IEC 62541-3

Edition 1.0 2010-07

INTERNATIONAL STANDARD

NORME INTERNATIONALE

colour inside

OPC unified architecture – Part 3: Address Space Model

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

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

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

About the IEC

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

About IEC publications

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

Useful links:

IEC publications search - www.iec.ch/searchpub

The advanced search enables you 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It also gives information on projects, replaced and withdrawn publications.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 on-line and also once a month by email.

Electropedia - www.electropedia.org

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

Customer Service Centre - webstore.iec.ch/csc

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

A propos de la CEI

La Commission Electrotechnique Internationale (CEI) 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 CEI

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

Liens utiles:

Recherche de publications CEI - www.iec.ch/searchpub

La recherche avancée vous permet de trouver des publications CEI 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.

Just Published CEI - webstore.iec.ch/justpublished

Restez informé sur les nouvelles publications de la CEI. Just Published détaille les nouvelles publications parues. Disponible en ligne et aussi une fois par mois par email.

Electropedia - www.electropedia.org

Le premier dictionnaire en ligne au monde de termes électroniques et électriques. Il contient plus de 30 000 termes et définitions en anglais et en français, ainsi que les termes équivalents dans les langues additionnelles. Egalement appelé Vocabulaire Electrotechnique International (VEI) en ligne.

Service Clients - webstore.iec.ch/csc

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

IEC 62541-3

Edition 1.0 2010-07

INTERNATIONAL STANDARD

NORME INTERNATIONALE

colour inside

OPC unified architecture – Part 3: Address Space Model

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

INTERNATIONAL ELECTROTECHNICAL COMMISSION

COMMISSION ELECTROTECHNIQUE INTERNATIONALE

PRICE CODE XE
CODE PRIX

ICS 25.040.40; 35.100

ISBN 978-2-83220-310-1

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 bilingual version (2012-10) corresponds to the monolingual English version, published in 2010-07.

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

FDIS	Report on voting
65E/160/FDIS	65E/164/RVD

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

The French version of this standard has not been voted upon.

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

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

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

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

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

INTRODUCTION

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

OPC UNIFIED ARCHITECTURE –

Part 3: Address Space Model

1 Scope

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

2 Normative references

The following referenced documents are indispensable for the application 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[1], *OPC Unified Architecture – Part 4: Services*

IEC 62541-5[1], *OPC Unified Architecture – Part 5: Information Model*

IEC 62541-6[1], *OPC Unified Architecture – Part 6: Mappings*

IEC 62541-8[1], *OPC Unified Architecture – Part 8: Data Access*

ISO/IEC 10918-1, *Information technology – Digital compression and coding of continuous-tone still images: Requirements and guidelines*

ISO/IEC 15948, *Information technology – Computer graphics and image processing – Portable Network Graphics (PNG): Functional specification*

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

XML Schema Part 1,available at <http://www.w3.org/TR/xmlschema-1/>

XML Schema Part 2, available at <http://www.w3.org/TR/xmlschema-2/>

XPATH, available at <http://www.w3.org/TR/xpath/>

IETF RFC 3066, *Tags for the Identification of Languages,* available at <http://tools.ietf.org/html/rfc3066>

IEEE 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic*, available at <http://ieeexplore.ieee.org/servlet/opac?punumber=2355>

---

1 To be published.

3 Terms, definitions, abbreviations and conventions

3.1 Terms and definitions

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

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
DataVariable
Variables that represent *values* of *Objects*, either directly or indirectly for complex *Variables*, where the *Variables* are always the *TargetNode* of a *HasComponent Reference*

3.1.3
EventType
ObjectType Node that represents the type definition of an *Event*

3.1.4
Hierarchical Reference
Reference that is used to construct hierarchies in the *AddressSpace*

NOTE All hierarchical *ReferenceTypes* are derived from *HierarchicalReferences*.

3.1.5
InstanceDeclaration
Node that is used by a complex *TypeDefinitionNode* to expose its complex structure; It is an instance used by a type definition

3.1.6
ModellingRule
metadata of an *InstanceDeclaration* that defines how the *InstanceDeclaration* will be used for instantiation; It also defines subtyping rules for an *InstanceDeclaration*

3.1.7
Property
Variables that are the *TargetNode* for a *HasProperty Reference*; *Properties* describe the characteristics of a *Node*

3.1.8
SourceNode
Node having a *Reference* to another *Node*; For example, in the *Reference* “A contains B”, “A” is the *SourceNode*

3.1.9
TargetNode
Node that is referenced by another *Nod;,* For example, in the *Reference* “A Contains B”, “B” is the *TargetNode*

3.1.10
TypeDefinitionNode
Node that is used to define the type of another *Node*; *ObjectType* and *VariableType Nodes* are *TypeDefinitionNodes*

3.1.11
VariableType
Node that represents the type definition for a *Variable*

3.2 Abbreviations

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.

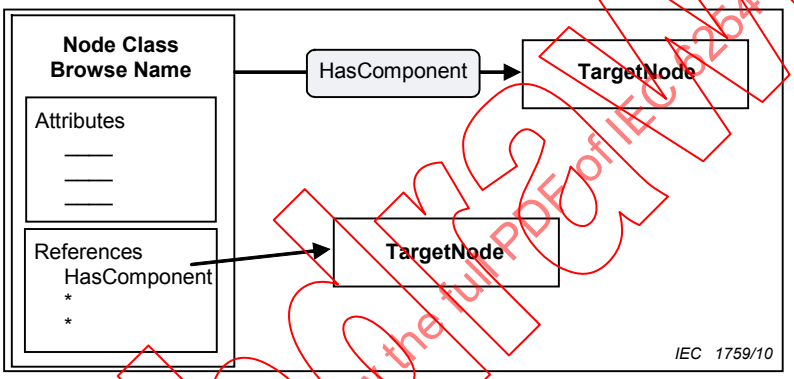


Figure 1 – AddressSpace Node diagrams

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 standard may define *Attributes*. Thus, all *Attributes* of the *NodeClass* are specified in the table and can only be extended by other parts of this series of standards.

This standard also defines *ReferenceTypes*, but *ReferenceTypes* can 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 standard can 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, subclasses of the *ReferenceType* may restrict its *SourceNodes*.

This standard defines *Properties*, but *Properties* can be defined by other standard organizations or vendors and *Nodes* can have *Properties* that are not standardised. *Properties* defined in this standard 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 implementation will chose to use *ModellingRules* appropriate for them.

4 AddressSpace concepts

4.1 Overview

The following subclauses define 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 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 the OPC Binary Types Description System as a format to specify data type structures and the normative Annex D 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.

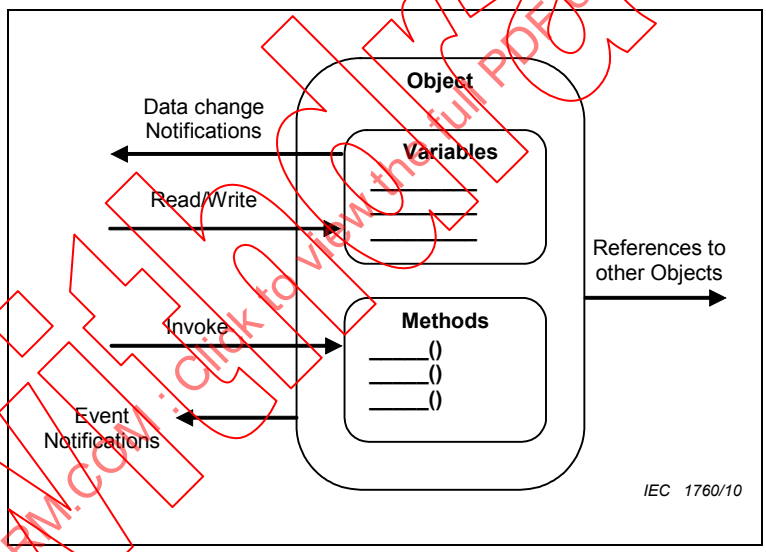


Figure 2 – OPC UA Object Model

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 this subclause discusses the details of the Node Model.

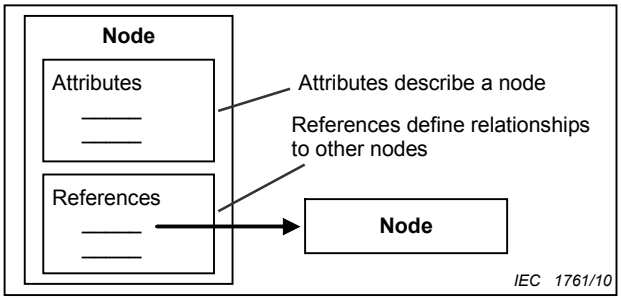


Figure 3 – AddressSpace Node Model

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* has to 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* are 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*.

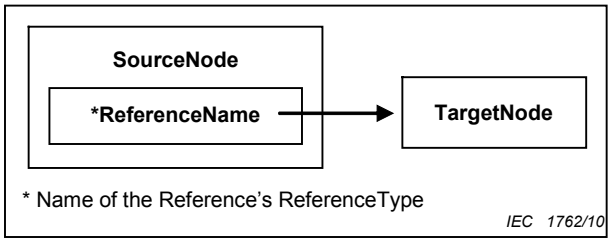


Figure 4 – Reference Model

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 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 characterise *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 specification 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", 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 specialised 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 *TypeDefinitionNodes* 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 specialised *VariableType* is used. Each setpoint *DataVariable* that uses this common definition will have a *HasTypeDefinition Reference* that identifies the common "SetPoint" *VariableType*.

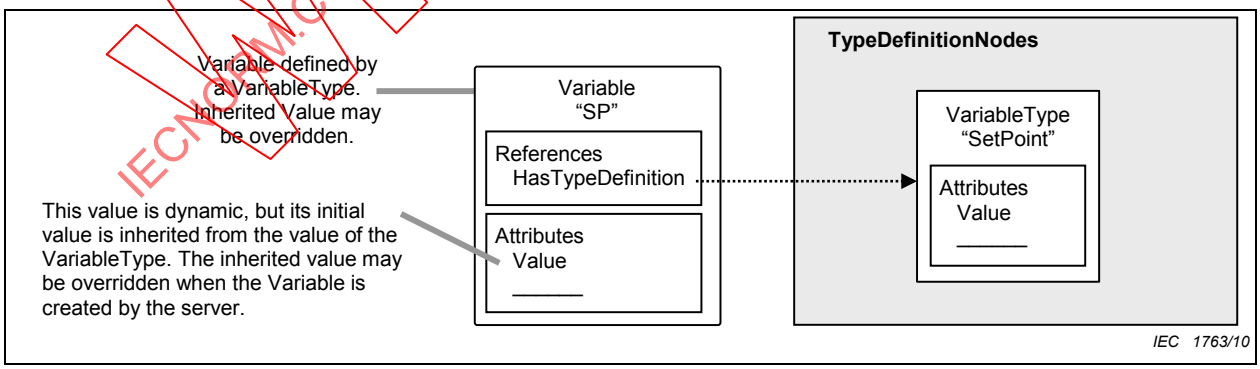


Figure 5 – Example of a Variable Defined by a VariableType

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.

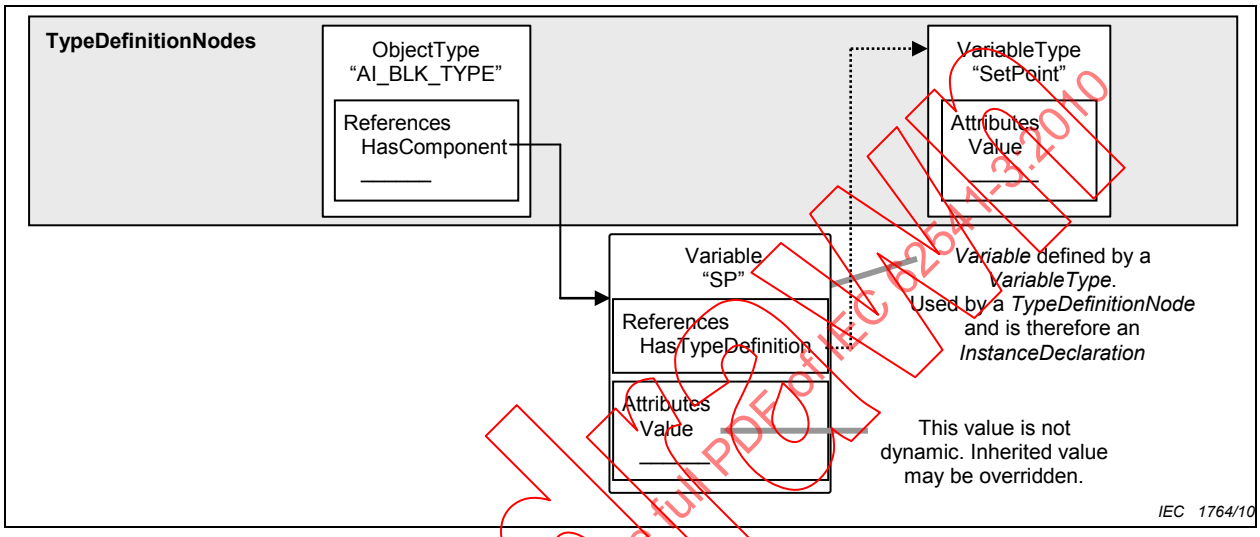


Figure 6 – Example of a Complex TypeDefinition

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 standard 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 are able to handle an instance of the subtype as if it were an instance of the supertype;
- instances of the supertype can be replaced by instances of the subtype;
- specialised 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*.

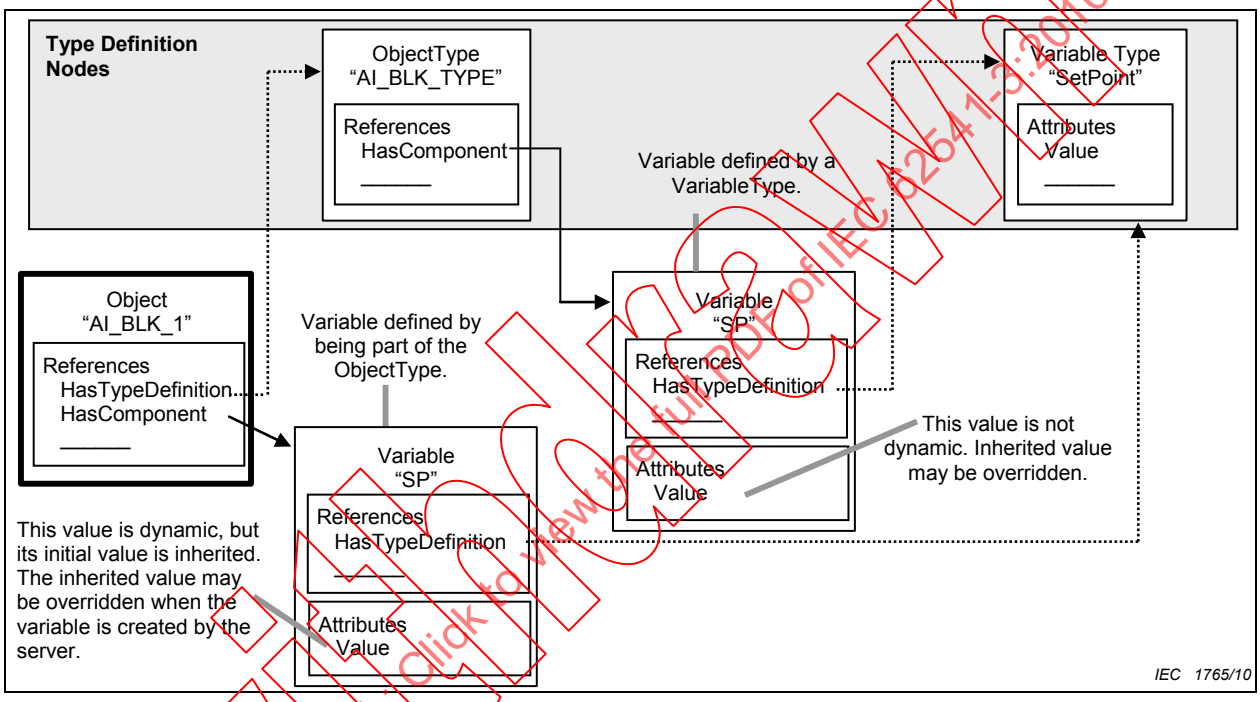


Figure 7 – Object and its Components defined by an ObjectType

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 *hierarchical References* in forward direction. 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 Monitoring and Subscription *Services* defined in IEC 62541-4 to subscribe to *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 part 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 part.

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 this standard or companion specifications as well as server specific *EventTypes* may be defined as not abstract and 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 categorised 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 organised into groups by using the *Event ReferenceTypes* described in 7.18 and 7.20. 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 categorised 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 categorised 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. Each *Method* is invoked within the context of an existing session. If the session is terminated during *Method* execution, the results of the *Method’s* execution cannot be returned to the client and are discarded. In that case, the Method execution is undefined, that is, the *Method* may be executed until it is finished or it may be aborted.

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

5 Standard NodeClasses

5.1 Overview

This clause 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 organise 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. 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 2. There are no *References* specified for the *Base NodeClass*.

Table 2 – 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	UInt32	See 5.2.7
UserWriteMask	O	UInt32	See 5.2.8
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*. 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-localised 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*.

Suclause 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 standard bodies defining standard type definitions use their namespace for the *NodeId* of the *TypeDefinitionNode* as well as for the *BrowseName* of the *TypeDefinitionNode.*

5.2.5 DisplayName

The *DisplayName Attribute* contains the localised 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 localised 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. 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 localised text using the same mechanisms for localisation 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 writeable 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 *WriteMask Attribute* is a 32-bit unsigned integer with the structure defined in Table 3. If the bit is set to 0, it means the *Attribute* is not writeable, if it is set to 1, it means it is writable. If a *Node* does not support a specific *Attribute*, the corresponding bit has to be set to 0.

