Toutes les *InstanceDeclarations* de l'*InstanceDeclarationHierarchy* et tous les *Nœuds* référencés avec une *Référence* non hiérarchique à partir d'une telle *InstanceDeclaration* sont ajoutés au tableau. Les *Références hiérarchiques* aux *Nœuds* sans *ModellingRule* ne sont pas prises en compte.

Tableau 18 – InstanceDeclarationHierarchy pour "BetaType"

BrowsePath	NodeId
/	6
/F	7
/B	8
/F/H	9
/B/J	10
/B/H	9

Chemin source	ReferenceType	Chemin cible	NodeId cible
/	HasComponent	/F	-
/	HasComponent	/B	-
/	Z	/B	-
/	HasTypeDefinition	-	BetaType
/F	HasComponent	/F/H	-
/F	HasTypeDefinition	-	BaseObjectType
/B	HasProperty	/B/J	-
/B	HasTypeDefinition	-	BaseObjectType
/F/H	HasTypeDefinition	-	PropertyType
/B/J	HasTypeDefinition	-	PropertyType
/B	HasComponent	/B/H	-
/B/H	HasTypeDefinition	-	BaseDataVariableType

Plusieurs *BrowsePaths* vers le même *Nœud* doivent être traités comme des *Nœuds* séparés. Une *Instance* peut fournir différents *Nœuds* pour chaque *BrowsePath*.

L'*InstanceDeclarationHierarchy* intégralement héritée pour "BetaType" peut à présent être construite par fusion avec l'*InstanceDeclarationHierarchy* pour "AlphaType". Le résultat est présenté dans le Tableau 19; les entrées ajoutées à partir de "AlphaType" sont grisées.

Tableau 19 – InstanceDeclarationHierarchy intégralement héritée pour "BetaType"

BrowsePath	NodeId
/	6
/F	7
/B	8
/F/H	9
/B/J	10
/B/H	9
/B/D	4
/C	3

Chemin source	ReferenceType	Chemin cible	NodeId cible
/	HasComponent	/F	-
/	HasComponent	/B	-
/	Z	/B	-
/	HasTypeDefinition	-	BetaType
/F	HasComponent	/F/H	-
/F	HasTypeDefinition	-	BaseObjectType
/B	HasProperty	/B/J	-
/B	HasTypeDefinition	-	BaseObjectType
/F/H	HasTypeDefinition	-	PropertyType
/B/J	HasTypeDefinition	-	PropertyType
/B	HasComponent	/B/H	-
/B/H	HasTypeDefinition	-	BaseDataVariableType
/	HasNotifier	/B	-
/B	HasProperty	/B/D	-
/	HasComponent	/C	-
/	Y	/C	-
/C	HasTypeDefinition	-	BaseDataVariableType
/B/D	HasTypeDefinition	-	PropertyType
/B/D	X	/C	-

Le *BrowsePath* "/B" existe déjà dans le tableau, de sorte qu'il n'est pas nécessaire de l'ajouter. Cependant, la référence *HasNotifier* de "/" vers "/B" n'existe pas et a été ajoutée.

Les *Nœuds* et *Références* définis dans le Tableau 19 peuvent être utilisés pour générer l'*InstanceDeclarationHierarchy* intégralement héritée représentée à la Figure 14. L'*InstanceDeclarationHierarchy* intégralement héritée contient toutes les informations nécessaires à un *TypeDefinitionNode* pour ce qui concerne sa structure complexe, sans qu'il soit nécessaire d'ajouter d'éventuelles informations supplémentaires à partir de ses super-types.

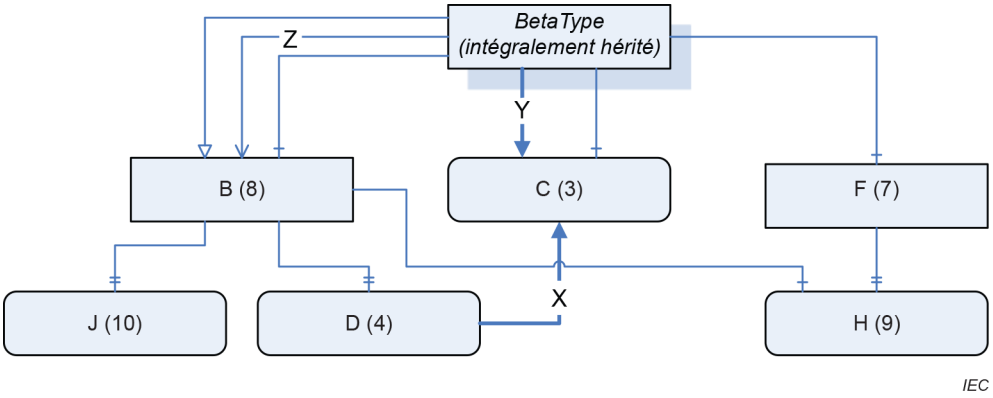


Figure 14 – InstanceDeclarationHierarchy intégralement héritée pour "BetaType"

6.3.3.3 Remplacement des InstanceDeclarations

Un sous-type remplace une *InstanceDeclaration* en spécifiant une *InstanceDeclaration* ayant le même *BrowsePath*. Une *InstanceDeclaration* remplacée doit avoir la même *NodeClass* et le même *BrowseName*. Le *TypeDefinitionNode* de l'*InstanceDeclaration* remplacée doit être le même ou un sous-type du *TypeDefinitionNode* spécifié dans le super-type.

Lorsqu'une *InstanceDeclaration* est remplacée, il est nécessaire de fournir les *Références hiérarchiques* qui relient le nouveau *Nœud* au sous-type (les *Références* sont utilisées pour déterminer le *BrowsePath* du *Nœud*).

Il est uniquement possible de remplacer les *InstanceDeclarations* directement référencées à partir du *TypeDefinitionNode*. Si une *InstanceDeclaration* indirectement référencée, comme "J" dans la Figure 14, doit être remplacée, les *InstanceDeclarations* directement référencées qui incluent "J", en l'occurrence "B", doivent être remplacées en premier lieu; "J" peut alors être remplacé dans une seconde phase.

Une *Référence* est remplacée si elle s'inscrit entre deux *Nœuds* remplacés et qu'elle a le même *ReferenceType* qu'une *Référence* définie dans le super-type. La *Référence* spécifiée dans le sous-type peut être un sous-type du *ReferenceType* utilisé dans le type parent.

Toute *Référence* non hiérarchique spécifiée pour l'*InstanceDeclaration* remplacée est traitée comme une nouvelle *Référence*, à moins que le *ReferenceType* ne permette qu'une seule *Référence* par *SourceNode*. Dans ce cas, le sous-type peut modifier la cible de la *Référence*, mais la nouvelle cible doit avoir la même *NodeClass* et, pour les *Objets* et les *Variables*, avoir également le même type ou un sous-type du type spécifié dans le type parent.

Le *Nœud* remplaçant peut spécifier de nouvelles valeurs pour les *Attributs* de *Nœud* autres que la *NodeClass* ou le *BrowseName*; toutefois les restrictions relatives aux *Attributs* spécifiées en 6.2.6 s'appliquent. Tout *Attribut* fourni par l'*InstanceDeclaration* remplacée doit être fourni par l'*InstanceDeclaration* remplaçante; des *Attributs* facultatifs supplémentaires peuvent être ajoutés.

La *ModellingRule* de l'*InstanceDeclaration* remplaçante peut être modifiée comme défini en 6.4.4.3.

Il est nécessaire que chaque *InstanceDeclaration* remplaçante ait ses propres *Références HasModellingRule* et *HasTypeDefinition*, même si elles n'ont pas été modifiées.

Il convient de ne pas remplacer un *Nœud* par un sous-type, à moins qu'il soit nécessaire de le modifier.

La sémantique de certains *TypeDefinitionNodes* et *ReferenceTypes* peut imposer des restrictions supplémentaires concernant les *Nœuds* remplaçants.