Table 3 – 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.
Reserved	22:32	Reserved for future use. Shall always be zero.

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 same bit mask as used in the *WriteMask Attribute*, defined in Table 3.

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

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 4. In contrast, a *Reference* is an inherent part of a *Node* and no *NodeClass* is used to represent *References*.

This standard 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 4 – 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 *References* 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*, that 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 or deleted to 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 are inherited to its subtypes.

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

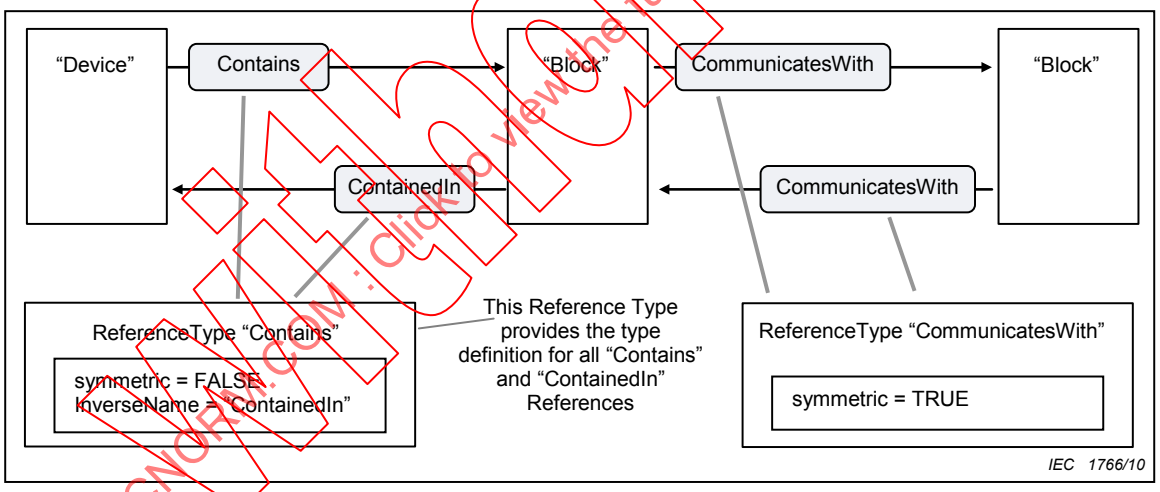


Figure 8 – Symmetric and Non-Symmetric References

It might not always be possible for servers to instantiate both forward and inverse *References* for non-symmetric *ReferenceTypes* as shown in Figure 8. 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 subscriber knows its publisher, but its publisher does not know its subscribers. The subscriber would have a "Subscribes To" *Reference* to the publisher, without the publisher having the corresponding "Publishes To" inverse *References* to its subscribers.

The *DisplayName* and the *InverseName* are the only standardised 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 standard does not specify how this semantic should be exposed. However, the

Description Attribute can be used for this purpose. This standard 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 shall never be able to return to A, that is, a “No Loop” constraint.

This standard 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 standard 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 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 standard. Additional parts of this series of standards 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 it is required that the subtype provides 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 inheritance.

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 specification 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 5.

All *Nodes* contained in a *View* shall be accessible starting from the *View Node* when browsing in the context of the *View*. The browse may take several hops, i.e. it is not necessary that all containing *Nodes* can be browsed directly from the *View Node*.

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 5 – 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 following References in the context of the View contains 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 or deleted to 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* 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 9.30, if *Nodes* are added to or deleted from the *View*. There are no additional *Properties* defined for *Views* in this document. Additional parts of this series of standards 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 6.

Table 6 – Object NodeClass

Name	Use	Data Type	Description
Attributes			
Base NodeClass Attributes	M	--	Inherited from the *Base NodeClass*. See 5.2
EventNotifier	M	Byte	The *EventNotifier Attribute* is used to indicate if the *Node* can be used to subscribe to *Events* or the read / write historic *Events*. The *EventNotifier* is an 8-bit unsigned integer with the structure defined in the following table: **Field** / **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*). Reserved / 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). Reserved / 4:7 / Reserved for future use. Shall always be zero. The second two bits also indicate if the history of the *Events* is available via the OPC UA server.
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.
HasModelParent	0..1		The *HasModelParent Reference* points to the *ModelParent* of the *Object* (see 6.6 for details on *ModelParents*).
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 7.18 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 7.20 for details.
Organizes	0..*		This *Reference* should be used only for *Objects* of the *ObjectType FolderType* (see 5.5.3).
HasDescription	0..1		This *Reference* shall be used only for *Objects* of the *ObjectType DataTypeEncodingType* (see 5.8.4).
<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 or deleted to 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 6.4.4.2.1 for details). This *Property* shall only be

Name	Use	Data Type	Description
			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 this series of standards 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.

An *Object* shall use at most one *HasModelParent Reference* to specify its *ModelParent* (see 6.6 for details).

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 7.18 and 7.20.

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 *InstanceDeclaration* (see 4.5) all *Nodes* referenced with *hierarchical References* in forward direction shall have unique *BrowseNames* in the context of this *Object*.

If the *Object* is created based on an *InstanceDeclaration*, 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 7.

Table 7 – 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 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 or deleted to 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 this series of standards 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 *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 following subclauses, 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*, 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 8.

Table 8 – 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.
ArrayDimensions	O	UInt32[]	This *Attribute* specifies the length of each dimension for an array value. The Attribute is intended to describe the capability of the *Variable*, not the current size. The number of elements shall be equal to the value of the *ValueRank Attribute*. Shall be null if *ValueRank* ≤0. A value of 0 for an individual dimension indicates that the dimension has a variable length. 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, the *Variable's ValueRank* has the value 1 and the *ArrayDimensions* is an array with one entry having the value 346.
AccessLevel	M	Byte	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 writeable this may be restricted to a certain user / user group. The *AccessLevel* is an 8-bit unsigned integer with the structure defined in the following table:

<table>
<tr><th>Name</th><th>Use</th><th>Data Type</th><th>Description</th></tr>
<tr><td></td><td></td><td></td><td>
<table>
<tr><th>Field</th><th>Bit</th><th>Description</th></tr>
<tr><td>CurrentRead</td><td>0</td><td>Indicates if the current value is readable
(0 means not readable, 1 means readable).</td></tr>
<tr><td>CurrentWrite</td><td>1</td><td>Indicates if the current value is writable
(0 means not writable, 1 means writable).</td></tr>
<tr><td>HistoryRead</td><td>2</td><td>Indicates if the history of the value 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 value is writable (0 means not writable, 1 means writable).</td></tr>
<tr><td>SemanticChange</td><td>4</td><td>Indicates if the Variable used as Property generates SemanticChangeEvents (see 9.31).</td></tr>
<tr><td>Reserved</td><td>5:7</td><td>Reserved for future use. Shall always be zero.</td></tr>
</table>
The first two bits also indicate if a current value of this Variable is available and the second two bits indicates if the history of the Variable is available via the OPC UA server.
</td></tr>
<tr><td>UserAccessLevel</td><td>M</td><td>Byte</td><td>
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 UserAccessLevel 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>CurrentRead</td><td>0</td><td>Indicates if the current value is readable (0 means not readable, 1 means readable).</td></tr>
<tr><td>CurrentWrite</td><td>1</td><td>Indicates if the current value is writable (0 means not writable, 1 means writable).</td></tr>
<tr><td>HistoryRead</td><td>2</td><td>Indicates if the history of the value 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 value 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 first two bits also indicate if a current value of this Variable is available and the second two bits indicate if the history of the Variable is available via the OPC UA server.
</td></tr>
<tr><td>MinimumSamplingInterval</td><td>O</td><td>Duration</td><td>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.</td></tr>
<tr><td>Historizing</td><td>M</td><td>Boolean</td><td>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.</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>HasModellingRule</td><td>0..1</td><td></td><td>Variables can point to at most one ModellingRule Object using a HasModellingRule Reference (see 6.4.4 for details on ModellingRules).</td></tr>
<tr><td>HasProperty</td><td>0..*</td><td></td><td>HasProperty References are used to identify the Properties of a DataVariable.
Properties are not allowed to be the SourceNode of HasProperty</td></tr>
</table>

Name	Use	Data Type	Description
			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.
HasModelParent	0..1		The *HasModelParent Reference* points to the *ModelParent* of the *Variable* (see 6.6 for details on *ModelParents*).
<other References>	0..*		*Data Variables* may be the *SourceNode* of any other *References*. *Properties* may only be the *SourceNode* of any *non-hierarchical Reference*.
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 or deleted to 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.
DataTypeVersion	O	String	Only used for *Variables* of the *VariableType DataTypeDictionaryType* and *DataTypeDescriptionType* as described in 5.8.
DictionaryFragment	O	ByteString	Only used for *Variables* of the *VariableType DataTypeDescriptionType* as described in 5.8.
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.

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, it should state that it is read and writable. If a read or write operation is called on the *Variable*, 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 *SemanticChange* bit of the *AccessLevel Attribute* shall be set when the *Property* describes the semantic of the *Node* that owns the *Property* and changes of the *Property* value will generate *SemanticChangeEvents*. For example, a *Property* describing the engineering unit of a *DataVariable* has the bit set, whereas a *Property* containing an Icon of the *DataVariable* will not. This behaviour is exactly the same as described by the *SemanticsChanged* bit of the *StatusCode* defined in IEC 62541-4. However, if you subscribe to a *Variable* you should look at the *StatusCode* to identify if the semantic has changed in order to receive this information before you are processing the value of the *Variable*.

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, 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.

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.

A *Variable* shall use at most one *HasModelParent Reference* to specify its *ModelParent* (see 6.6 for details).

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 following subclauses.

5.6.3 Properties

Properties are used to define the characteristics of *Nodes*. *Properties* are defined using the *Variable NodeClass*, specified in Table 8. 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 8.

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 DataTypeVersion* is used only for *DataTypeDictionaries* and *DataTypeDescriptions* as defined in 5.8. The Standard *Property DictionaryFragment* is used only for *DataTypeDescriptions* as defined in 5.8. The *Property AllowNulls* indicates if NULL values are allowed for the *Value Attribute*. There are no additional *Properties* defined for *DataVariables* in this part of this document. Additional parts of this multi-part specification 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 *hierarchical References* in forward direction 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*, specified in Table 9.

Table 9 – 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.
ArrayDimensions	O	UInt32[]	This *Attribute* specifies the length of each dimension for an array value. The Attribute is intended to describe the capability of the *VariableType*, not the current size. The number of elements shall be equal to the value of the *ValueRank Attribute*. Shall be null if *ValueRank* ≤0. A value of 0 for an individual dimension indicates that the dimension has a variable length. 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.
<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 or deleted to 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 this series of standards 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 *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 10.

Table 10 – 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*).
HasModelParent	0..1		The *HasModelParent Reference* points to the *ModelParent* of the *Method* (see 6.6 for details on *ModelParents*).
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 or deleted to 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, 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 *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".

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 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 this multi-part specification 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.

A *Method* shall use at most one *HasModelParent Reference* to specify its *ModelParent* (see 6.6 for details).

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, 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 *hierarchical References* in forward direction 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 complex 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 Figure 9.

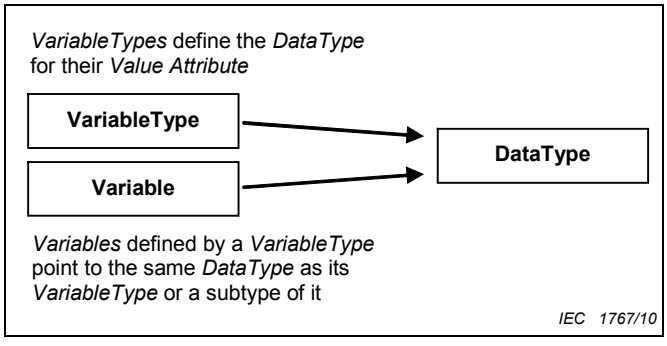


Figure 9 – Variables, VariableTypes and their DataTypes

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 10 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*. The *DataTypeEncoding Object* points to exactly one *Variable* of type *DataTypeDescriptionType*. The *DataTypeDescription Variable* belongs to a *DataTypeDictionary Variable*.

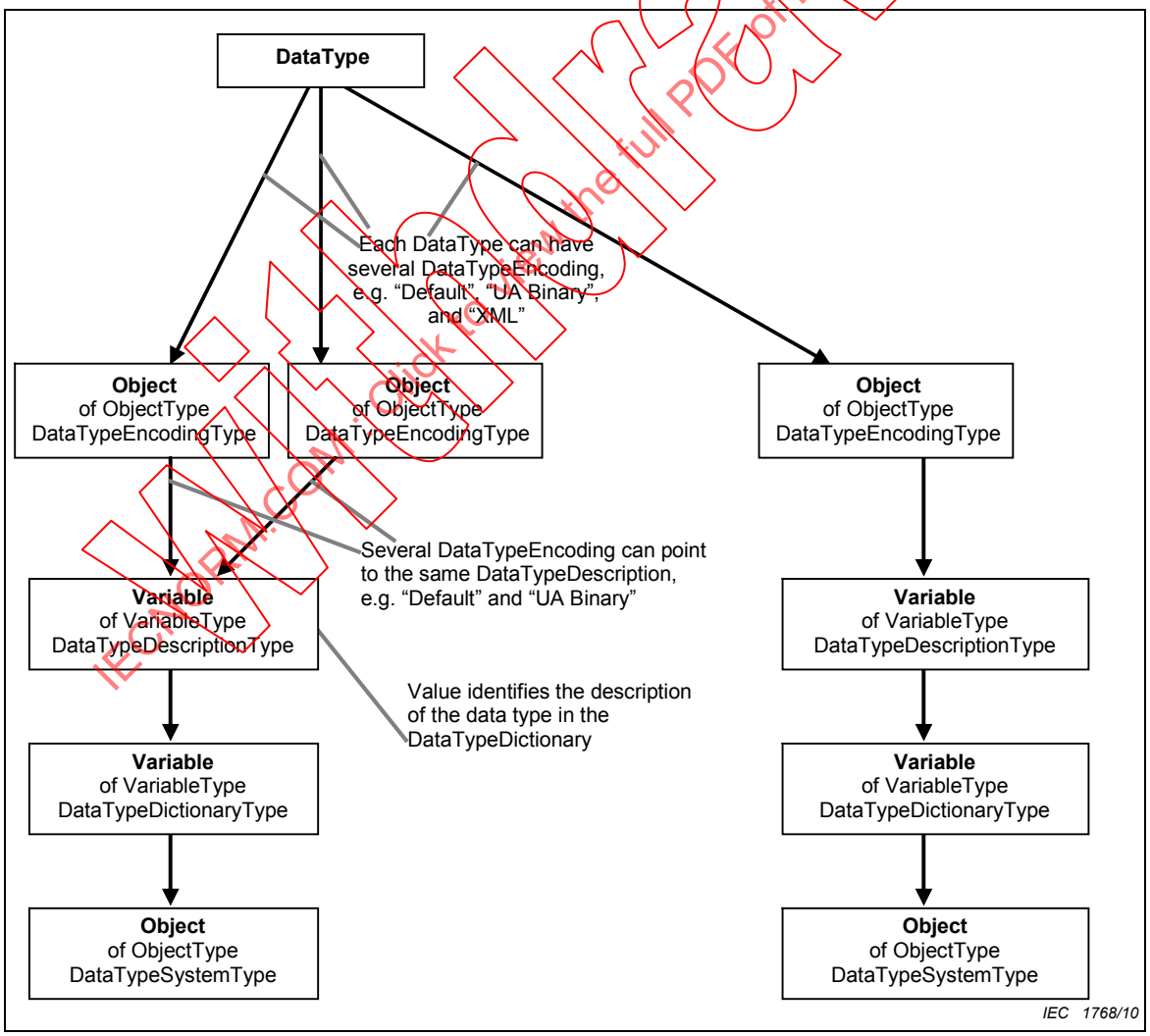


Figure 10 – DataType Model

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

The *DataTypeDictionary* describes a set of *DataTypes* in sufficient detail to allow clients to parse/interpret *Variable Values* that they receive and to construct *Values* that they send. The *DataTypeDictionary* is represented as a *Variable* of type *DataTypeDictionaryType* in the *AddressSpace*, the description about the *DataTypes* is contained in its *Value Attribute*. All containing *DataTypes* exposed in the *AddressSpace* are represented as *Variables* of type *DataTypeDescriptionType*. The *Value* of one of these Variables identifies the description of a *DataType* in the *Value Attribute* of the *DataTypeDictionary*.

The *DataType* of a *DataTypeDictionary Variable* is always a ByteString. The format and conventions for defining *DataTypes* in this ByteString are defined by *DataTypeSystems*. *DataTypeSystems* are identified by *NodeIds*. They are represented in the *AddressSpace* as *Objects* of the *ObjectType DataTypeSystemType*. Each *Variable* representing a *DataTypeDictionary* references a *DataTypeSystem Object* to identify their *DataTypeSystem*.

A client must recognise the *DataTypeSystem* to parse any of the type description information. OPC UA clients that do not recognise a *DataTypeSystem* will not be able to interpret its type descriptions, and consequently, the values described by them. In these cases, clients interpret these values as opaque ByteStrings.

OPC Binary and W3C XML Schema are examples of *DataTypeSystems*. The OPC Binary *DataTypeSystem* is defined in Annex C. OPC Binary uses XML to describe binary data values. W3C XML Schema is specified in XML Schema Part 1 and XML Schema Part 2

5.8.2 Encoding Rules for different kinds of DataTypes

Different kinds of *DataTypes* are distinguished between and are handled differently regarding their encoding and 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 a *Property EnumStrings* containing an array of *LocalizedText*. The Integer representation of the enumeration value points to a position of that array. An example of an enumeration *DataType* is *NodeClass* 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 defined in 8.24.