6.4 Instances d'ObjectTypes et de VariableTypes

6.4.1 Vue d'ensemble

Toute *Instance* d'un *TypeDefinitionNode* peut être la racine d'une hiérarchie qui reflète l'*InstanceDeclarationHierarchy* du *TypeDefinitionNode.* Chaque *Nœud* de la hiérarchie de l'Instance a un *BrowsePath* qui peut être le même que celui de l'une des *InstanceDeclarations* de la hiérarchie du *TypeDefinitionNode.* L'*InstanceDeclaration* dont le *BrowsePath* est identique est l'*InstanceDeclaration* invoquée pour le *Nœud*. Si un *Nœud* a une *InstanceDeclaration,* il doit avoir le même *BrowseName* et la même *NodeClass* que l'*InstanceDeclaration* et, dans le cas des *Variables* et des *Objets*, le même *TypeDefinitionNode* ou un sous-type de celui-ci.

Les Instances peuvent référencer plusieurs *Nœuds* dont le *BrowsePath* est identique. Les *Clients* pour lesquels il est nécessaire de distinguer les *Nœuds* fondés sur l'*InstanceDeclarationHierarchy* des *Nœuds* qui ne sont pas fondés sur l'*InstanceDeclarationHierarchy* peuvent effectuer cette distinction en utilisant le *Service* TranslateBrowsePathsToNodeIds défini dans l'IEC 62541-4

6.4.2 Création d'une Instance

Les Instances héritent des valeurs initiales des *Attributs* qu'ils partagent avec le *TypeDefinitionNode* à partir duquel ils sont instanciés, à l'exception de la *NodeClass* et du *NodeId.*

Lorsqu'un *Serveur* crée une instance d'un *TypeDefinitionNode*, il doit créer la même hiérarchie de *Nœuds* sous le nouvel *Objet* ou la nouvelle *Variable*, en fonction de la *ModellingRule* de chaque *InstanceDeclaration*. Les *ModellingRules* normalisées sont définies en 6.4.4.5. Les *Nœuds* dans la hiérarchie nouvellement créée peuvent être des copies des *InstanceDeclarations*, de l'*InstanceDeclaration* proprement dite ou d'un autre *Nœud* dans l'*AddressSpace* ayant le même *TypeDefinitionNode* et le même *BrowseName*. Si de nouvelles copies sont créées, les valeurs d'*Attribut* des *InstanceDeclarations* sont alors utilisées comme valeurs initiales.

Un exemple simple d'un *TypeDefinitionNode* et d'une *Instance* est fourni à la Figure 15. Les *Nœuds* référencés par le *TypeDefinitionNode* sans *ModellingRule* n'apparaissent pas dans l'instance. Les *Instances* peuvent avoir des enfants ayant des *BrowseNames* dupliqués; cependant, seul l'un de ces enfants correspond à l'*InstanceDeclaration*.

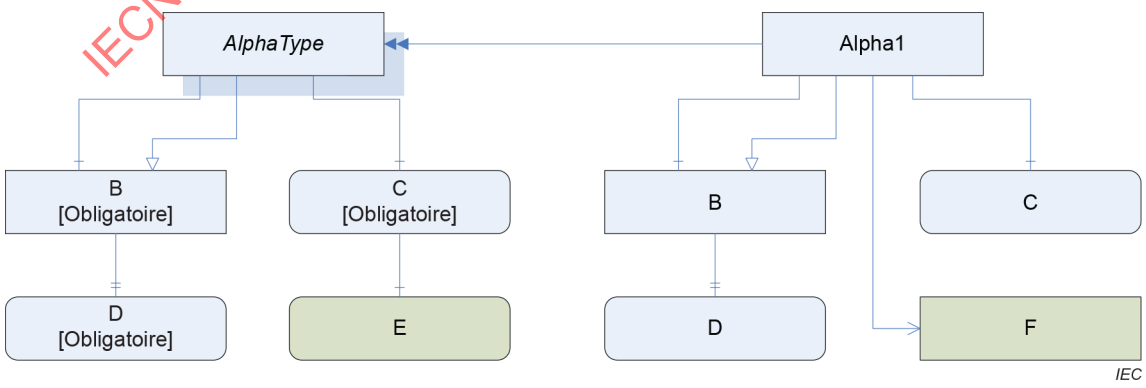


Figure 15 – Instance et son TypeDefinitionNode

Il incombe au *Serveur* de décider quelles *InstanceDeclarations* apparaissent dans toute instance unique. Dans certains cas, le *Serveur* ne définit pas l'ensemble de l'instance et

fournit des références distantes aux *Nœuds,* dans un autre *Serveur*. Les *ModellingRules* décrites en 6.4.4.5 autorisent que les *Serveurs* indiquent que certains *Nœuds* sont toujours présents; les *Clients* doivent toutefois se préparer à des situations où le *Nœud* existe dans un *Serveur* différent.

Un *Client* peut utiliser les informations des *TypeDefinitionNodes* pour accéder à des *Nœuds* qui sont dans la hiérarchie de l'instance. Il doit transmettre le *NodeId* de l'instance et le *BrowsePath* des *Nœuds* enfants, fondés sur le *TypeDefinitionNode,* au *Service* TranslateBrowsePathsToNodeIds (voir IEC 62541-4). Le *Service* renvoie le *NodeId* pour chacun des *Nœuds* enfants. Si un *Nœud* existe, le *BrowseName* et la *NodeClass* doivent correspondre à l'InstanceDeclaration. Dans le cas d'*Objets* ou de *Variables,* le *TypeDefinitionNode* doit également correspondre au *TypeDefinitionNode* initial ou être un sous-type de celui-ci.

6.4.3 Contraintes applicables à une Instance

Les *Objets* et *Variables* peuvent modifier leurs valeurs d'*Attribut* après création. Des règles particulières s'appliquent pour certains *Attributs* définis en 6.2.6.

Des *Références* supplémentaires peuvent être ajoutées aux *Nœuds* et les *Références* peuvent être supprimées tant que les *ModellingRules* définies sur des *InstanceDeclarations* du *TypeDefinitionNode* demeurent satisfaites.

Pour les *Variables* et *Objets,* la *Référence HasTypeDefinition* doit toujours pointer vers le même *TypeDefinitionNode* que l'*InstanceDeclaration* ou un sous-type de celui-ci.

Si deux *InstanceDeclarations* de l'*InstanceDeclarationHierarchy* intégralement héritée ont été directement reliées par plusieurs *Références*, toutes ces *Références* doivent relier les mêmes *Nœuds*. Un exemple est fourni à la Figure 16. Les instances A1 et A2 sont admises, étant donné que B1 référence le même *Nœud* en utilisant les deux *Références*, tandis qu'A3 n'est pas admis puisque deux *Nœuds* différents sont référencés. Cette restriction s'applique uniquement aux *Nœuds* directement reliés. Par exemple, A2 référence une C1 directement et une C1 différente via B1.

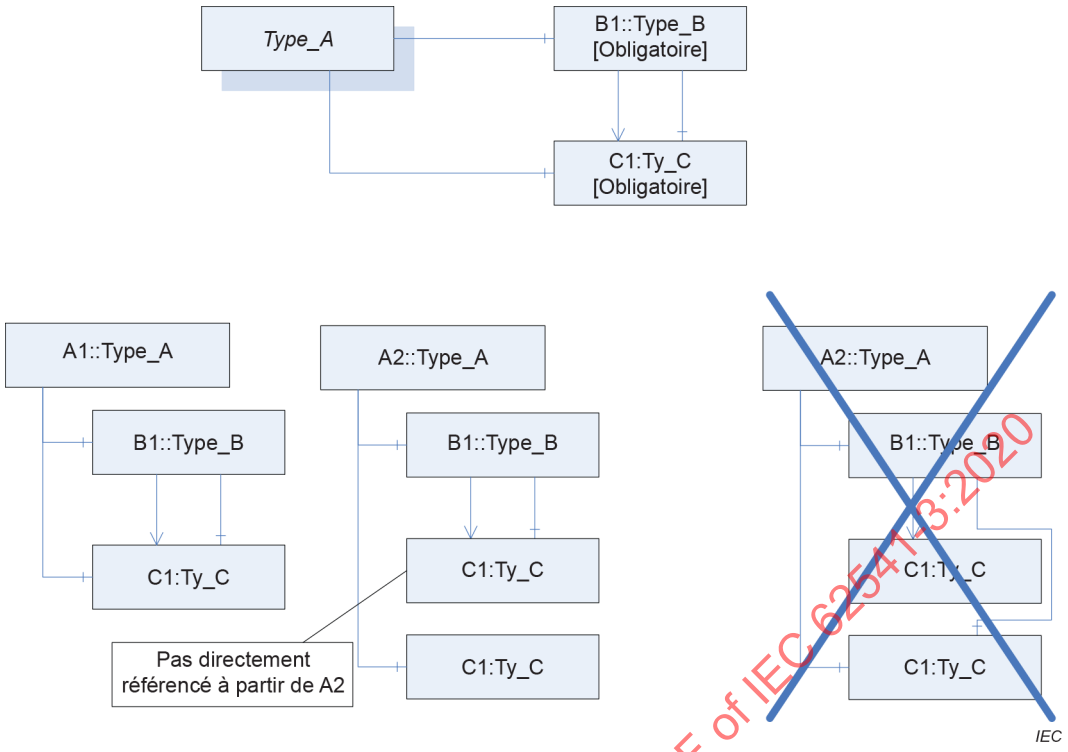


Figure 16 – Exemple de plusieurs Références entre InstanceDeclarations

6.4.4 ModellingRules

6.4.4.1 Généralités

Pour une définition des *ModellingRules*, voir 6.4.4.5. D'autres parties de l'IEC 62541 peuvent définir des *ModellingRules* supplémentaires. Dans OPC UA, les *ModellingRules* sont un concept extensible; par conséquent, les fournisseurs peuvent définir leurs propres *ModellingRules*.

Les *ModellingRules* définies dans le présent document ne spécifient pas la manière de traiter des *Références* non hiérarchiques entre *InstanceDeclarations*, c'est-à-dire qu'il incombe au *Serveur* d'établir si ces *Références* existent ou non dans une hiérarchie d'instance. D'autres *ModellingRules* peuvent également définir le comportement de *Références* non hiérarchiques entre *InstanceDeclarations*.

Les *ModellingRules* sont représentées dans *l'AddressSpace* comme des *Objets* de l'*ObjectType ModellingRuleType*. Certaines *Propriétés* définissent la sémantique commune aux *ModellingRules*. La présente édition de la norme spécifie uniquement une *Propriété* pour les *ModellingRules*. Les éditions futures peuvent définir des *Propriétés* supplémentaires pour les *ModellingRules*. L'IEC 62541-5 spécifie la représentation des *Objets ModellingRule,* leurs *Propriétés* et leurs types dans l'*AddressSpace*. La sémantique des *Propriétés* pour les *ModellingRules* est définie en 6.4.4.2.

Le 6.4.4.4 explique comment la *ModellingRule* peut être modifiée lorsque l'instanciation des *InstanceDeclarations* concerne les *Propriétés*. Le 6.4.4.3 explique comment la *ModellingRule* peut être modifiée lorsque le remplacement des *InstanceDeclarations* dans les sous-types concerne les *Propriétés*.

6.4.4.2 Propriétés décrivant les ModellingRules – NamingRule

La *NamingRule* est une *Propriété* obligatoire d'une *ModellingRule*. Elle spécifie le but d'une *InstanceDeclaration*. Chaque *InstanceDeclaration* référence une *ModellingRule*; ainsi, la *NamingRule* est définie pour chaque *InstanceDeclaration*.

Il est admis trois valeurs pour la *NamingRule* d'une *ModellingRule*: *Optional*, *Mandatory* et *Constraint*.

La sémantique suivante s'applique pendant toute la durée de vie d'une instance reposant sur un *TypeDefinitionNode* ayant une *InstanceDeclaration*.

Pour une instance A1 d'un *TypeDefinitionNode* "AlphaType" ayant une *InstanceDeclaration* B1 disposant d'une *ModellingRule* qui utilise la *NamingRule Optional*, la règle suivante s'applique: Pour chaque *BrowsePath* de "AlphaType" à B1, l'instance A1 peut ne pas avoir un *Nœud* similaire (voir 6.2.4) pour B1 avec le même *BrowsePath*. Si ce *Nœud* existe, alors le *Service* TranslateBrowsePathsToNodeIds (voir IEC 62541-4) renvoie ce *Nœud* comme première entrée de la liste.

Pour une instance A1 d'un "AlphaType" *TypeDefinitionNode* ayant une *InstanceDeclaration* B1 disposant d'une *ModellingRule* qui utilise la *NamingRule Mandatory,* la règle suivante s'applique: Pour chaque *BrowsePath* de "AlphaType" à B1, l'instance A1 doit avoir un *Nœud similaire* (voir 6.2.4) pour B1 avec le même *BrowsePath* si tous les *Nœuds* du *BrowsePath* existent. Par exemple, si un *Nœud* dans le *BrowsePath* a une *NamingRule Optional* et qu'il est omis dans l'instance, tous les enfants de ce *Nœud* sont également omis, indépendamment de leurs *ModellingRules*.

Si une *InstanceDeclaration* a une *ModellingRule* utilisant la *NamingRule Constraint,* elle indique que le *BrowseName* de l'*InstanceDeclaration* est sans importance, mais qu'une autre sémantique est définie avec la *ModellingRule*. Le *Service* TranslateBrowsePathsToNodeIds (voir IEC 62541-4) ne peut généralement pas être utilisé pour accéder à des instances reposant sur ces *InstanceDeclarations*.

6.4.4.3 Règles de sous-typage pour les Propriétés des ModellingRules

Il est admis que des sous-types remplacent des *ModellingRules* en ce qui concerne leurs *InstanceDeclarations*. De manière générale, les contraintes de sous-typage doivent être rendues plus contraignantes et non pas le contraire. Par conséquent, il n'est pas admis pour le supertype de spécifier qu'il doit exister une instance avec le nom (*Règle de Nommage* "*Mandatory"*) et, pour le sous-type, rendre la règle facultative (*Règle de Nommage* "*Optional"*). Le Tableau 20 précise les modifications admises pour les *Propriétés* lorsque les *ModellingRules* sont modifiées dans le sous-type.

Tableau 20 – Règles applicables aux Propriétés des ModellingRules en cas de sous-typage

	Valeur pour le super-type	Valeur pour le sous-type
NamingRule	Mandatory	Mandatory
NamingRule	Optional	Mandatory ou Optional
NamingRule	Constraint	Constraint

6.4.4.4 Règles d'instanciation pour les Propriétés des ModellingRules

Il existe deux cas d'utilisation différents lors de la création d'une instance A reposant sur un *TypeDefinitionNode* "Type_A". A est utilisé comme une instance normale ou comme l'*InstanceDeclaration* d'un autre *TypeDefinitionNode*.

Dans le premier cas, il n'est pas exigé que des instances nouvellement créées ou référencées, fondées sur des *InstanceDeclarations*, aient une *ModellingRule*; cependant, il est admis qu'elles aient une *ModellingRule* indépendante de la *ModellingRule* de leur *InstanceDeclaration*.

Un exemple est fourni à la Figure 17. Les instances A1, A2 et A3 sont des instances valides de "Type_A", même si B de A1 n'a aucune *ModellingRule* et si B de A3 a une *ModellingRule* différente de celle de B de "Type_A".