5.8.3 DataType NodeClass

The *DataType NodeClass* describes the syntax of a *Variable Value*. The *DataTypes* may be simple or complex, depending on the *DataTypeSystem*. *DataTypes* are defined using the *DataType NodeClass*, specified in Table 11.

Table 11 – 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*.
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 or deleted to 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.
EnumStrings	O	LocalizedText[]	The *EnumStrings Property* only applies for *Enumeration DataTypes*. It shall not be applied for other *DataTypes*. 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.

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*. This Version is not affected by the *DataTypeVersion Property* of *DataTypeDictionaries* and *DataTypeDescriptions*. The *Property EnumStrings* contains human-readable representations of enumeration values and is only applied to *Enumeration DataTypes*. There are no additional *Properties* defined for *DataTypes* in this standard. Additional parts of this series of standards may define additional *Properties* for *DataTypes*.

HasSubtype References may be used to expose a data type hierarchy in the *AddressSpace*. This hierarchy shall reflect the hierarchy specified in the *DataTypeDictionary*. The semantic of subtyping depends on the *DataTypeSystem*. Servers need not provide *HasSubtype References*, even if their *DataTypes* span a type hierarchy. Clients should not make any assumptions about any other semantic with that information than provided by the *DataTypeDictionary*. For example, it might not be possible to cast a value of one data type to its base data type.

HasEncoding References point from the *DataType* to its *DataTypeEncodings*. Following such a *Reference*, the client can browse to the *DataTypeDictionary* describing the structure of the *DataType* for the used encoding. 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 DataTypeDictionary, DataTypeDescription, DataTypeEncoding and DataTypeSystem

A *DataTypeDictionary* is an entity that contains a set of type descriptions, such as an XML schema. *DataTypeDictionaries* are defined as *Variables* of the *VariableType DataTypeDictionaryType*.

A *DataTypeSystem* specifies the format and conventions for defining *DataTypes* in *DataTypeDictionaries*. *DataTypeSystems* are defined as *Objects* of the *ObjectType DataTypeSystemType*.

The *ReferenceType* used to relate *Objects* of the *ObjectType DataTypeSystemType* to *Variables* of the *VariableType DataTypeDictionaryType* is the *HasComponent ReferenceType*. Thus, the *Variable* is always the *TargetNode* of a *HasComponent Reference*; a requirement for *Variables*. However, for *DataTypeDictionaries* the server shall always provide the inverse *Reference*, since it is necessary to know the *DataTypeSystem* when processing the *DataTypeDictionary*.

An example of a *DataTypeDictionary* is an XML document containing an XML schema. In this case, the *DataTypeSystem* is the W3C XML Schema and the top level element declarations in the schema document are the data type descriptions. Each of these descriptions is defined in different versions of an XML schema using the same XML target namespace. This target namespace is used as the namespace component of the *DataTypeId* in the server's *AddressSpace*. Since the same target namespace can be used in other XML schemas, clients shall be aware that two *DataTypeIds* with the same namespace are not necessarily defined in the same *DataTypeDictionary*.

Changes may be a result of a change to a type description, but it is more likely that dictionary changes are a result of the addition or deletion of type descriptions. This includes changes made while the server is offline so that the new version is available when the server restarts.

Clients may subscribe to the *DataTypeVersion Property* to determine if the *DataTypeDictionary* has changed since it was last read.

The server may, but is not required to, make the *DataTypeDictionary* contents available to clients through the *Value Attribute*. Clients should assume that *DataTypeDictionary* contents are relatively large and that they will encounter performance problems if they automatically read the *DataTypeDictionary* contents each time they encounter an instance of a specific *DataType*. The client should use the *DataTypeVersion Property* to determine whether the locally cached copy is still valid. If the client detects a change to the *DataTypeVersion*, then it shall re-read the *DataTypeDictionary*. This implies that the *DataTypeVersion* shall be updated by a server even after restart since clients may persistently store the locally cached copy.

The *Value Attribute* of the *DataTypeDictionary* containing the type descriptions is a ByteString whose formatting is defined by the *DataTypeSystem*. For the "XML Schema" *DataTypeSystem*, the ByteString contains a valid XML Schema document. For the "OPC Binary" *DataTypeSystem*, the ByteString contains a string that is a valid XML document. The server shall ensure that any change to the contents of the ByteString is matched with a corresponding change to the *DataTypeVersion Property*. In other words, the client may safely use a cached copy of the *DataTypeDictionary*, as long as the *DataTypeVersion* remains the same.

DataTypeDictionaries are complex *Variables* which expose their *DataTypeDescriptions* as *Variables* using *HasComponent References.* A *DataTypeDescription* provides the information necessary to find the formal description of a *DataType* within the *DataTypeDictionary*. The *Value* of a *DataTypeDescription* depends on the *DataTypeSystem* of the *DataTypeDictionary*. When using "OPC Binary" dictionaries the *Value* shall be the name of the *TypeDescription*. When using "XML Schema" dictionaries the Value shall be an Xpath expression XPATH which points to an XML element in the schema document.

Like *DataTypeDictionaries* each *DataTypeDescription* provides the *Property DataTypeVersion* indicating whether the type description of the *DataType* has changed. Changes to the *DataTypeVersion* may impact the operation of *Subscriptions*. If the *DataTypeVersion* changes for a *Variable* that is being monitored for a *Subscription* and that uses this *DataTypeDescription*, then the next data change *Notification* sent for the *Variable* will contain a status that indicates the change in the *DataTypeDescription*.

DataTypeEncoding Objects of the *DataTypes* reference their *DataTypeDescriptions* of the *DataTypeDictionaries* using *HasDescription* References. However, servers are not required to provide the inverse *References* that relate the *DataTypeDescriptions* back to the *DataTypeEncoding Objects*. If a *DataType Node* is exposed in the *AddressSpace*, it shall provide its *DataTypeEncodings* and if a *DataTypeDictionary* is exposed, it should expose all its *DataTypeDescriptions*. Both of these *References* shall be bi-directional.

The *VariableTypes DataTypeDictionaryType* and *DataTypeDescriptionType* and the *ObjectTypes DataTypeSystemType* and *DataTypeEncodingType* are formally defined in IEC 62541-5.

Figure 11 provides an example how *DataTypes* are modelled in the *AddressSpace*.

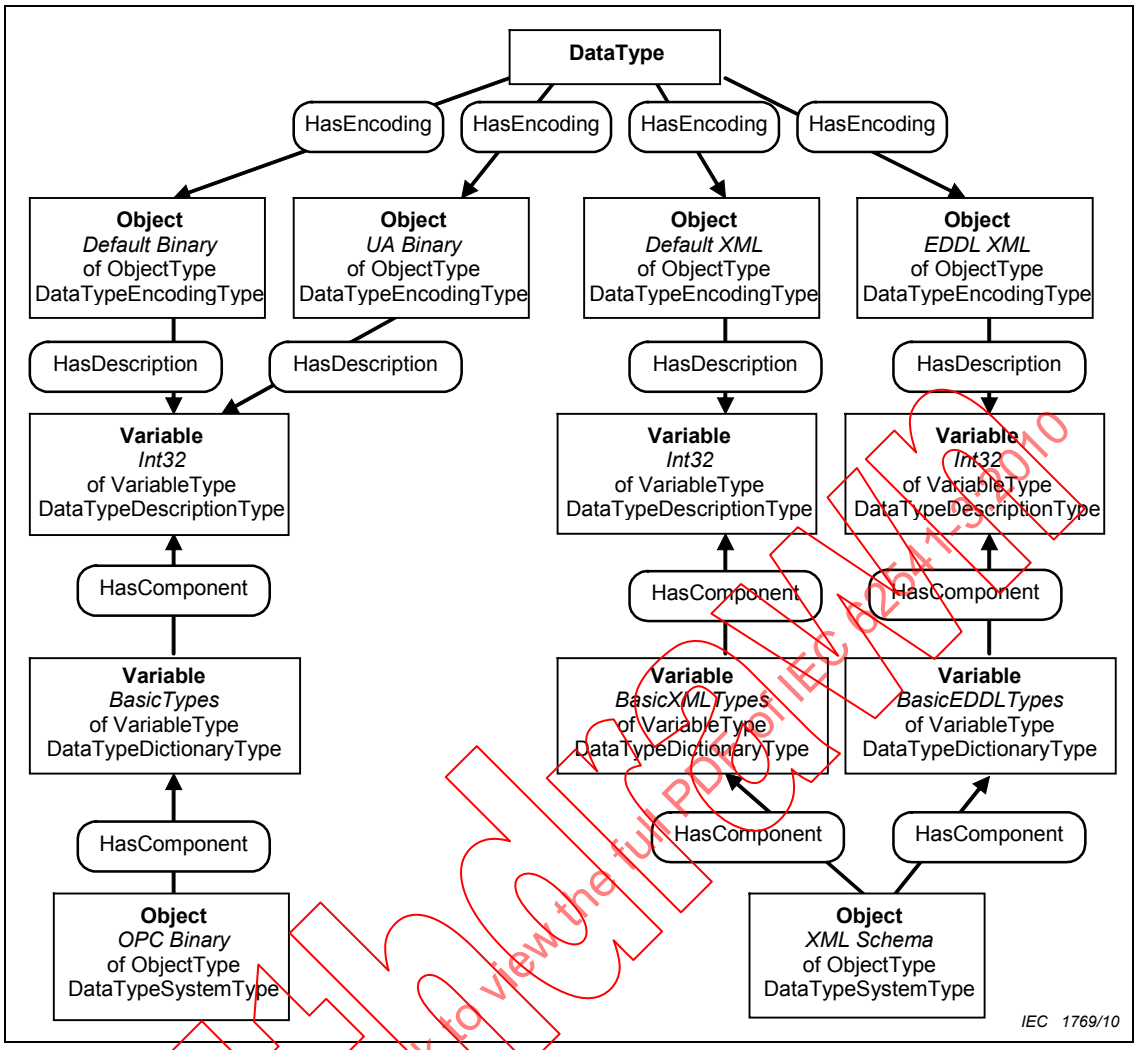


Figure 11 – Example of DataType Modelling

In some scenarios an OPC UA server may have resource limitations which make it impractical to expose large *DataTypeDictionaries*. In these scenarios the server may be able to provide access to descriptions for individual DataTypes even if the entire dictionary cannot be read. For this reason, this standard defines a *Property* for the *DataTypeDescription* called *DictionaryFragment* (see 5.6.2). This *Property* is a ByteString that contains a subset of the *DataTypeDictionary* which describes the format of the *DataType* associated with the *DataTypeDescription*. Thus, the server splits the large *DataTypeDictionary* into several small parts and clients can access without affecting the overall system performance.

However, servers should provide the whole *DataTypeDictionary* at once if this is possible. It is typically more efficient to read the whole *DataTypeDictionary* at once instead of reading individual parts.

5.9 Summary of Attributes of the NodeClasses

Table 12 summarises all *Attributes* defined in this document and points out which *NodeClasses* use them either optional (O) or mandatory (M).

Table 12 – Overview about Attributes

Attribute	Variable	Variable Type	Object	Object Type	Reference Type	DataType	Method	View
AccessLevel	M							
ArrayDimensions	O	O						
BrowseName	M	M	M	M	M	M	M	M
ContainsNoLoops								M
DataType	M	M						
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
Symmetric					M			
UserAccessLevel	M							
UserExecutable							M	
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 following clauses 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*.

6.2.2 Instances without ModellingRules

If no *ModellingRule* exists, then the *Node* is neither considered for instantiation of a type nor considered 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 *hierarchical References* in forward direction.

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 *hierarchical References* in forward direction 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 modify the value of an entry in the array from 0 to a different value. All other values in the array shall remain the same.

6.3 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 following subclauses specify 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 following recursively the inverse *HasSubtype References* from the subtype. This collection of *InstanceDeclarations* is called the fully-inherited *InstanceDeclarationHierarchy* of a subtype.

The following subclauses define 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 straight forward and can be illustrated with the example shown in Figure 12 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*.

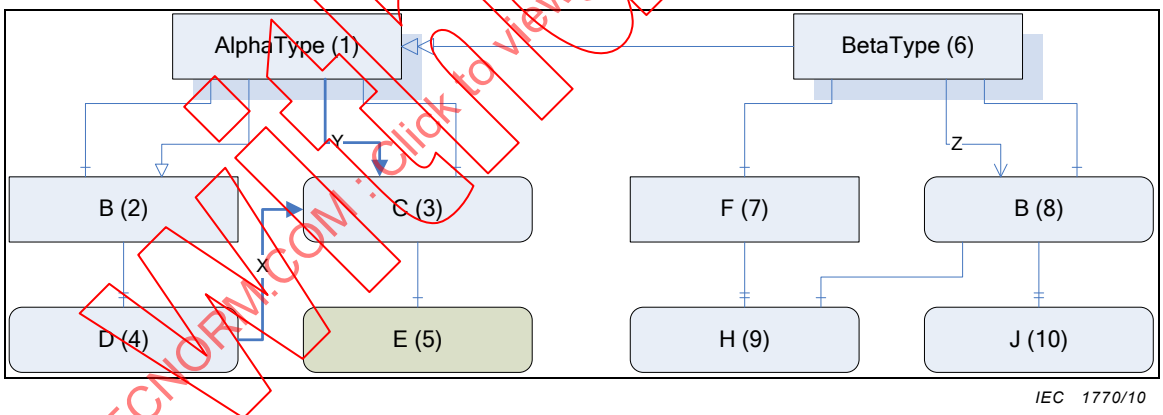


Figure 12 – Subtyping TypeDefinitionNodes

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 13 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 13 – 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 14 where the entries added from "AlphaType" are shaded with grey.

Table 14 – 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" is not and was added.

The *Nodes* and *References* defined in Table 14 can be used to create the fully-inherited *InstanceDeclarationHierarchy* shown in Figure 13. The fully-inherited *InstanceDeclarationHierarchy* contains all necessary information about a *TypeDefinitionNode* regarding its complex structure without needing any additional information from its supertypes.

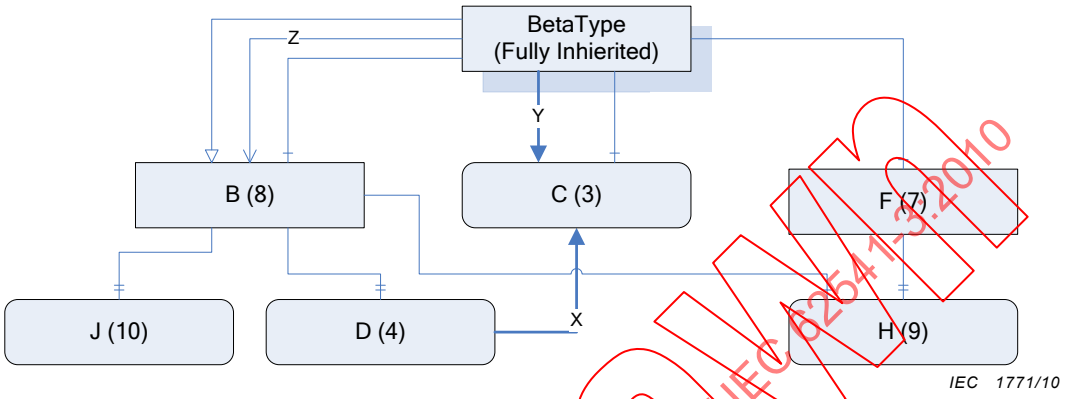


Figure 13 – The Fully-Inherited InstanceDeclarationHierarchy for BetaType

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 13, has to be overridden, then the directly referenced *InstanceDeclarations* that includes "J", in that case "B", has to be overridden first and "J" can then 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* has to 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 *InstanceDeclarations* in the hierarchy of the *TypeDefinitionNode*. The *InstanceDeclaration* with the same *BrowsePath* is the called *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, the *Attribute* values of the *InstanceDeclarations* are used as initial values.

Figure 14 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*.

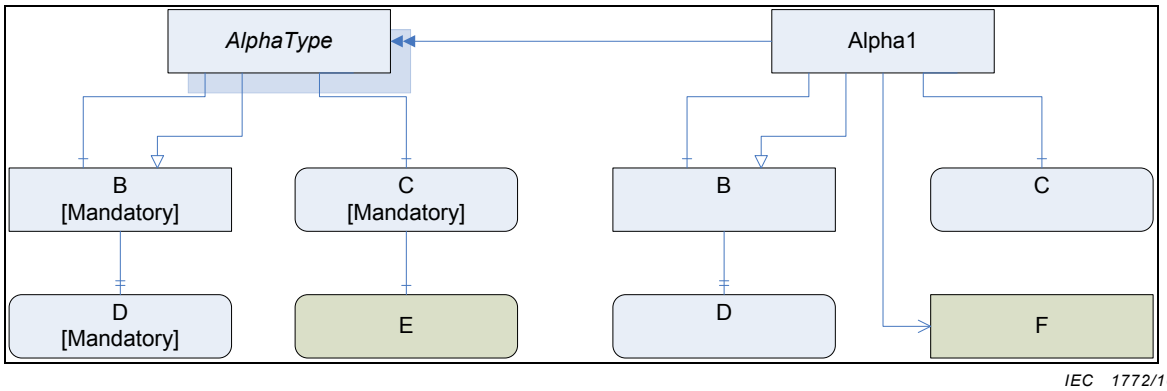


Figure 14 – An Instance and its TypeDefinitionNode

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 *Node*s which are in the hierarchy of the instance. It shall pass the *NodeId* of the instance and the *BrowsePath* of the child *Node*s based on the *TypeDefinitionNode* to the *TranslateBrowsePathsToNodeIds* service (see IEC 62541-4). This *Service* returns the *NodeId* for each of the child *Node*s. 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 15. 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.