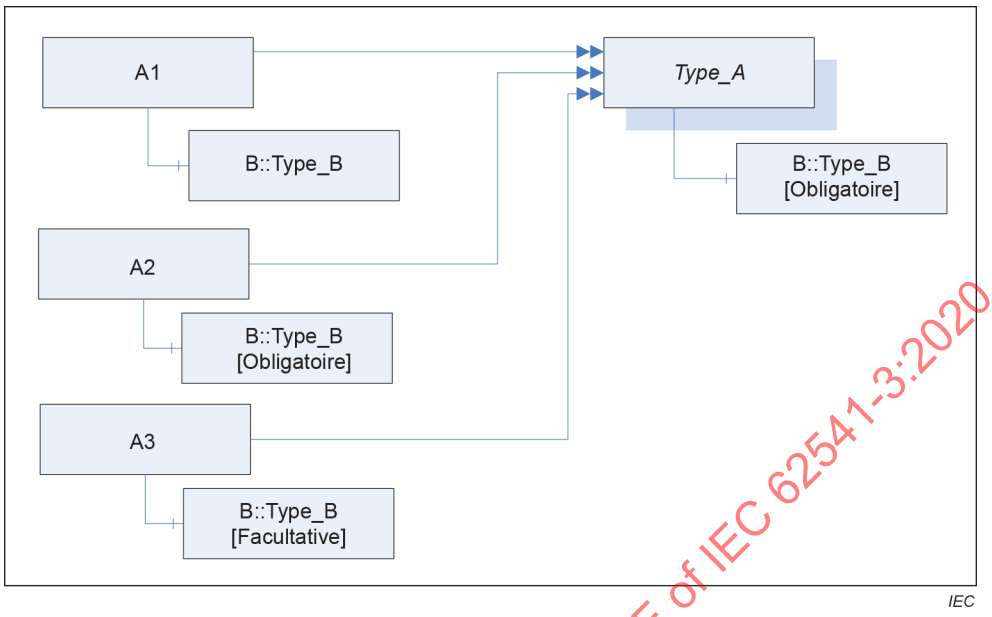


Figure 17 – Exemple de modification d'instances sur la base des InstanceDeclarations

Dans le second cas, toutes les instances qui sont directement ou indirectement référencées à partir de A et fondées sur des *InstanceDeclarations* de "Type_A", maintiennent initialement la même *ModellingRule* que leurs *InstanceDeclarations.* Les *ModellingRules* peuvent être mises à jour; les modifications autorisées aux *ModellingRules* de ces *Nœuds* sont les mêmes que celles définies pour le sous-typage en 6.4.4.3.

La Figure 18 donne un exemple de ce scénario. Le "Type_B" utilise une *InstanceDeclaration* fondée sur le "Type_A" (partie supérieure de la figure). Par la suite, la *ModellingRule* de l'*InstanceDeclaration* A1 est modifiée (partie inférieure de la figure). La *NamingRule* de A1 est devenue *Mandatory* (modifiée à partir de *Optional*).

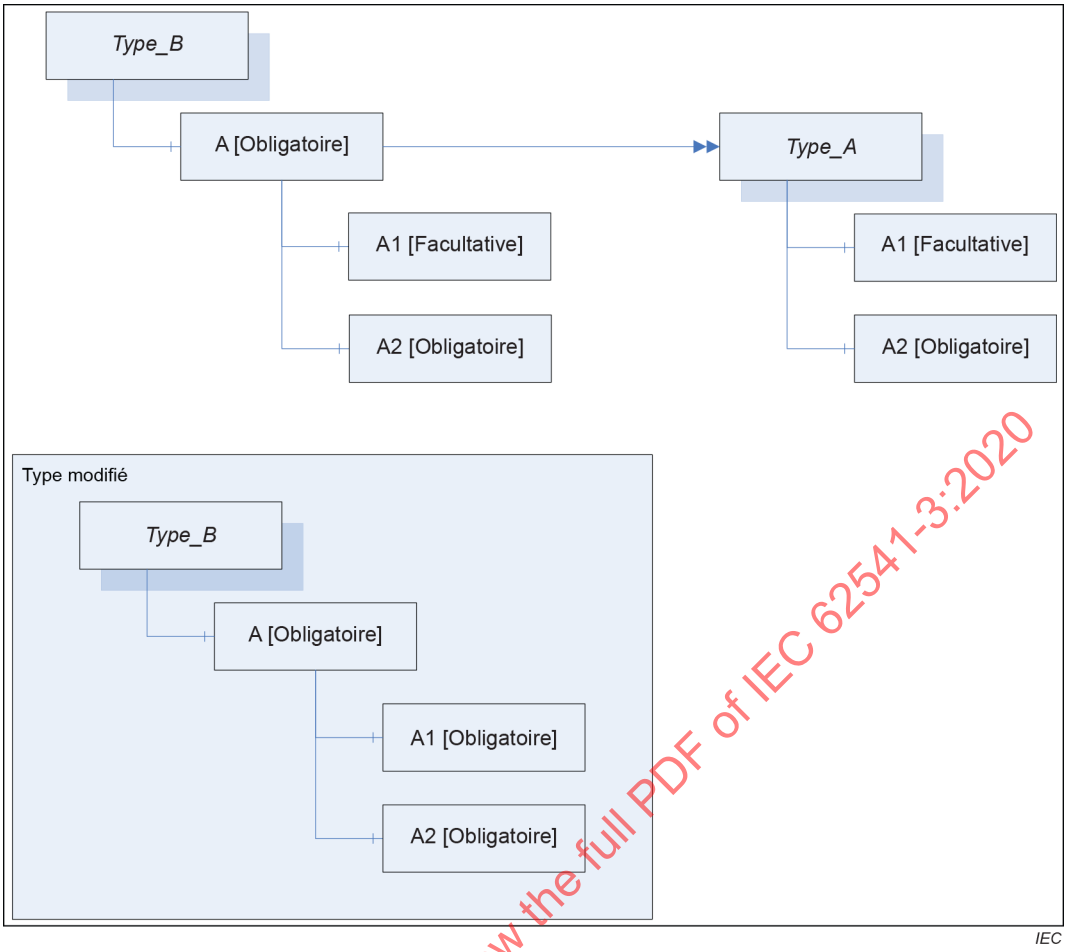


Figure 18 – Exemple de modification d'InstanceDeclarations reposant sur une InstanceDeclaration

6.4.4.5 ModellingRules normalisées

6.4.4.5.1 Intitulés des ModellingRules normalisées

La suite du 6.4.4.5 définit les *ModellingRules*. Le Tableau 21 répertorie les *Propriétés* de ces *ModellingRules*.

Tableau 21 – Propriétés des ModellingRules

Intitulé	NamingRule
Mandatory	Mandatory
Optional	Optional
ExposesItsArray	Constraint
OptionalPlaceholder	Constraint
MandatoryPlaceholder	Constraint

6.4.4.5.2 Mandatory

Une *InstanceDeclaration* marquée d'une *ModellingRule Mandatory* satisfait précisément à la sémantique définie pour la *NamingRule Mandatory*. Cela signifie que pour chaque *BrowsePath* existant sur l'instance, il doit exister un *Nœud* similaire; cependant, il n'est pas défini si un nouveau *Nœud* est créé ou si un *Nœud* existant est référencé.

Par exemple, le *TypeDefinitionNode* d'un bloc fonctionnel "AI_BLK_TYPE" aura un point de consigne "SP1". Une instance de ce type "AI_BLK_1" aura un point de consigne nouvellement créé "SP1" comme *Nœud* similaire à l'*InstanceDeclaration* SP1. La Figure 19 présente cet exemple.

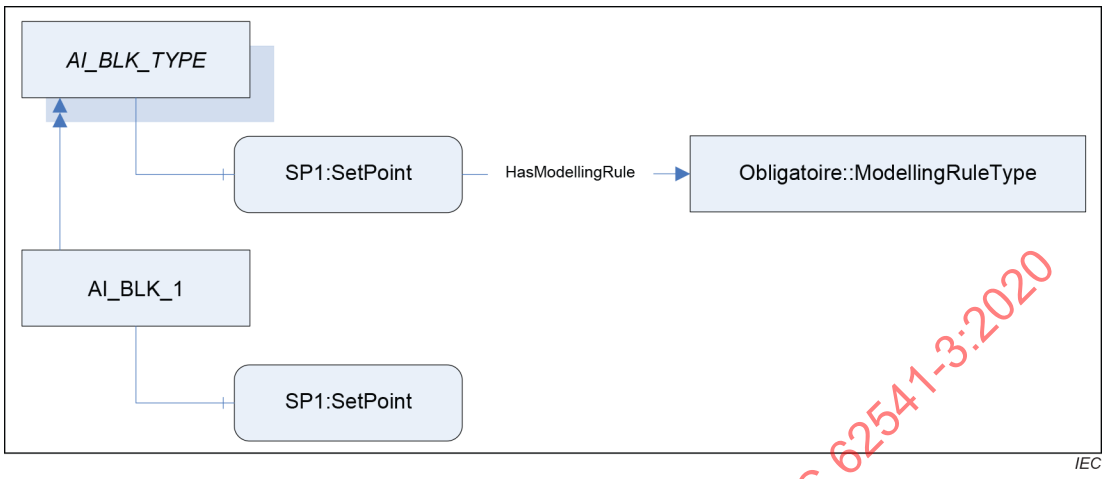


Figure 19 – Utilisation de la ModellingRule normalisée Mandatory

Un exemple complexe combinant des *ModellingRules Mandatory* et *Optional* est fourni en 6.4.4.5.3.

6.4.4.5.3 Optional

Une *InstanceDeclaration* marquée d'une *ModellingRule Optional* satisfait précisément à la sémantique définie pour la *NamingRule* Optional. Cela signifie que pour chaque *BrowsePath* existant sur l'instance, il peut exister un *Nœud* similaire, mais il n'est pas défini si un nouveau *Nœud* est créé ou si un *Nœud* existant est référencé.

Un exemple d'utilisation des *ModellingRules Optional* et *Mandatory* est présenté à la Figure 20. L'exemple contient un *ObjectType* "Type_A" et toutes les combinaisons valides d'instances nommées A1 à A13. Si le B facultatif est fourni, le E obligatoire doit également être fourni. Dans le cas contraire, il ne doit pas l'être. F est référencé par C et D. Dans l'instance, il peut s'agir du même *Nœud* ou de deux *Nœuds* différents ayant le même *BrowseName* (*Nœud* similaire à l'*InstanceDeclaration* F). L'exemple ne tient pas compte du fait que les instances ont ou n'ont pas de *ModellingRules*. Il est pris pour hypothèse que chaque F est similaire à l'*InstanceDeclaration* F, etc.

Dans le cas d'une *Référence* non hiérarchique entre E et F dans l'*InstanceDeclarationHierarchy*, il n'est pas spécifié si elle a lieu ou non dans la hiérarchie d'instance. Dans le cas de A10, il peut exister une référence à partir de E à l'un des deux F, aux deux ou à aucun d'entre eux.

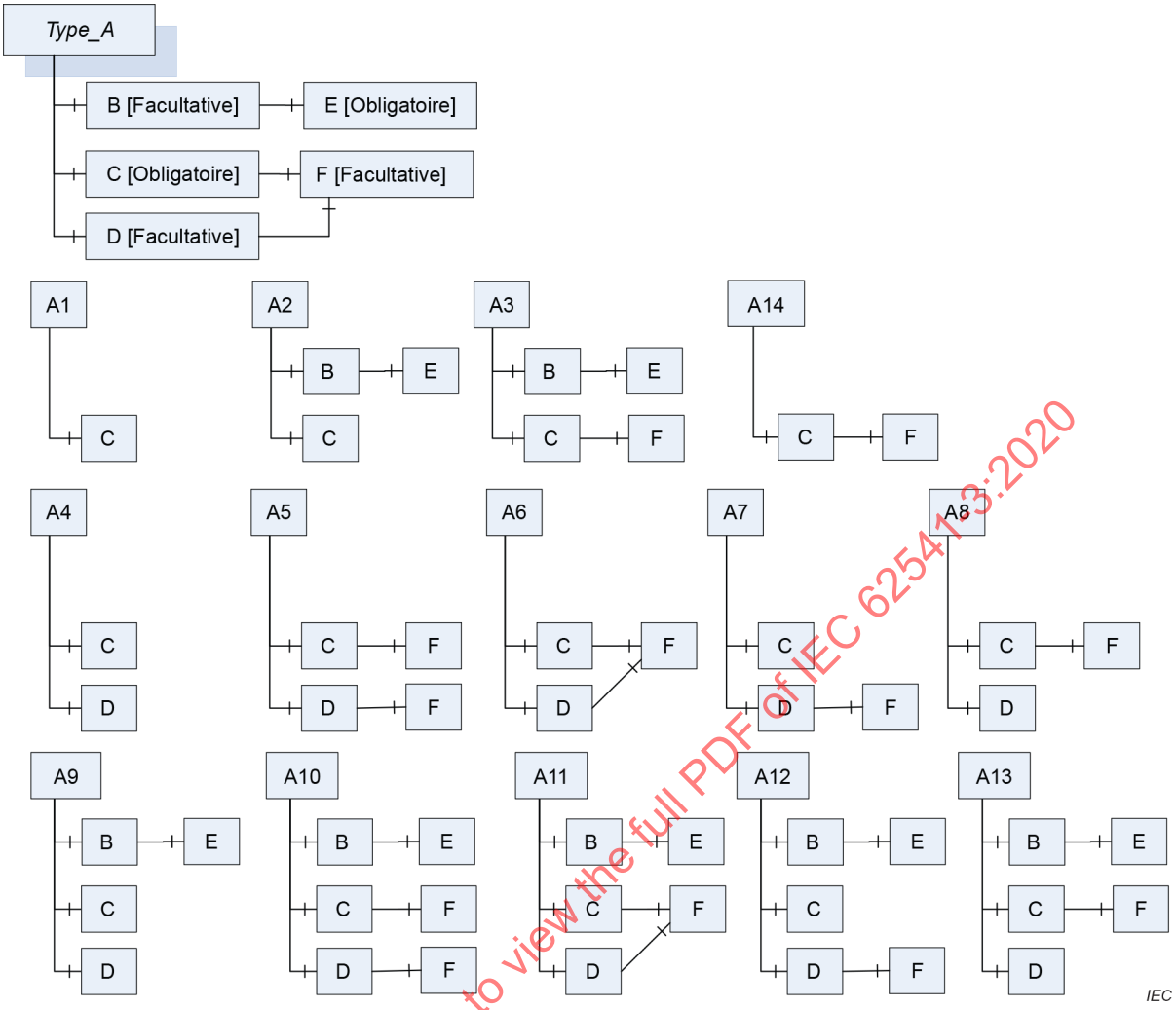


Figure 20 – Exemple d'utilisation des ModellingRules normalisées Optional et Mandatory

6.4.4.5.4 ExposesItsArray

La *ModellingRule ExposesItsArray* présente une sémantique particulière pour les *VariableTypes* ayant comme type de données une matrice unidimensionnelle ou multidimensionnelle. Elle indique que chaque valeur de la matrice est également présentée comme une *Variable* dans *l'AddressSpace*.

La *ModellingRule ExposesItsArray* peut uniquement s'appliquer qu'aux *InstanceDeclarations* de *NodeClass Variable* qui font partie d'un *VariableType* ayant comme type de données une matrice unidimensionnelle ou multidimensionnelle.

La *Variable* A ayant cette *ModellingRule* doit être référencée par une *Référence hiérarchique* directe à partir d'un *VariableType* B. B doit avoir une valeur *ValueRank* supérieure ou égale à zéro. Il convient que A ait un type de données qui reflète au moins des parties des données qui sont gérées dans la matrice de B. Chaque instance de B doit référencer une instance de A pour chacun de ses éléments de matrice. La *Référence* utilisée doit être du même type que la *Référence hiérarchique* qui relie B à A ou un sous-type de celui-ci. S'il existe plusieurs *Références hiérarchiques* directes entre A et B, toutes les instances reposant sur B doivent être référencées avec ces *Références*.

Un exemple est donné à la Figure 21. A est une instance de "Type_A" ayant deux entrées dans sa matrice de valeurs. Par conséquent, elle référence deux instances du même type que l*InstanceDeclaration* ArrayExpose. Les *BrowseNames* de ces instances ne sont pas définis

par la *ModellingRule*. En général, il n'est pas possible d'obtenir une *Variable* représentant une entrée spécifique de la matrice (par exemple la seconde). Les *Clients* tentent généralement d'obtenir la matrice ou d'accéder directement aux *Variables*, de sorte qu'il n'est pas nécessaire de fournir cette information.