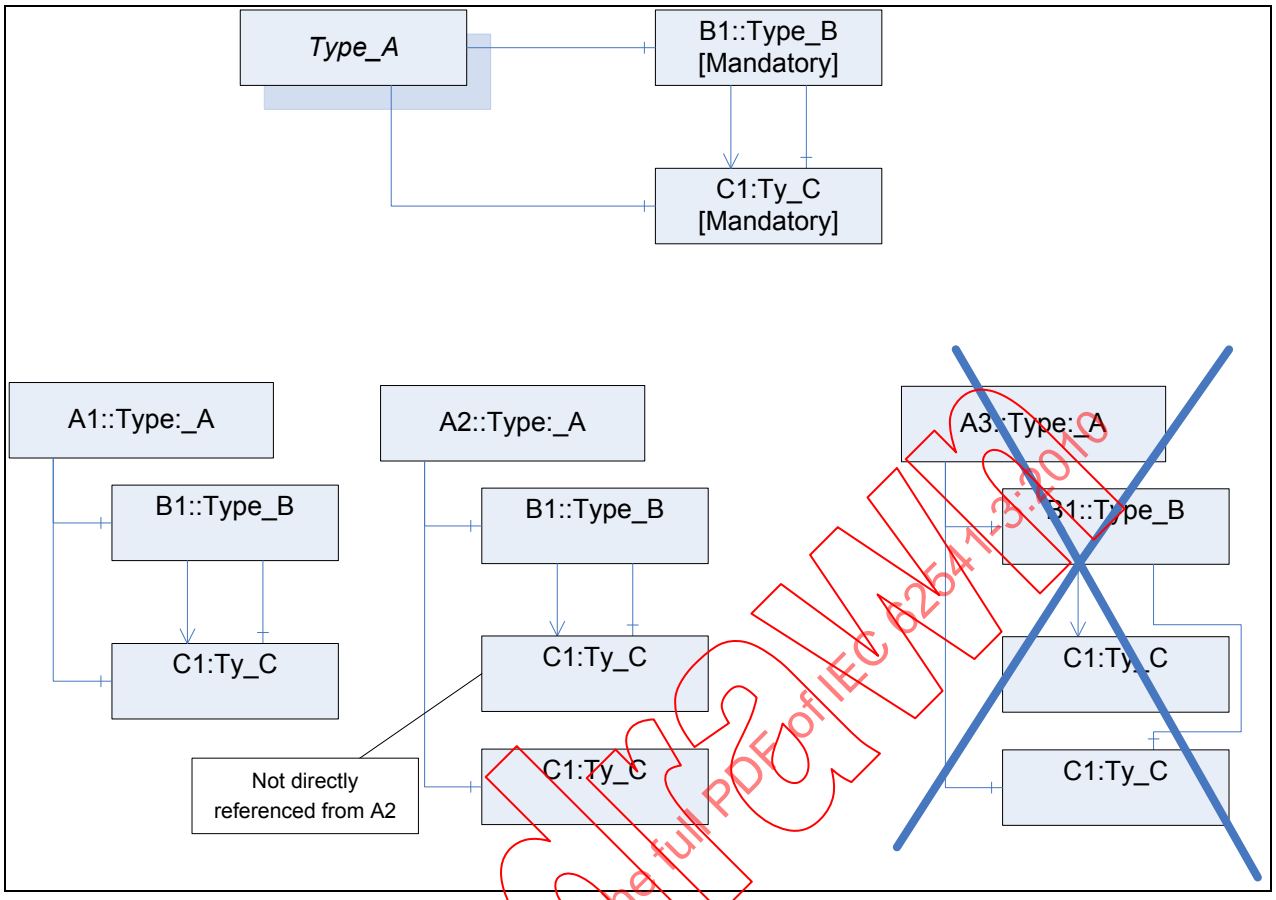


Figure 15 – Example for several References between InstanceDeclarations

6.4.4 ModellingRules

6.4.4.1 General

For a definition of *ModellingRules*, see 6.4.4.5. Other parts of this series of standards 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 standard 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 version of this standard only specifies one *Property* for *ModellingRules*. Future versions 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*. 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

6.4.4.2.1 NamingRule

NamingRule is a mandatory *Property* of a *ModellingRule*. It specifies the purpose of an *InstanceDeclaration*. Each *InstanceDeclaration* references to 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 the TranslateBrowsePathsToNodeIds *service* (see IEC 62541-4) returns this *Node* as 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* it 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*.

6.4.4.2.2

Void.

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 15 specifies the allowed changes on the *Properties* when exchanging the *ModellingRules* in the subtype.

Table 15 – 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 16 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 a different *ModellingRule* then B of Type_A.

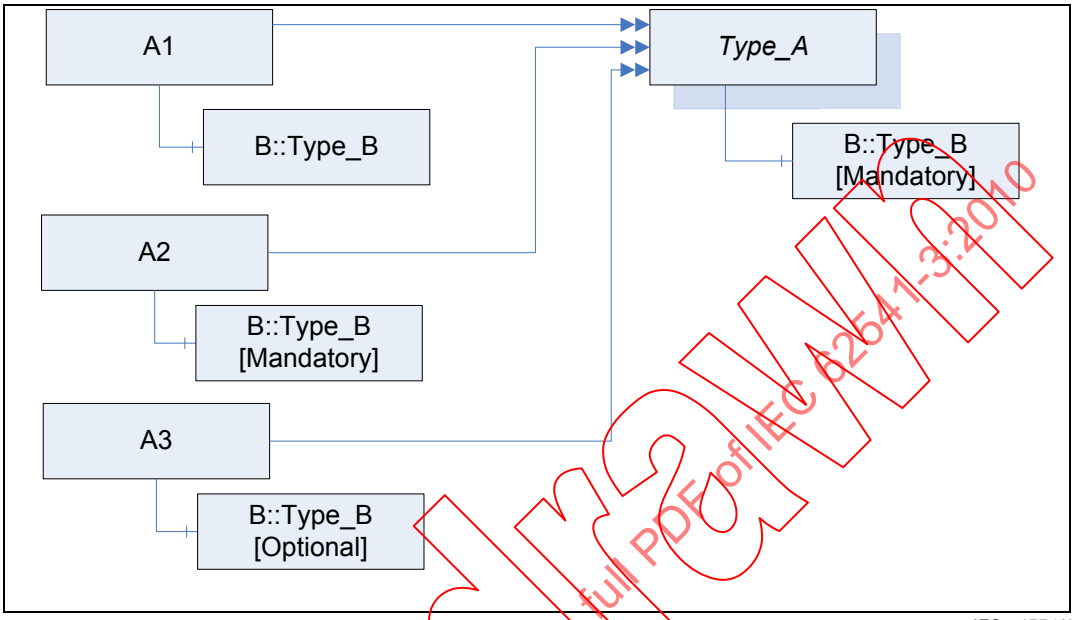


Figure 16 – Example on changing instances based on InstanceDeclarations

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 17 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 *ModellingRules* of the *InstanceDeclaration* A1 is changed (lower part of the Figure). A1 has become the *NamingRule* of *Mandatory* (changed from *Optional*).

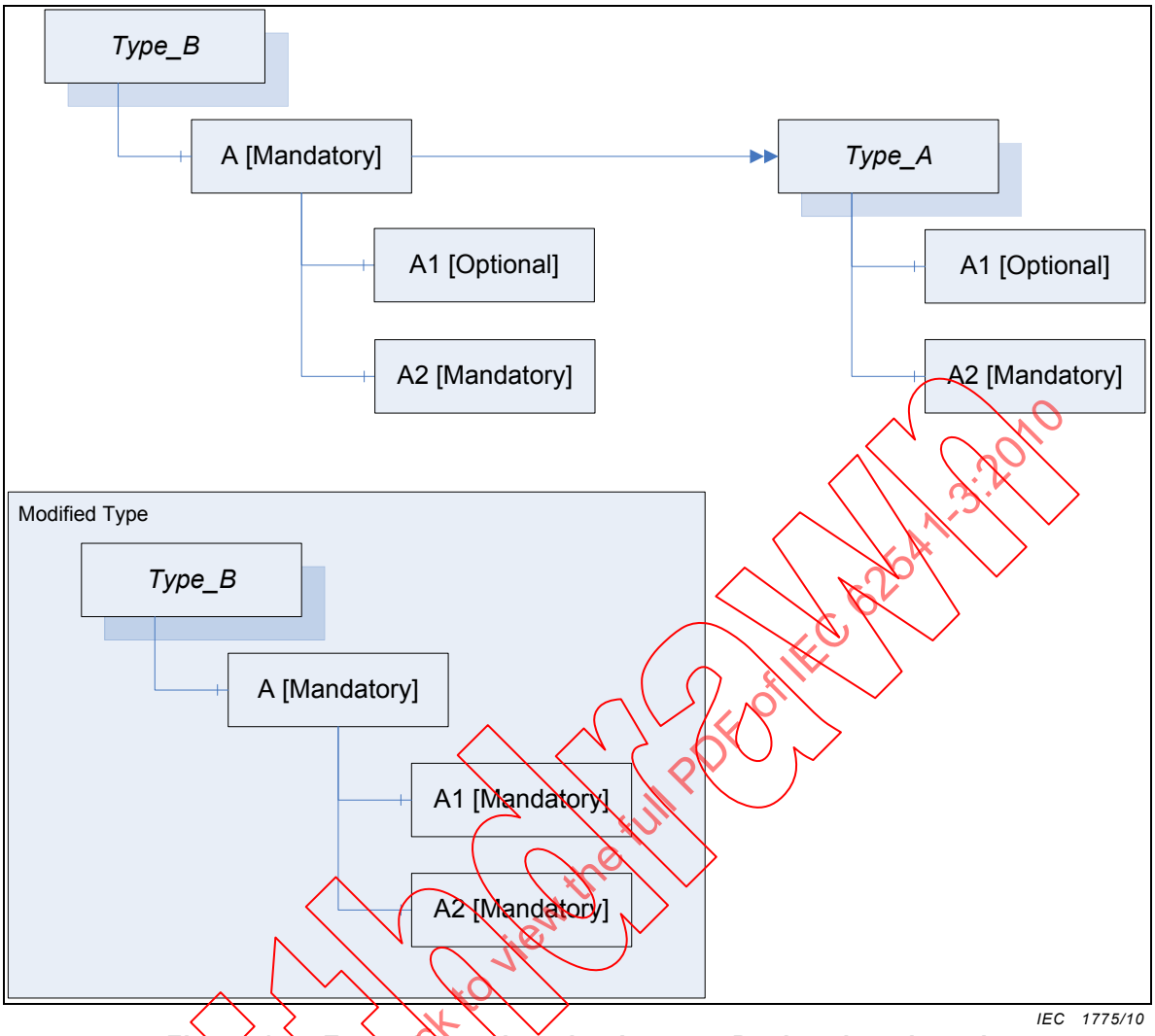


Figure 17 – Example on changing InstanceDeclarations based on an InstanceDeclaration

6.4.4.5 Standard ModellingRules

6.4.4.5.1 Titles of Standard ModellingRules

The following subclauses define *ModellingRules*. In Table 16 the *Properties* of those *ModellingRules* are summarized.

Table 16 – Properties of ModellingRules

Title	NamingRule
Mandatory	Mandatory
Optional	Optional
ExposesItsArray	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 18 illustrates the example.

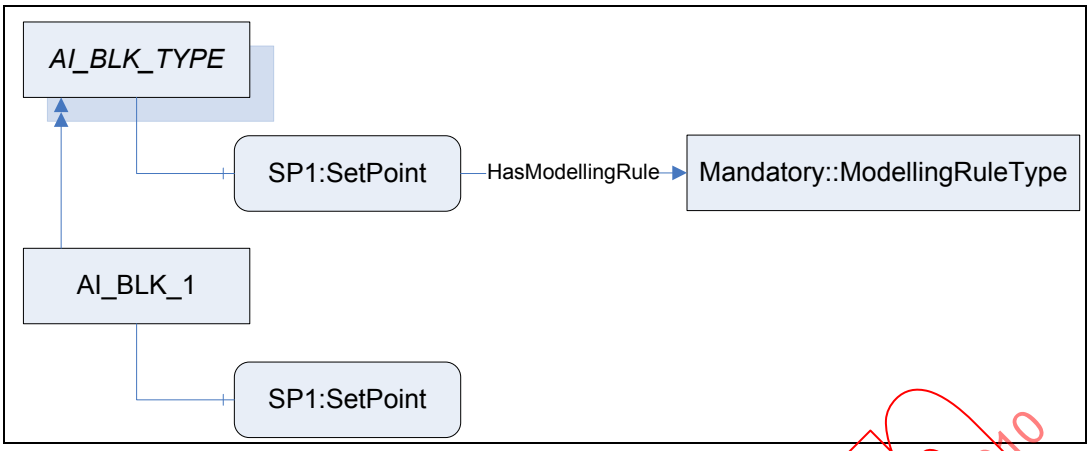


Figure 18 – Use of the Standard ModellingRule New

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 19 an example using the *ModellingRules Optional* and *Mandatory* is shown. The example contains an *ObjectType* A_Type 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 A13, there could be a reference from on F but not from the other, or from both or none of them.

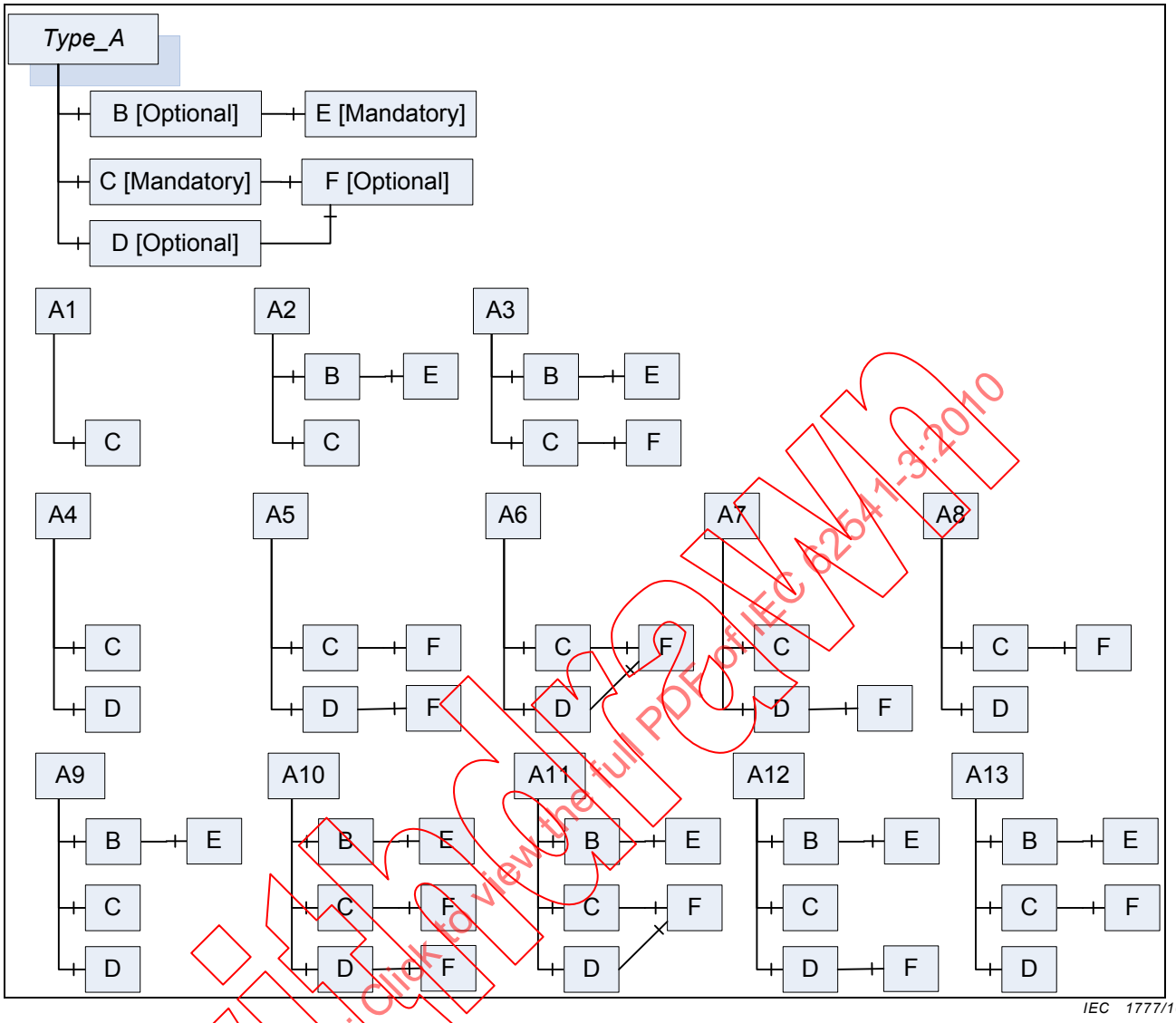


Figure 19 – Example using the Standard ModellingRules Optional and Mandatory

6.4.4.5.4 ExposesItsArray

The *ExposesItsArray ModellingRule* exposes a special semantic on *VariableTypes* having a single- or multidimensional array as 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 data type.

The *Variable* A having this *ModellingRule* shall be referenced by a *hierarchical Reference* in forward direction from a *VariableType* B. B shall have a *ValueRank* value equal 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 *hierarchical References* in forward direction between A and B, then all instances based on B shall be referenced with all those *References*.

Figure 20 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 is 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 providing that information.

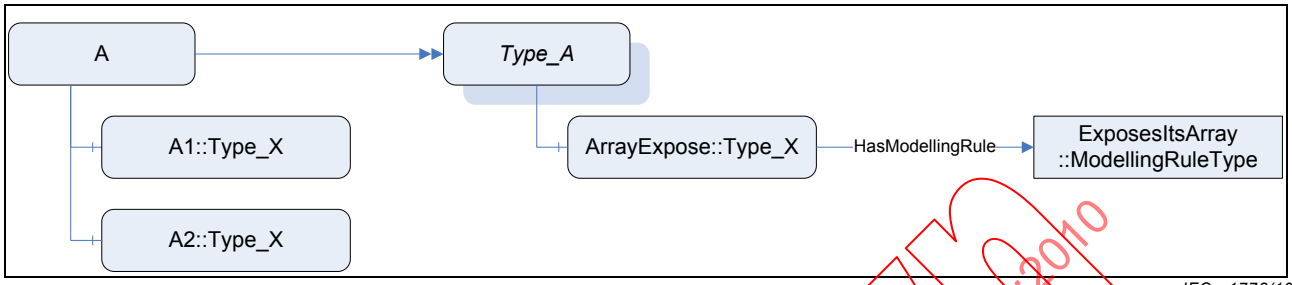


Figure 20 – Example on using ExposesItsArray

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

Figure 21 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.