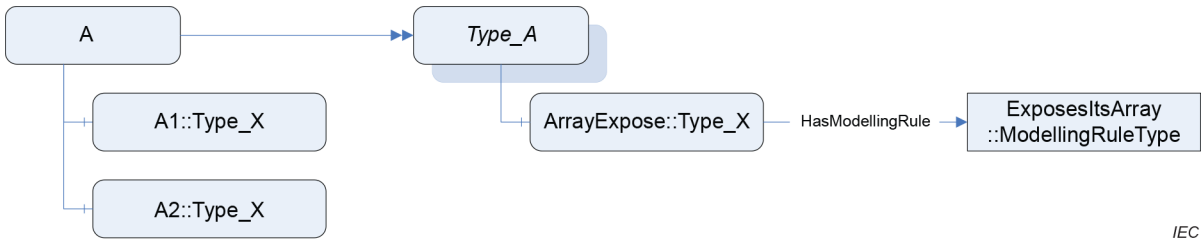


Figure 21 – Exemple d'utilisation de la ModellingRule ExposesItsArray

Il est également admis de référencer A par d'autres *InstanceDeclarations*. Ces *Références* doivent être reflétées sur chaque instance reposant sur A.

Un exemple est donné à la Figure 22. La *propriété* EUUnit est référencée par ArrayExpose; par conséquent, chaque instance reposant sur ArrayExpose référence l'instance reposant sur l'*InstanceDeclaration* EUUnit.

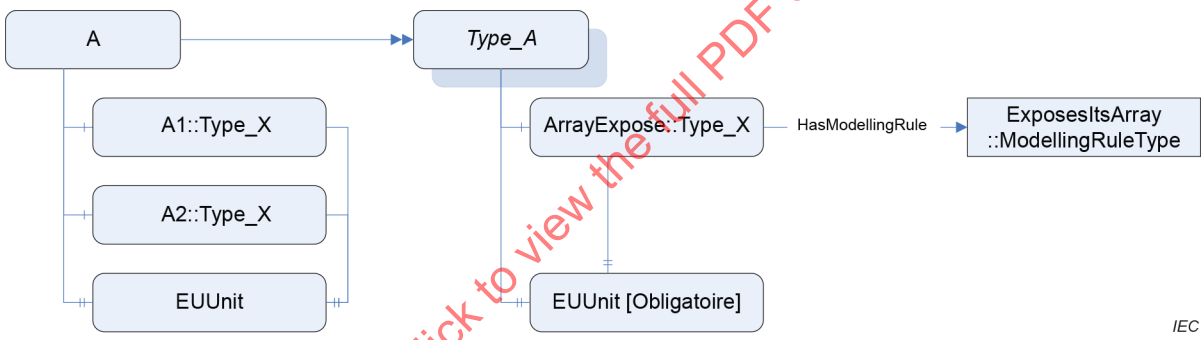


Figure 22 – Exemple complexe d'utilisation d'ExposesItsArray

6.4.4.5.5 OptionalPlaceholder

Pour l'*Objet* et la *Variable*, la *ModellingRule OptionalPlaceHolder* a pour objet de présenter l'information qu'une *TypeDefinition* complexe attend de la part des instances de la *TypeDefinition* pour ajouter des instances avec des *Références* spécifiques sans définir des *BrowseNames* pour les instances. Par exemple, un Appareil peut avoir un Dossier pour les DeviceParameters; il convient que les DeviceParameters soient reliés à la *Référence HasComponent*. Cependant, les noms des DeviceParameters sont spécifiques aux instances. L'exemple est présenté à la Figure 23, où une instance Device A ajoute deux DeviceParameters dans le Dossier.

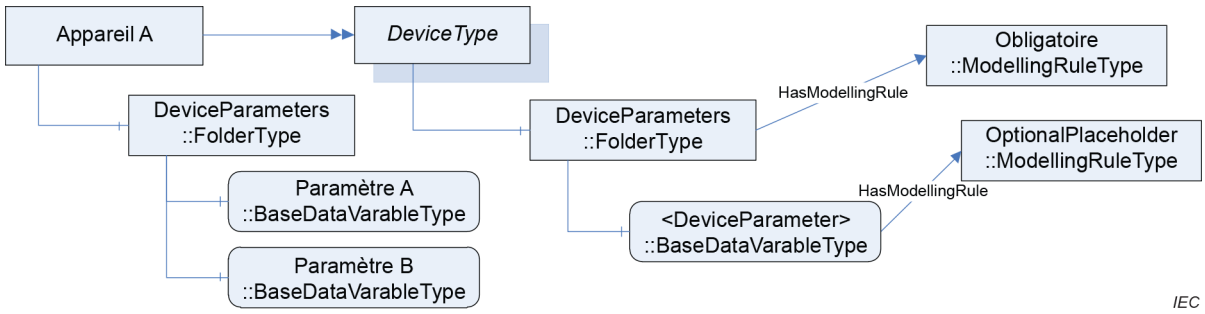


Figure 23 – Exemple d'utilisation d'OptionalPlaceholder avec un Objet et une Variable

La ModellingRule OptionalPlaceHolder n'ajoute aucune contrainte supplémentaire aux instances de la *TypeDefinition.* Elle ne fournit que des informations utiles lors de la présentation d'une *TypeDefinition.* Lorsque l'*InstanceDeclaration* est complexe, ce qui signifie qu'elle référence d'autres *InstanceDeclarations* avec des *Références* hiérarchiques, ces *InstanceDeclarations* ne sont plus par la suite prises en compte pour l'instanciation de la *TypeDefinition.*

Il est recommandé que le *BrowseName* et le *DisplayName* des *InstanceDeclarations* ayant la *ModellingRule OptionalPlaceHolder* soient placés entre crochets.

Lors du remplacement de l'*InstanceDeclaration*, la *ModellingRule* doit demeurer *OptionalPlaceHolder.*

Pour les *Méthodes,* la *ModellingRule OptionalPlaceholder* sert à définir le *BrowseName* où les sous-types et les instances apportent plus d'informations. La définition de la *Méthode* avec l'*OptionalPlaceholder* définit seulement le *BrowseName*. Une instance ou un sous-type définit les *InputArguments* et les *OutputArguments*. Un sous-type doit aussi modifier le caractère *Optional* ou *Mandatory* de la *ModellingRule*. La *Méthode* est facultative pour les *instances.* Par exemple, un Appareil peut avoir une *Méthode* pour réaliser les étalonnages, les arguments spécifiques pour la *Méthode* dépendent toutefois de l'instance de l'Appareil. Dans cet exemple, Device A ne met pas en œuvre la *Méthode*, Device B met en œuvre la *Méthode* sans argument et Device C met en œuvre la *Méthode* en acceptant un argument de mode pour choisir la manière dont l'étalonnage doit être exécuté. Cet exemple est représenté à la Figure 24.