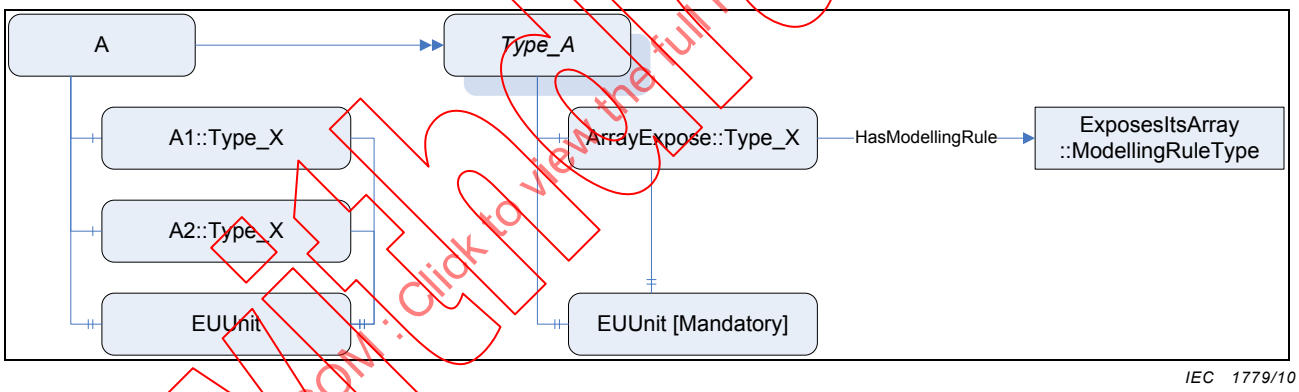


Figure 21 – Complex example on using ExposesItsArray

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 have to 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 has to add the *Property* to all instances of the type or adding the *Property* on the type has to be rejected.

6.6 ModelParent

Each *Object*, *Variable* and *Method* may be referenced by several *hierarchical References*. To indicate the scope of such a *Node* 'A' it can expose its *ModelParent*. The *ModelParent* of 'A' is the scope where 'A' is defined. When a client intends to change 'A' it can use the *ModelParent* to determine whether it has the correct scope.

For example, a *TypeDefinitionNode* 'Type_A' may have an *InstanceDeclaration* called 'Icon' which is shared by all instances. Thus the *ModelParent* of 'Icon' is 'Type_A'. A client may browse to an instance 'A' that is of 'Type_A' to change the value of 'Icon' for 'A'. 'A' is

referencing the shared *InstanceDeclaration* 'Icon' and by identifying its *ModelParent* a client can figure out that the scope of the *Node* was not 'A' but 'Type_A'. Thus the client may not change the value of the *Node* but rather create a copy, reference the copy instead of the *InstanceDeclaration* and change the value of the copy.

To identify a *ModelParent* the *ReferenceType HasModelParent* is used referencing from the *Object*, *Variable* or *Method* to the *Node* representing the *ModelParent*. The *ModelParent* shall reference the *Object*, *Variable* or *Method* with a *hierarchical Reference* in forward direction.

HasModelParent References shall be provided for all *ModellingRules* defined in this document. It is not required to expose the *HasModelParent References* for other *ModellingRules*, but it is recommended. It is allowed to provide a *HasModelParent Reference* on *Objects*, *Variables*, and *Methods* having no *HasModellingRule* reference. However, this is not required and my not always be possible since there may be no clear defined *ModelParent*.

Figure 22 illustrates the *HasModelParent* relationships for a *TypeDefinitionNode* and two instances. In this example, the model parent of *Node* "C" is the *TypeDefinitionNode* and all instances reference it with hierarchical references.

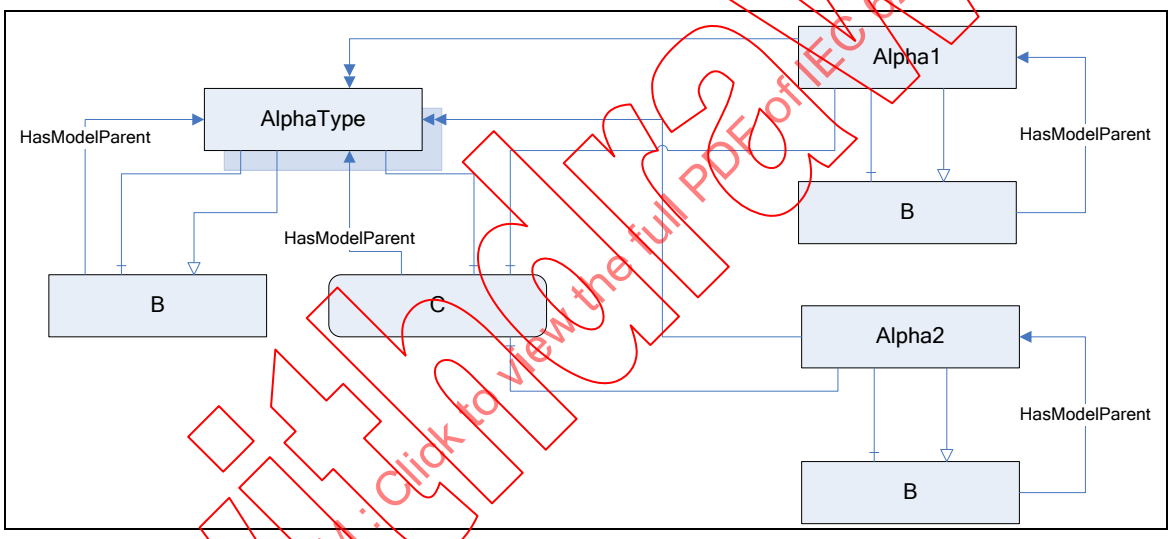


Figure 22 – Example on ModelParents

7 Standard ReferenceTypes

7.1 General

This standard defines *ReferenceTypes* as an inherent part of the OPC UA Address Space Model. Figure 23 informally describes the hierarchy of these *ReferenceTypes*. Other parts of this series of standards may specify additional *ReferenceTypes*. The following subclauses define the *ReferenceTypes*. IEC 62541-5 defines their representation in the *AddressSpace*.

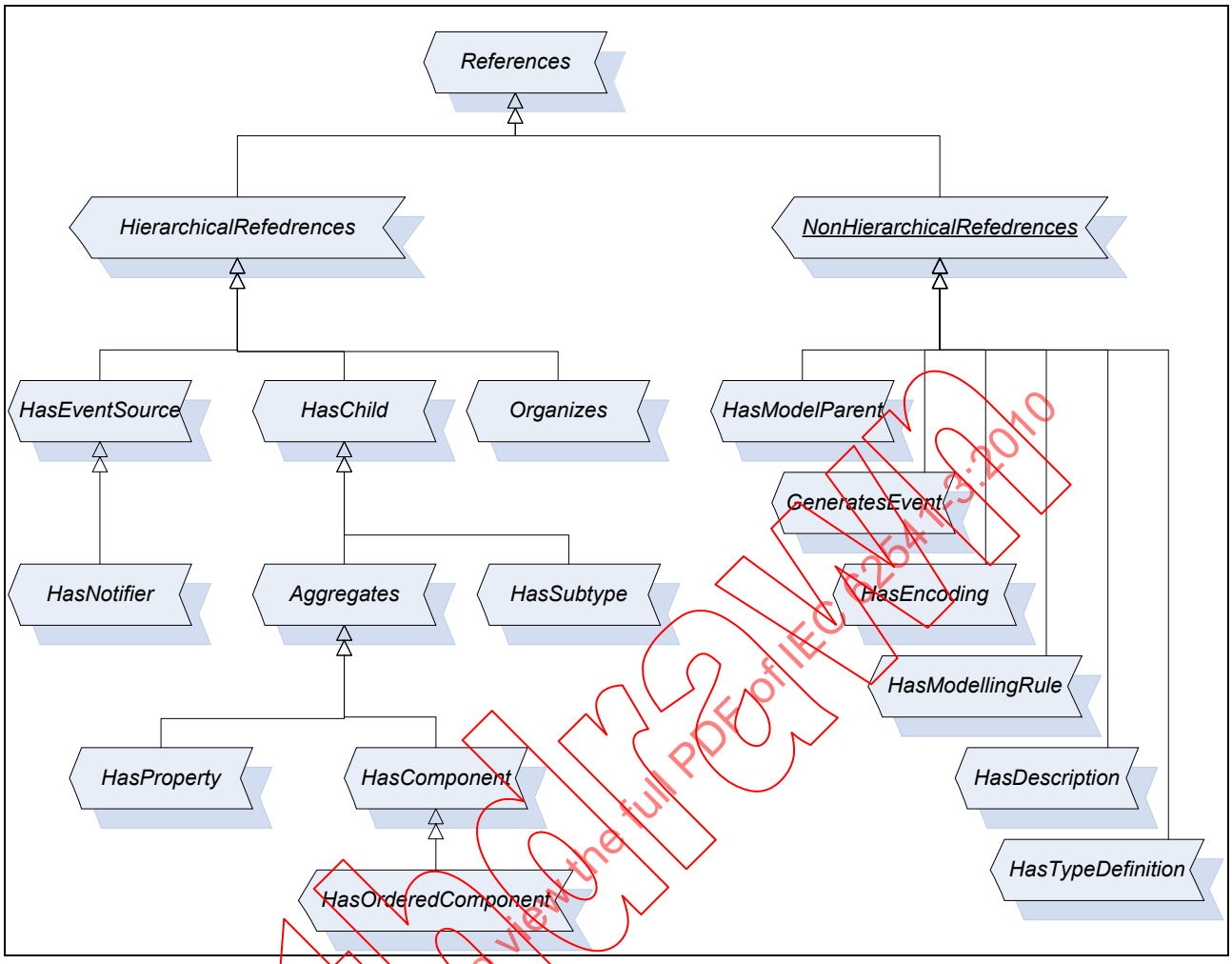


Figure 23 – Standard ReferenceType Hierarchy

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 hierarchy-like way. *HierarchicalReferences* does not forbid loops. For example, starting from *Node* “A” and following *HierarchicalReferences* may lead 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* shall never be able 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* as well as complex *Variables* 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 62541-5 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, which semantic is specified in 5.3.3.3; a *DataType* hierarchy as specified in 5.8.3, as well as other subtype hierarchies as 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 organise *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* 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 HasModelParent ReferenceType

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

The semantic of this *ReferenceType* is to expose the *ModelParent* of in *Object, Variable or Method*. The *ModelParent* mechanisms are described in 6.6.

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

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

7.14 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*, the *TargetNode* shall be an *ObjectType*; if the *SourceNode* is a *Variable*, the *TargetNode* shall be a *VariableType*.

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

7.15 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 *DataType*.

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

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

7.16 HasDescription ReferenceType

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

The semantic of this *ReferenceType* is to reference the *DataTypeDescription* of a *DataTypeEncoding*.

The *SourceNode* of *References* of this type shall be an *Object* of the *ObjectType DataTypeEncodingType* or one of its subtypes.

The *TargetNode* of this *ReferenceType* shall be a *Variable* of the *VariableType DataTypeDescriptionType* or one of its subtypes (see 5.8.4).

7.17 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.

7.18 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* have to 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.

7.19 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 *Event* from its source 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 its subtypes shall never be able to return to "A". But it is permitted that, following the *References*, there may be more than one path leading to another *Node* "B".

7.20 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 7.18.

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*, 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 24. In this example an unfiltered *Event* subscription directed to the "Level Sensor" *Object* will provide the *Event* sources "Low Level" and "High Level" 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".

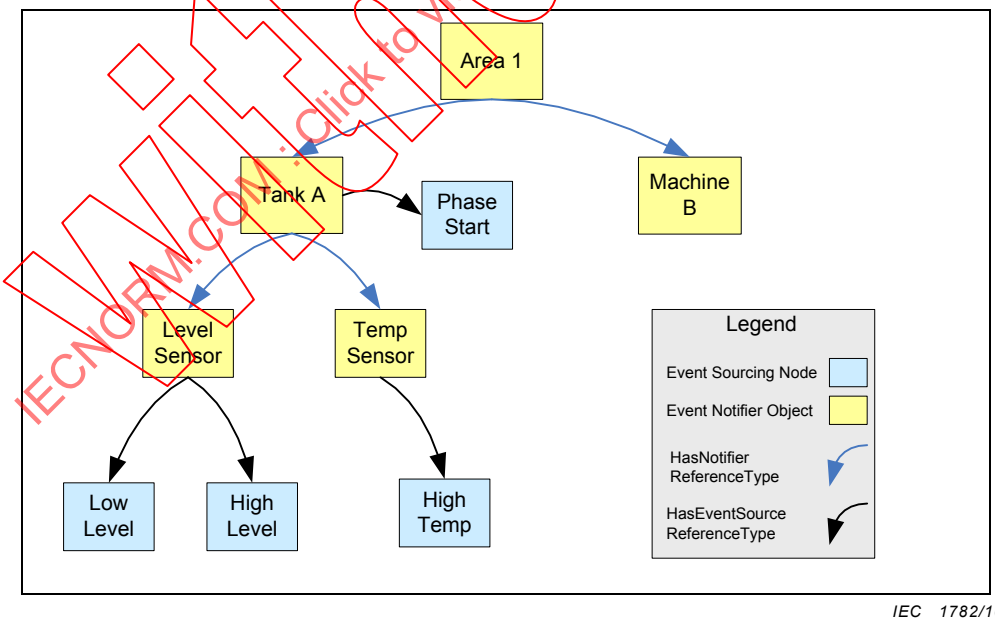


Figure 24 – Event Reference Example

A second example of a more complex organization of *Event References* is represented in Figure 25. 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".

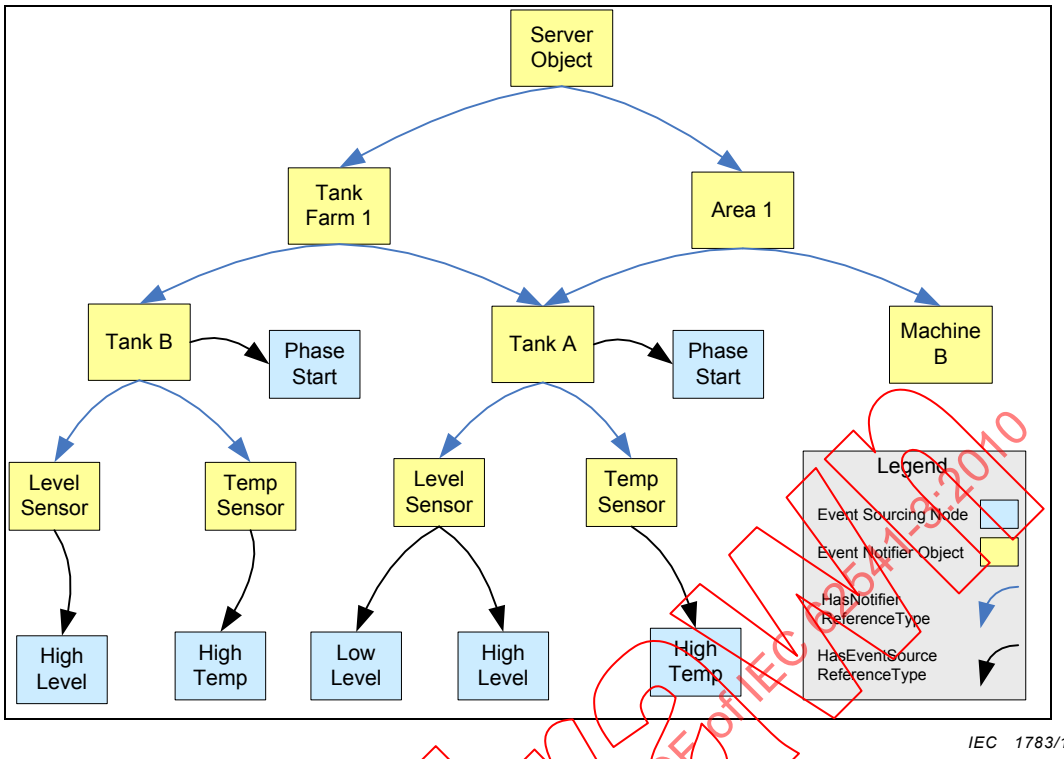


Figure 25 – Complex Event Reference Example

8 Standard DataTypes

8.1 General

The following subclauses define *DataTypes*. Their representation in the *AddressSpace* and the *DataType* hierarchy is specified in IEC 62541-5. Other parts of this series of standards 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 17.

Table 17 – 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.

8.2.3 IdentifierType

The IdentifierType element identifies the type of the *NodeId*, its format and its scope. Its values are defined in Table 18.

Table 18 – 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.

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 4096 characters. Identifier values of IdType OPAQUE_3 are restricted to 4096 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 summarised in Table 19.

Table 19 – 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 20. The name part of the *QualifiedName* is restricted to 512 characters.

Table 20 – 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 3066. 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 3066 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 21 shows examples of OPC UA *LocaleIds*. Clients and servers always provide *LocaleIds* that explicitly identify the language and the country/region.

Table 21 –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 22.

Table 22 – LocalizedText Definition

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

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 23.