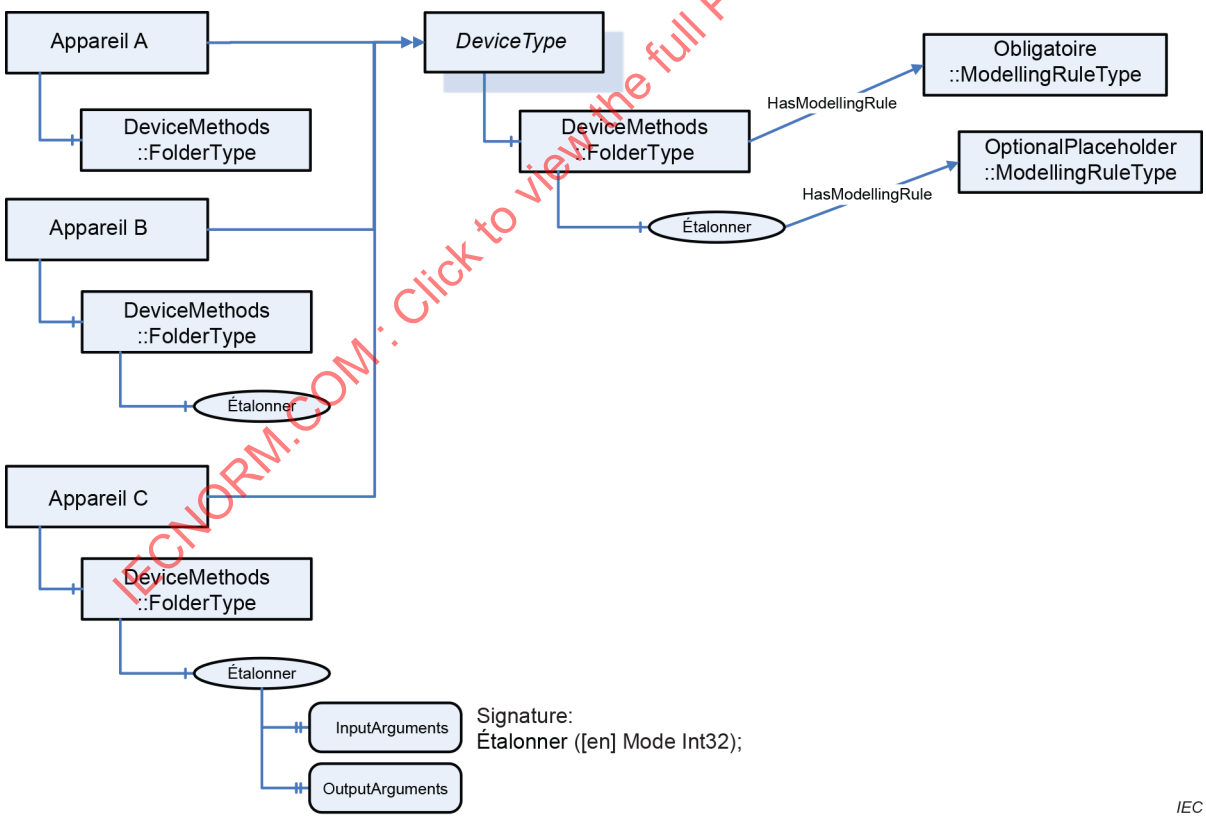


Figure 24 – Exemple d'utilisation d'OptionalPlaceholder avec une Méthode

6.4.4.5.6 MandatoryPlaceholder

Pour l'*Objet* et la *Variable*, la *ModellingRule MandatoryPlaceholder* a un objectif similaire à celle de la *ModellingRule OptionalPlaceholder*. Elle présente les informations qu'une *TypeDefinition* attend des instances de la *TypeDefinition* pour ajouter des instances définies

par l'*InstanceDeclaration.* Cependant, la *ModellingRule MandatoryPlaceHolder* exige l'existence d'au moins une de ces instances.

Par exemple, lorsque le DeviceType exige qu'au moins un DeviceParameter doive exister sans préciser son *BrowseName,* il utilise *MandatoryPlaceHolder* comme représenté à la Figure 25. Device A est une instance valide, car elle dispose du DeviceParameter exigé. Device B n'est pas valide puisqu'il utilise le *ReferenceType* erroné pour référencer un DeviceParameter (*Organizes* au lieu de *HasComponent)* et Device C n'est pas valide parce qu'il ne fournit aucun DeviceParameter.

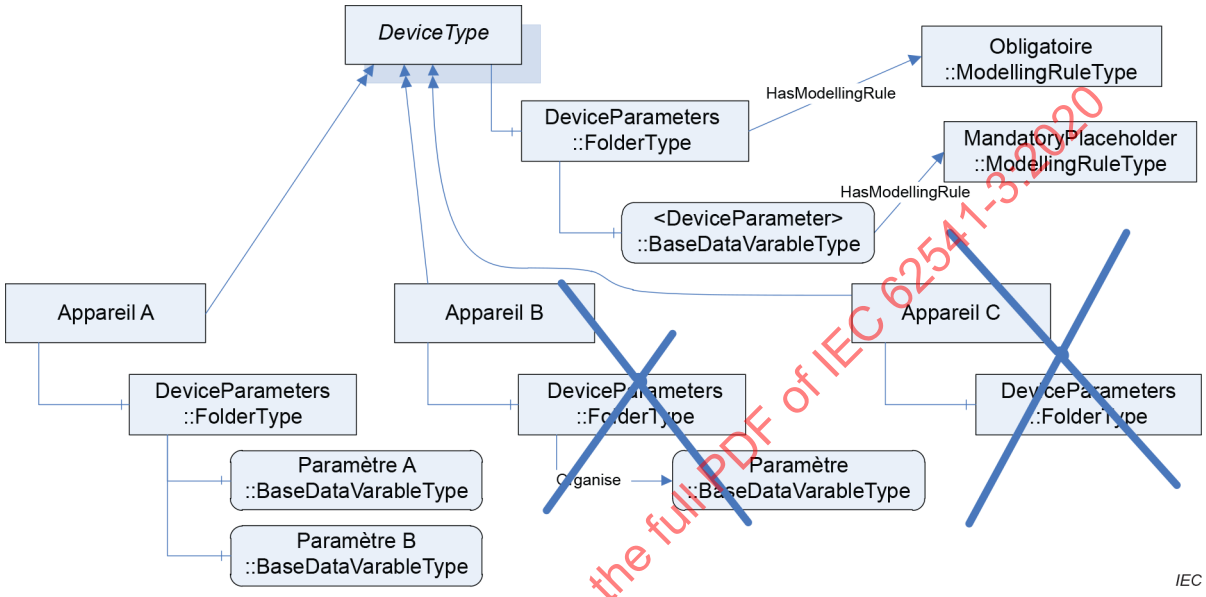


Figure 25 – Exemple d'utilisation de MandatoryPlaceholder pour un Objet et une Variable

La *ModellingRule MandatoryPlaceHolder* exige que chaque instance fournisse au moins une instance avec la *TypeDefinition* de l'*InstanceDeclaration* ou un sous-type, et soit référencée avec le même *ReferenceType* ou un sous-type comme l'*InstanceDeclaration.* Elle n'exige pas de *BrowseName* spécifique et ne peut donc pas être utilisée pour le *Service* TranslateBrowsePathsToNodeIds (voir IEC 62541-4).

Lorsque l'*InstanceDeclaration* est complexe, ce qui signifie qu'elle référence d'autres *InstanceDeclarations* avec des *Références hiérarchiques*, ces *InstanceDeclarations* ne sont plus par la suite prises en compte pour l'instanciation de la *TypeDefinition.*

Il est recommandé que le *BrowseName* et le *DisplayName des InstanceDeclarations* ayant la *ModellingRule MandatoryPlaceHolder* soient placés entre crochets.

Lors du remplacement de l'*InstanceDeclaration,* la *ModellingRule* doit demeurer *MandatoryPlaceHolder.*

Pour les *Méthodes,* la *ModellingRule MandatoryPlaceholder* sert à définir le *BrowseName* où les sous-types et les instances apportent plus d'informations. La définition de la *Méthode* avec le *MandatoryPlaceholder* définit uniquement le *BrowseName.* Une instance ou un sous-type définit les *InputArguments* et les *OutputArguments*. Un sous-type doit aussi modifier la *ModellingRule à Mandatory*. La *Méthode* est obligatoire pour les *instances.*

6.5 Modifications des définitions de type déjà utilisées

Aucun comportement n'est spécifié concernant les sous-types et les instances lorsque les *ObjectTypes* et les *VariableTypes* sont modifiés. Il incombe au *Serveur* d'établir si ces

modifications sont reflétées sur les sous-types et les instances de types. Cependant, toutes les contraintes définies pour les sous-types et les instances doivent être respectées. Par exemple, il n'est pas admis d'ajouter une *Propriété* en utilisant la *ModellingRule Mandatory* sur un type donné s'il existe des instances de ce type sans la *Propriété*. Dans ce cas, le *Serveur* doit ajouter la *Propriété* à l'ensemble des instances de type ou rejeter l'ajout de la *Propriété* ajoutée.