Table 23 – 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.
arrayDimensions	UInt32[]	Specifies the length of each dimension for an array dataType. It is intended to describe the capability of the dataType, not the current size. The number of elements shall be equal to the value of the *valueRank*. Must be null if *valueRank* <= 0. A value of 0 for an individual dimension indicates that the dimension has a variable length.
description	LocalizedText	A localised 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 IEEE 754-1985 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 a special handling for the encoding as defined in IEC 62541-6. All enumeration *DataTypes* have to inherit from this *DataType*.

8.15 Float

This *Built-in DataType* defines a value that adheres to the IEEE 754-1985 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 18. 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. JPG is defined in ISO/IEC 10918-1.

8.23 ImagePNG

This *Simple DataType* defines a *ByteString* representing an image in PNG format. PNG is defined in ISO/IEC 15948.

8.24 Integer

This abstract *DataType* defines an integer which 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 23.

Table 24 – TimeZoneDataType Definition

Name	Type	Description
TimeZoneDataType	structure	
offset	Uint16	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 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 6.4.4.2.1). Its values are defined in Table 25.

Table 25 – 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 26.

Table 26 – 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 (0x00 - 0x08, 0x0E-0x1F or 0x7F).

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 a special handling for the encoding as defined in IEC 62541-6. All *Structured DataTypes* have to inherit from this *DataType* if they are not defined as primitives in this standard (like *NodeId* defined in 8.2, a *NodeId* is structured but treated in a special way as defined in IEC 62541-6).

8.34 UInteger

This abstract *DataType* defines an unsigned integer which 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. IEC 62541-6 defines details about this *DataType*.

8.39 XmlElement

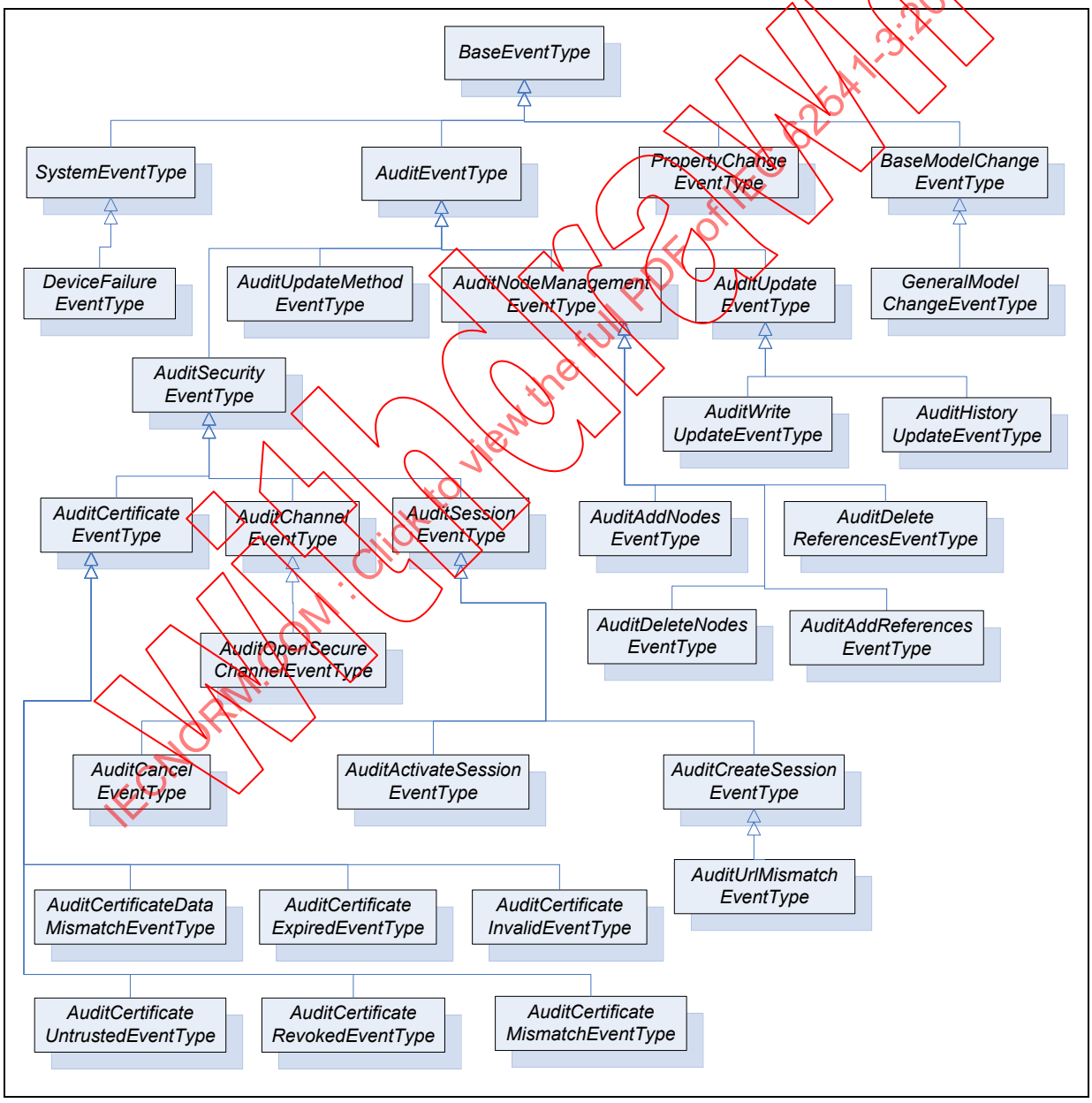
This *Built-in DataType* is used to define XML elements. IEC 62541-6 defines details about this *DataType*.

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*.

9 Standard EventTypes

9.1 General

The following subclauses define *EventTypes*. Their representation in the *AddressSpace* is specified in IEC 62541-5. Other parts of this series of standards may specify additional *EventTypes*. Figure 26 informally describes the hierarchy of these *EventTypes*.



IEC 1784/10

Figure 26 – 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 generated as a result of some *Event* that occurs within the server or by a system that the server is representing.

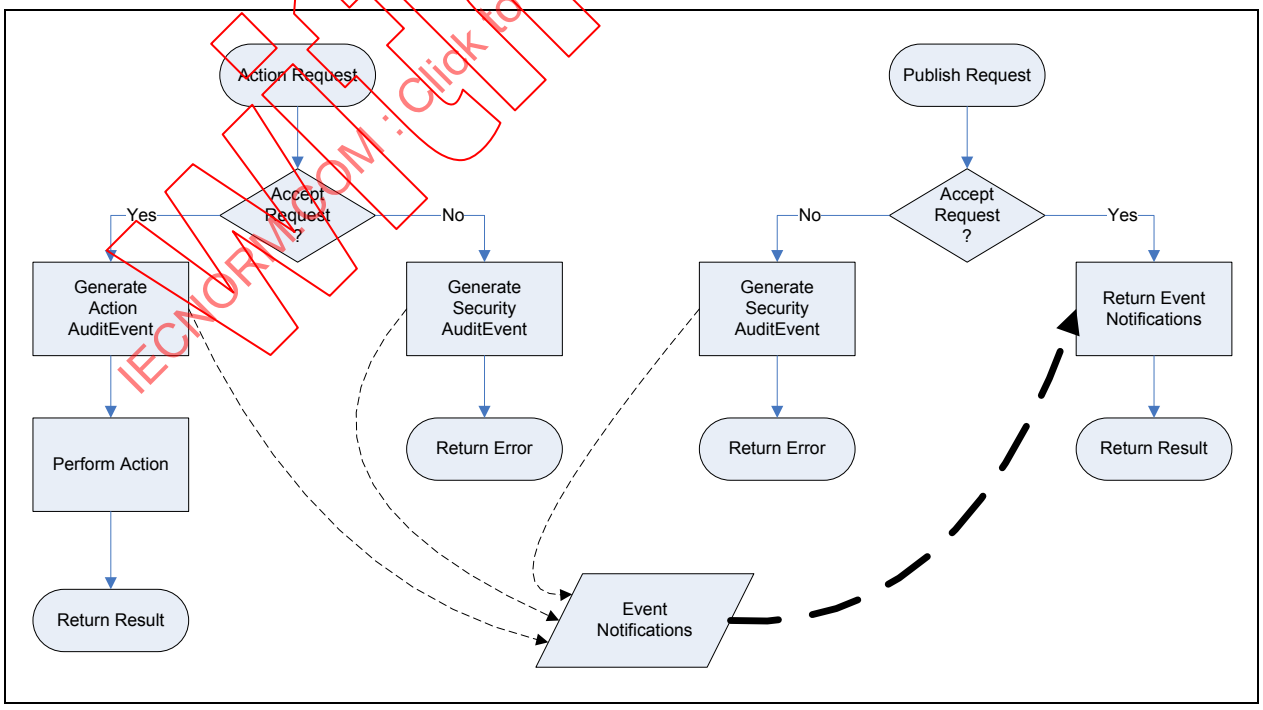
9.4 AuditEventType

AuditEvents 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 27 illustrates the defined behaviour of an OPC UA server in response to an auditable action request. If the action is accepted, 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.



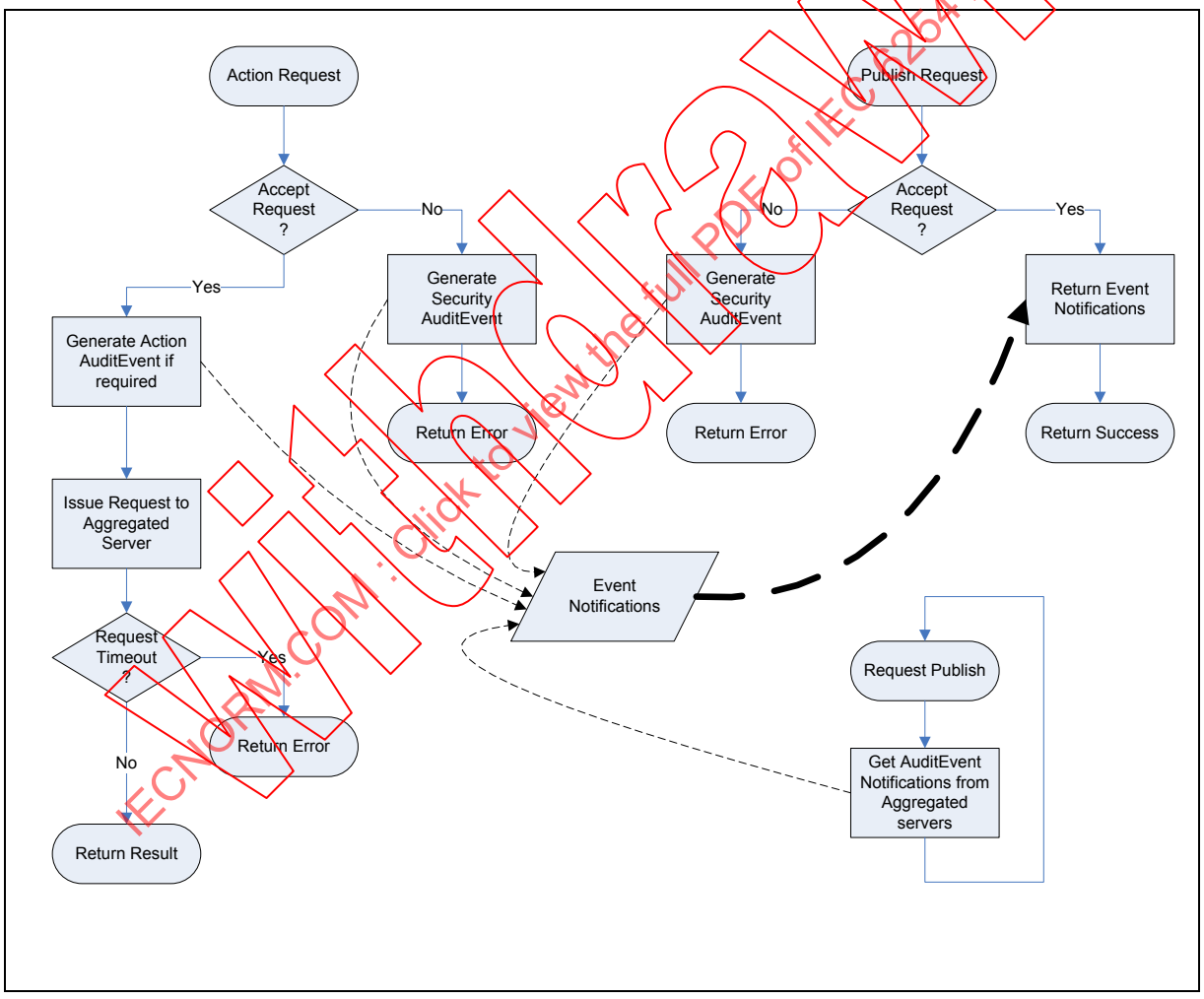
IEC 1785/10

Figure 27 – Audit Behaviour of a Server

Figure 28 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*, 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.



IEC 1786/10

Figure 28 – Audit Behaviour of an Aggregating Server

9.5 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.

9.6 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.

9.7 AuditOpenSecureChannelEventType

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

9.8 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.

9.9 AuditCreateSessionEventType

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

9.10 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).

9.11 AuditActivateSessionEventType

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

9.12 AuditCancelEventType

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

9.13 AuditCertificateEventType

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

9.14 AuditCertificateDataMismatchEventType

This is a subtype of AuditCertificateEventType and is used only for categorization of Certificate related *Events*. This type follows all behaviour 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 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.

9.15 AuditCertificateExpiredEventType

This is a subtype of AuditCertificateEventType and is used only for categorization of Certificate related *Events*. This type follows all behaviour of its parent type. This *AuditEvent* is generated if the current time is not after the start of the validity period and before the end.

9.16 AuditCertificateInvalidEventType

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

9.17 AuditCertificateUntrustedEventType

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

9.18 AuditCertificateRevokedEventType

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

9.19 AuditCertificateMismatchEventType

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

9.20 AuditNodeManagementEventType

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

9.21 AuditAddNodesEventType

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

9.22 AuditDeleteNodesEventType

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

9.23 AuditAddReferencesEventType

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

9.24 AuditDeleteReferencesEventType

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

9.25 AuditUpdateEventType

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

9.26 AuditWriteUpdateEventType

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

9.27 AuditHistoryUpdateEventType

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

9.28 AuditUpdateMethodEventType

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

9.29 DeviceFailureEventType

A *DeviceFailureEvent* indicates a failure in a device of the underlying system.

9.30 ModelChangeEvents

9.30.1 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.

9.30.2 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.

9.30.3 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 it has to use the filter mechanisms of the *Event* subscription filtering only for the default *View* and suppress the *ModelChangeEvents* having other *Views* as context.

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

9.30.4 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.

9.30.5 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.

9.30.6 GeneralModelChangeEventType

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

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

9.30.7 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*.

9.31 SemanticChangeEventType

9.31.1 General

SemanticChangeEvents 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).

9.31.2 ViewVersion and NodeVersion Properties

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

9.31.3 Views

SemanticChangeEvents are handled in the context of a *View* the same way as *ModelChangeEvents*. This is defined in 9.30.3.

9.31.4 Event Compression

SemanticChangeEvents can be compressed the same way as *ModelChangeEvents*. This is defined in 9.30.4.

Annex A
(informative)

How to use the Address Space Model

A.1 Overview

This annex points out some general considerations how the Address Space Model can be used. This annex is for information only, that is, each server vendor can model its data in the appropriated way that fits to 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 on 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 how to model the data. However, the following subclauses will give some general consideration about the different concepts of OPC UA to model data and when they should be used and when not.

Annex A of IEC 62541-5 gives an overview over 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 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 the object should be exposed as an object in the client's GUI or just as a value. Whenever a modeller is in doubt 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*[2] and should be used when there are several *Variables* having the same semantic (e.g. set point). It is not needed to define a *VariableType* just reflecting the *DataType* of the *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 syntactic 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’s not clear what concept to use based on the semantic described in Clause 4, the different syntax can help. The following points identify that it has to be a *DataVariable*:

- If it’s 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’s a component of a complex *Variable* exposing a part of the value of the complex *Variable*.

A.4.3 Many Variables and / or complex DataTypes

When complex 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 complex *DataType* and a simple *Variable* using this *DataType*.

c) Create a complex *DataType* and a complex *Variable* using this *DataType* and also exposing the complex 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 those data without knowledge of user-defined *DataTypes*; and the client can access individual parts of the complex 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 are accessed in a transaction context and the complex *DataType* can be constructed in a way that the server does not have to convert the data and can pass them 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 *DataTypeDescription* 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.

---

2 *VariableTypes* other then the *PropertyType* which is used for all *Properties*.

The third approach combines both other 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, the third approach is suitable. However, the server might not always have the knowledge to interpret the complex 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 needed 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 appropriated 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 needed behaviour for instantiation and subtyping.

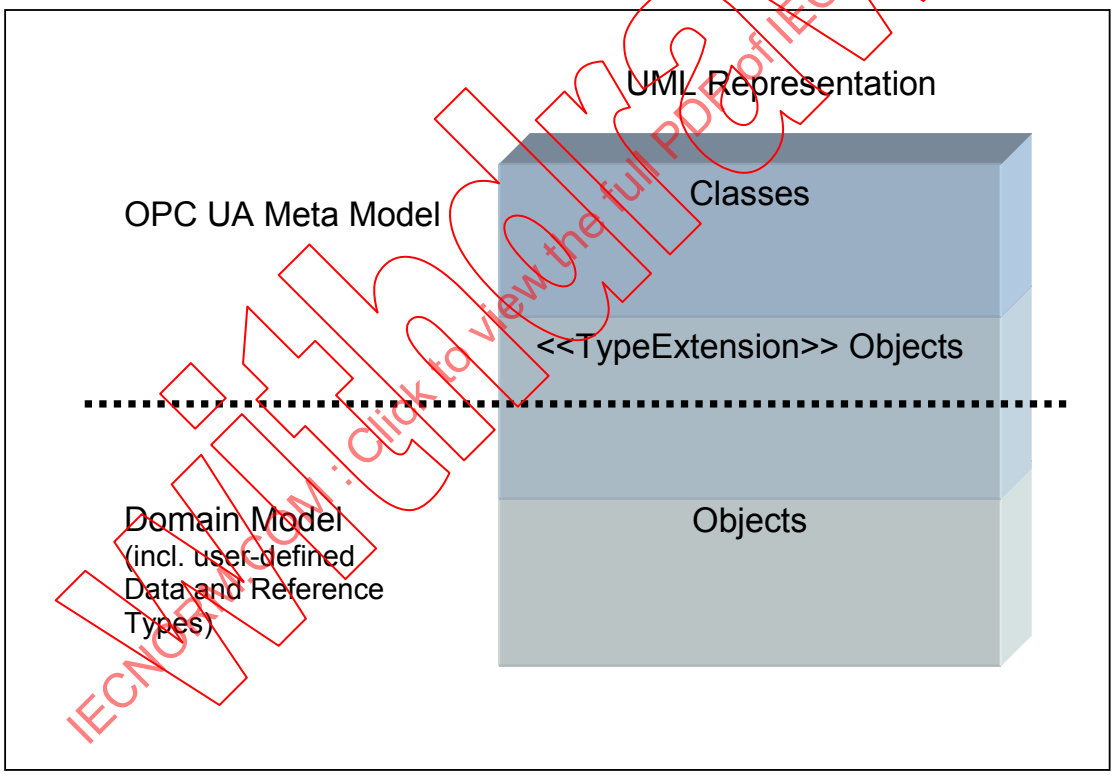
Annex B
(informative)

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 organise the *AddressSpace* and to provide information about the server as specified in IEC 62541-5.



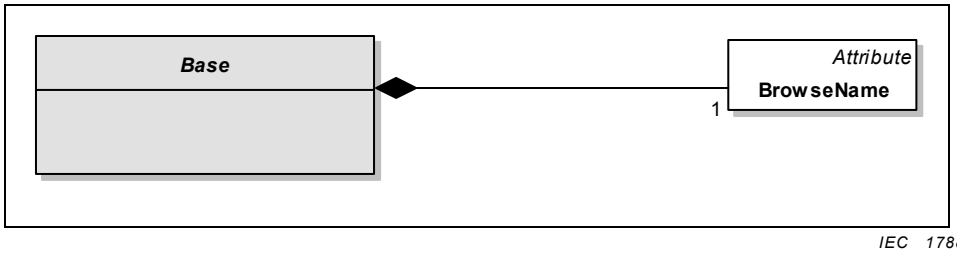


Figure B.2 – Notation (I)

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*).

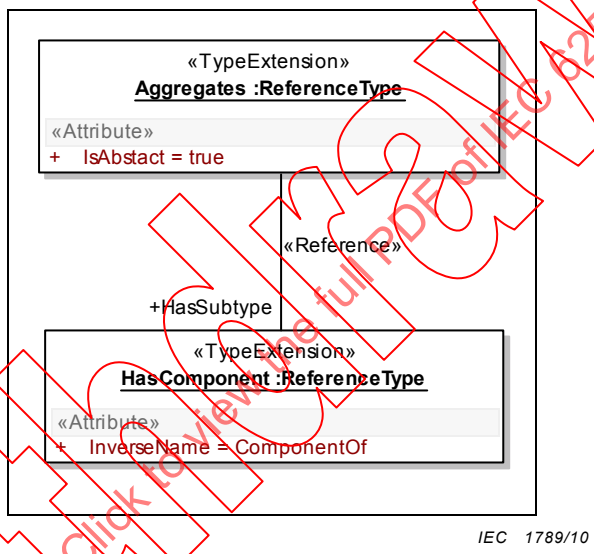


Figure B.3 – Notation (II)

OPC *References* are represented as UML associations marked with the stereotype <<Reference>>. If a particular *ReferenceType* is used, its name is used as 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 *SubclassOf*). 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 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 grey (e.g. *Base* in Figure B.2). Only those classes have all 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

Remark: Other parts of this series of standards can extend the OPC UA Meta Model by adding *Attributes* and defining new *ReferenceTypes*.

B.3.1 Base

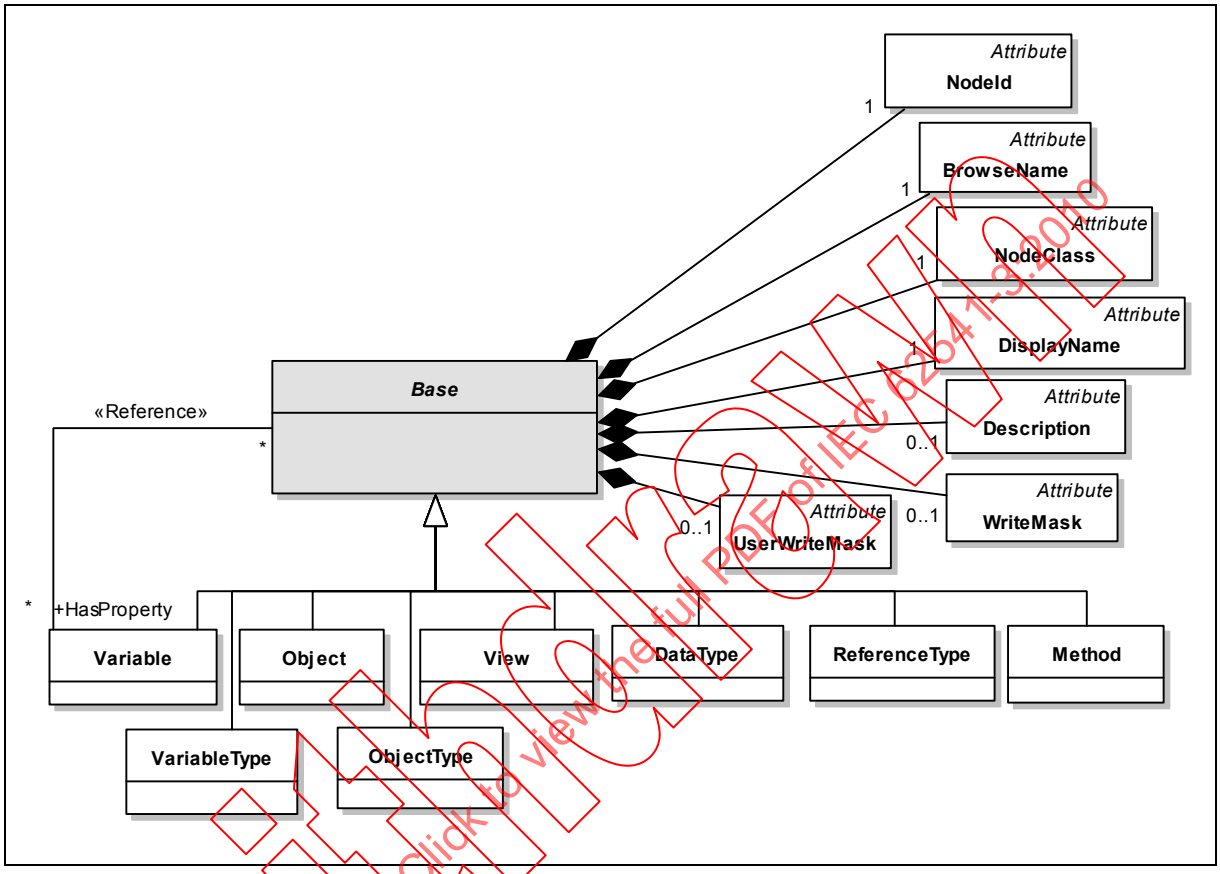


Figure B.4 – BaseNode

B.3.2 ReferenceType

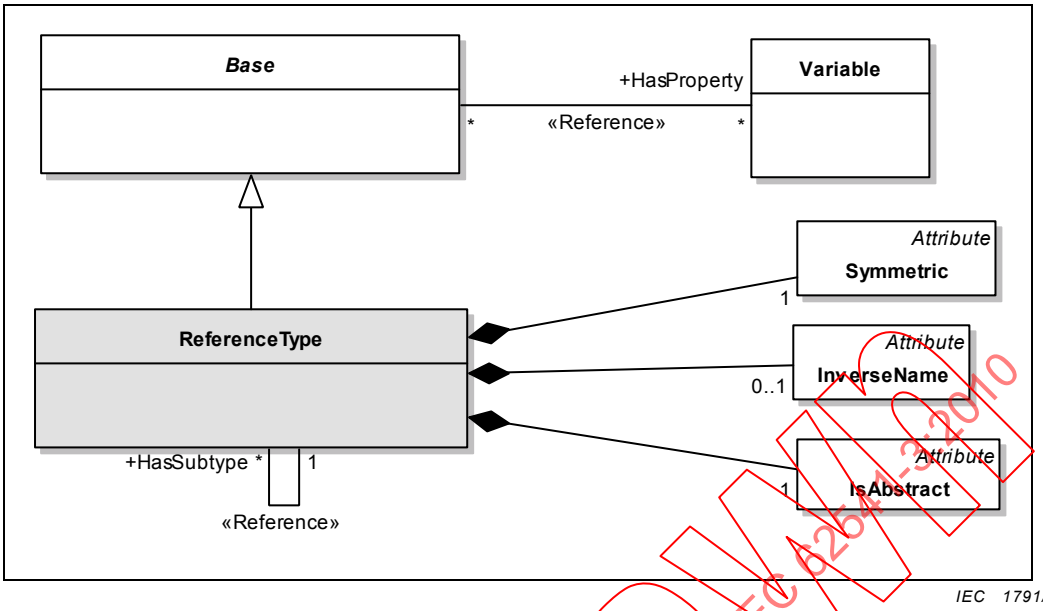
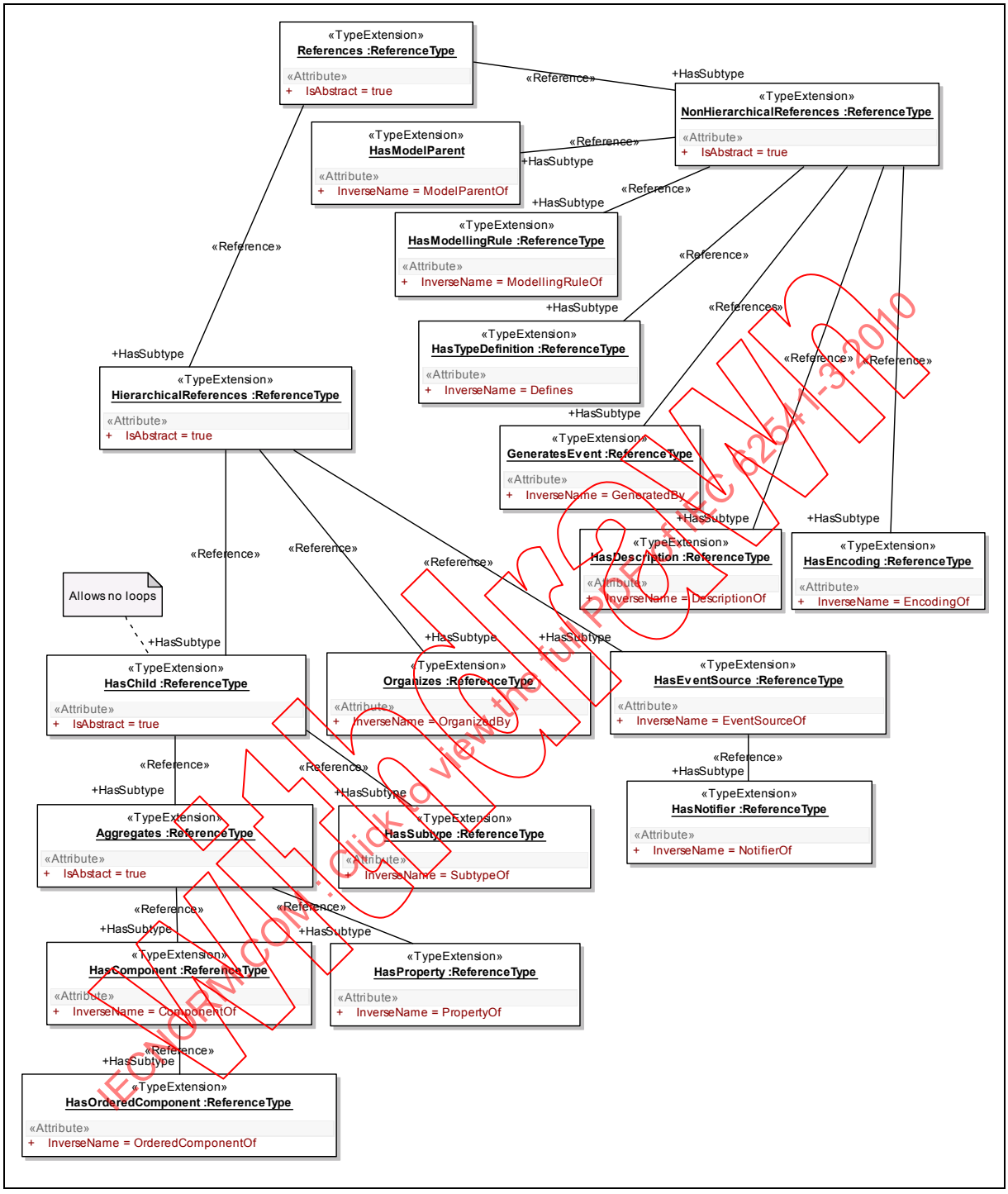