7 ReferenceTypes normalisés

7.1 Généralités

Le présent document définit les *ReferenceTypes* comme partie inhérente du Modèle d'espace d'adressage OPC UA. La Figure 26 fournit une description informelle de la hiérarchie de ces *ReferenceTypes*. D'autres parties de l'IEC 62541 peuvent spécifier des *ReferenceTypes* supplémentaires. La suite de l'Article 7 définit les *ReferenceTypes*. L'IEC 62541-5 définit leur représentation dans l'*AddressSpace*.

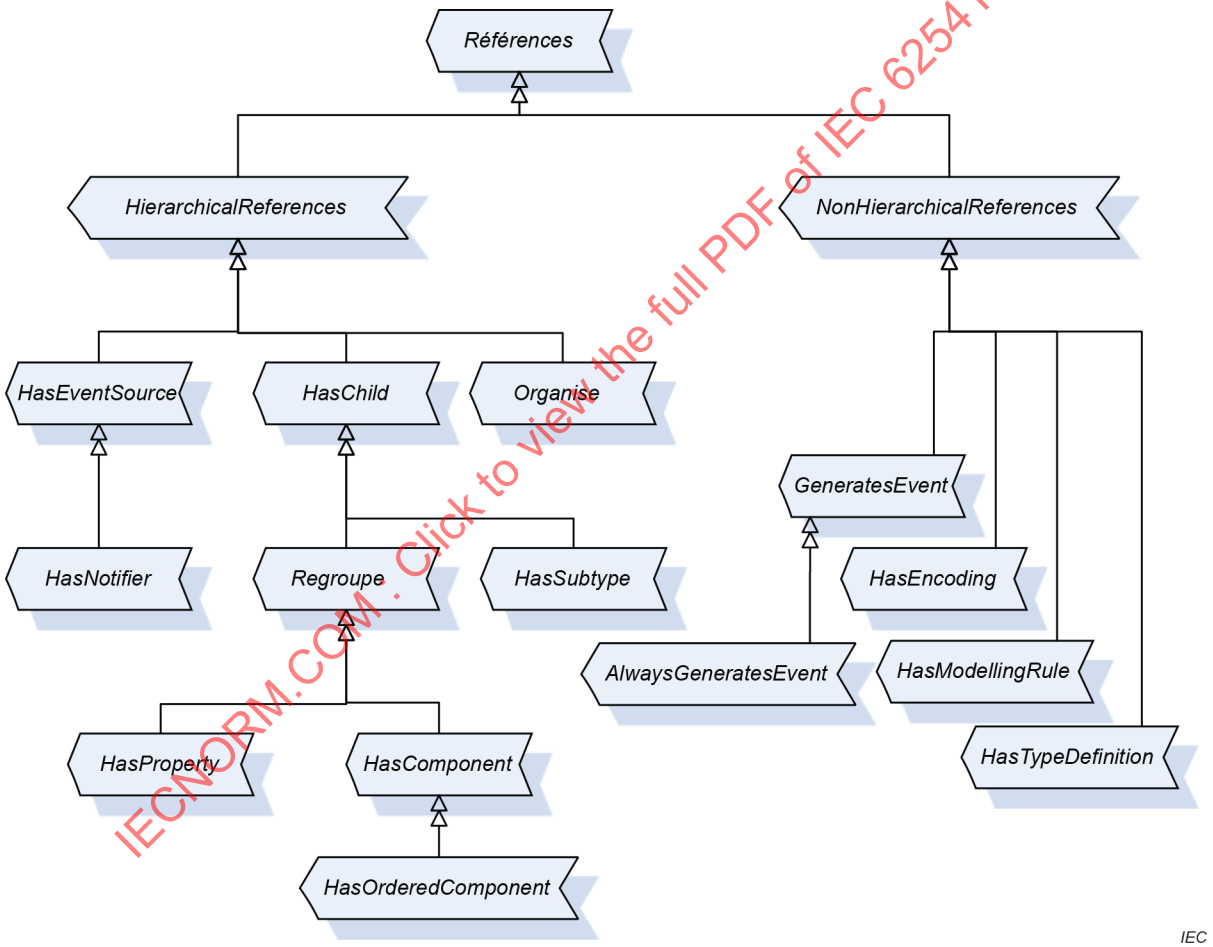


Figure 26 – Hiérarchie d'un ReferenceType normalisé

7.2 ReferenceType Références

Le *ReferenceType Références* est un *ReferenceType* abstrait; seuls ses sous-types peuvent être utilisés.

Aucune sémantique n'est associée à ce *ReferenceType*. Il s'agit du type de base de tous les *ReferenceTypes*. Tous les *ReferenceTypes* doivent être un sous-type de ce *ReferenceType* de base, directement ou indirectement. Le principal objectif de ce *ReferenceType* est de

permettre l'utilisation de filtres et de requêtes simples dans les *Services* correspondants de l'IEC 62541-5.

Il n'existe pas de contraintes définies pour ce *ReferenceType* abstrait.

7.3 ReferenceType HierarchicalReferences

Le *ReferenceType HierarchicalReferences* est un *ReferenceType* abstrait; seuls ses sous-types peuvent être utilisés.

La sémantique des *HierarchicalReferences* a pour objet d'indiquer que les *Références* des *HierarchicalReferences* couvrent une hiérarchie. Ceci signifie qu'il peut être utile de présenter des *Nœuds* liés à des *Références* de ce type de manière hiérarchique. Les *HierarchicalReferences* n'interdisent pas les boucles. Par exemple, en partant du *Nœud* A et en suivant les *HierarchicalReferences*, il peut être possible d'explorer de nouveau le *Nœud* A.

Il n'est pas admis d'avoir une *Propriété* comme *SourceNode* d'une *Référence* de quelque sous-type que ce soit de ce *ReferenceType* abstrait.

Le *SourceNode* et le *TargetNode* d'une *Référence* de *ReferenceType HierarchicalReferences* ne peuvent pas être identiques; en d'autres termes, il n'est pas admis d'avoir des autoréférences en utilisant des *HierarchicalReferences*.

7.4 ReferenceType NonHierarchicalReferences

Le *ReferenceType NonHierarchicalReferences* est un *ReferenceType* abstrait; seuls ses sous-types peuvent être utilisés.

La sémantique des *NonHierarchicalReferences* a pour objet de montrer que ses sous-types ne couvrent pas une hiérarchie et qu'il convient de ne pas les suivre pour tenter de présenter une hiérarchie. Pour faire la distinction entre les *Références hiérarchiques* et non hiérarchiques, tous les *ReferenceTypes* concrets doivent hériter soit des *Références hiérarchiques*, soit des *Références* non hiérarchiques, directement ou indirectement.

Il n'existe pas de contraintes définies pour ce *ReferenceType* abstrait.

7.5 ReferenceType HasChild

Le *ReferenceType HasChild* est un *ReferenceType* abstrait; seuls ses sous-types peuvent être utilisés. Il s'agit d'un sous-type des *HierarchicalReferences*.

La sémantique a pour objet d'indiquer que les *Références* de ce type couvrent une hiérarchie sans boucles.

En partant du *Nœud* A et en suivant uniquement les *Références* des sous-types du *ReferenceType HasChild*, il ne doit jamais être possible de revenir à A. Il est cependant admis qu'en suivant les *Références,* il peut exister plusieurs chemins menant vers un autre *Nœud* B.

7.6 ReferenceType

Le *ReferenceType Aggregates* est un *ReferenceType* abstrait; seuls ses sous-types peuvent être utilisés. Il s'agit d'un sous-type de*HasChild*.

La sémantique a pour objet d'indiquer qu'une partie (le *TargetNode*) appartient au *SourceNode*. Elle ne spécifie pas le propriétaire du *TargetNode*.

Il n'existe pas de contraintes définies pour ce *ReferenceType* abstrait.