Figure B.5 – Reference and ReferenceType

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

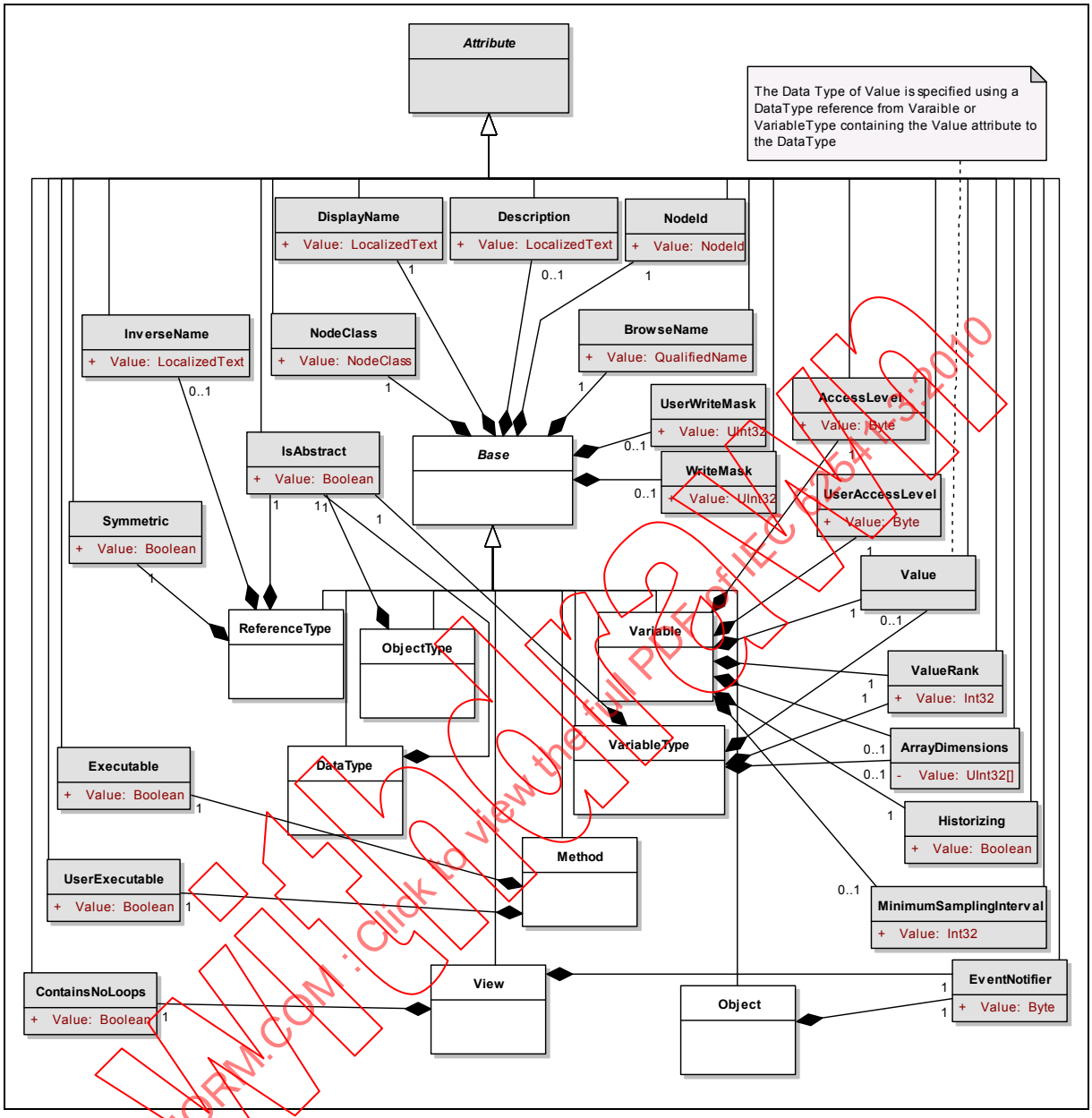
B.3.3 Predefined ReferenceTypes



IEC 1792/10

Figure B.6 – Predefined ReferenceTypes

B.3.4 Attributes



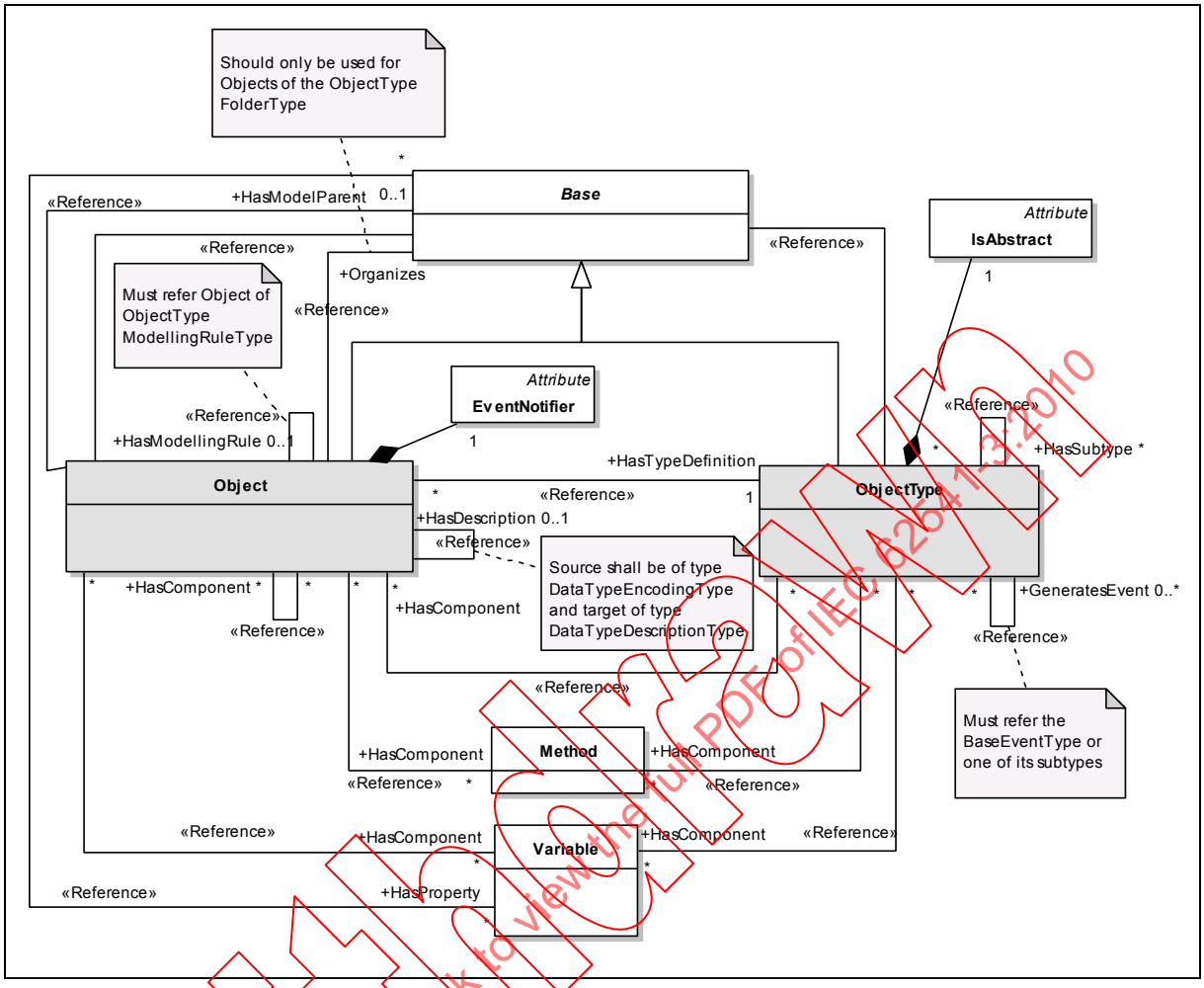
IEC 1793/10

Figure B.7 – Attributes

There may be more *Attributes* defined in other parts of this series of standards.

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

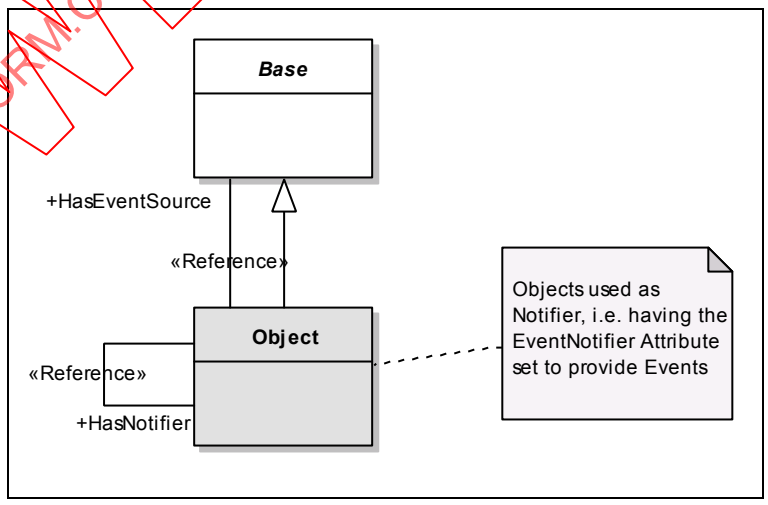
B.3.5 Object and ObjectType



IEC 1794/10

Figure B.8 – Object and ObjectType

B.3.6 EventNotifier



IEC 1795/10

Figure B.9 – EventNotifier

B.3.7 Variable and VariableType

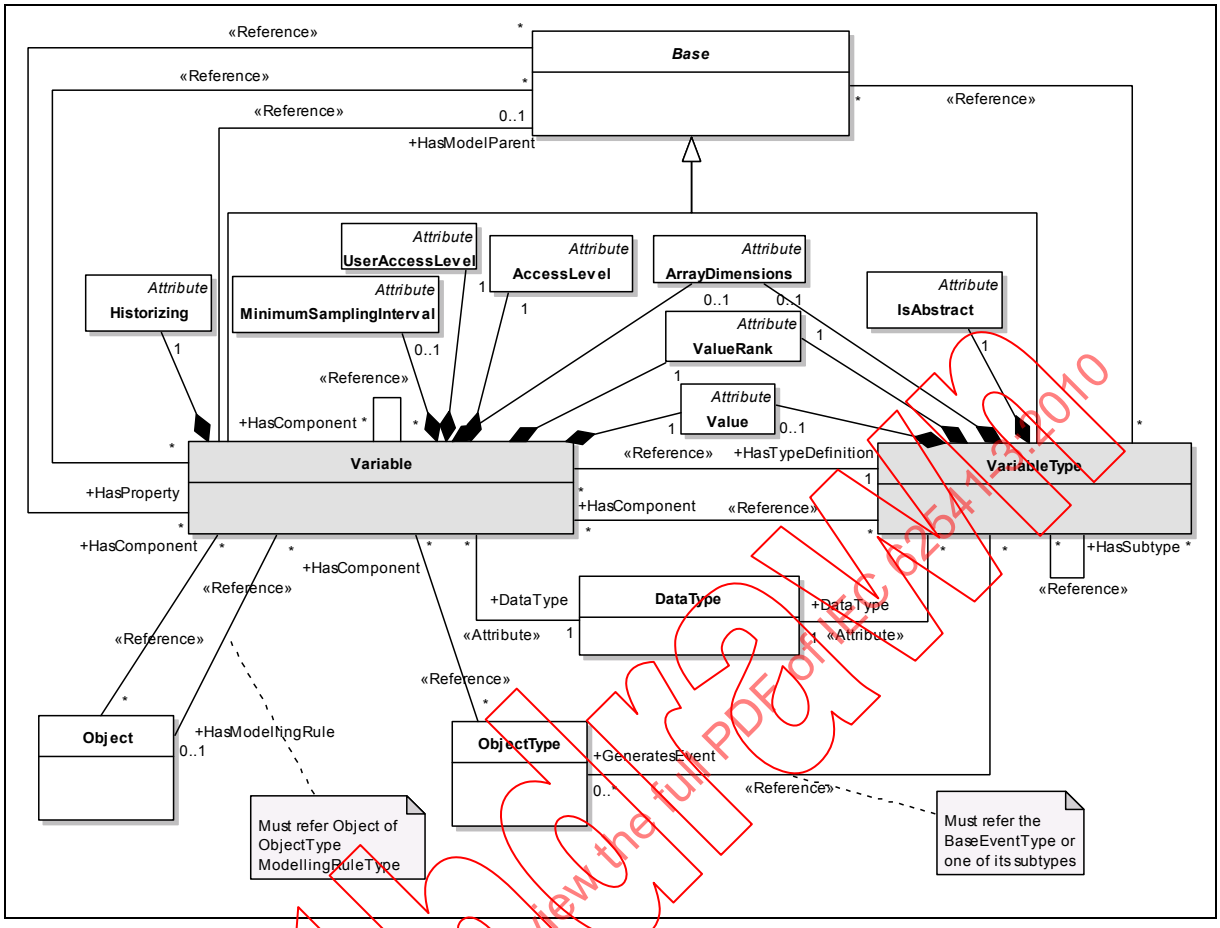


Figure B.10 – Variable and VariableType

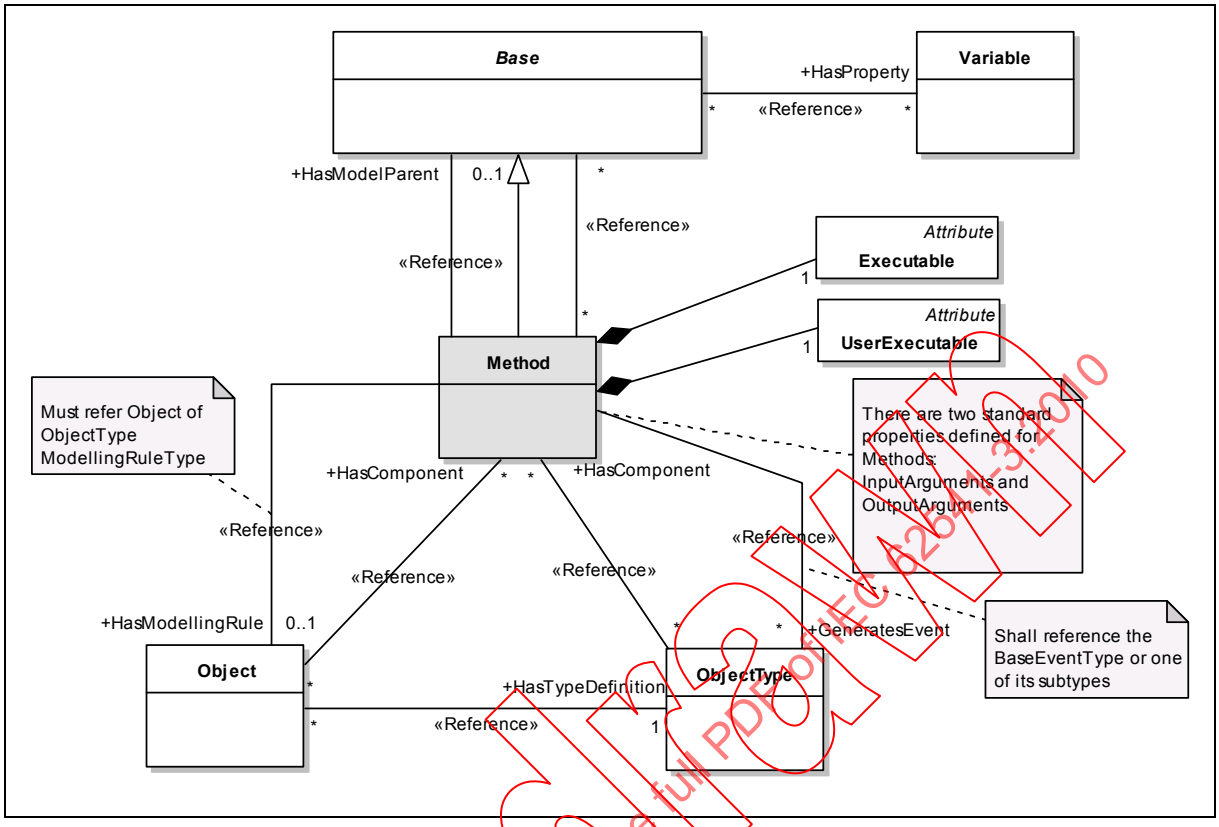
The *DataType* of a *Variable* shall be the same or a subtype of the *DataType* of its *VariableType* (referred with *HasTypeDefinition*).

If a *HasProperty* points to a *Variable* from a *Base* "A" 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



IEC 1797/10

Figure B.11 – Method

B.3.9 DataType

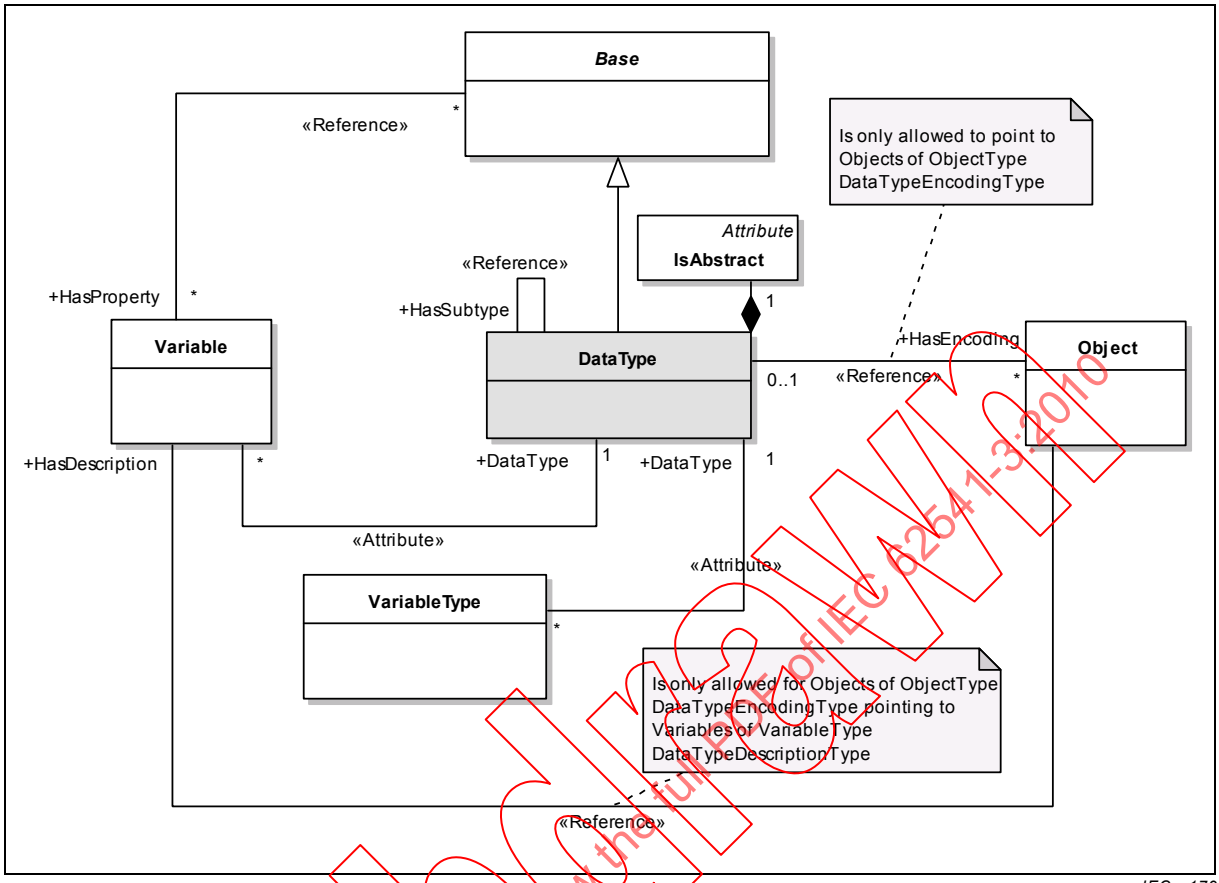


Figure B.12 – DataType

B.3.10 View

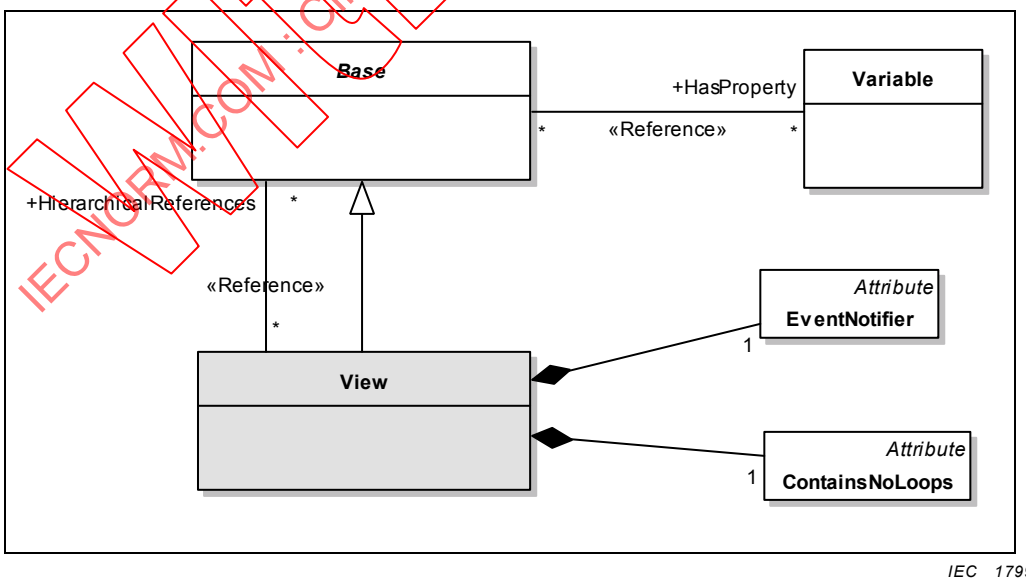


Figure B.13 – View

Annex C
(normative)

OPC Binary Type Description System

C.1 Concepts

The OPC Binary XML Schema defines the format of OPC Binary *TypeDictionaries*. Each OPC Binary *TypeDictionary* is an XML document that contains one or more *TypeDescriptions* that describe the format of a binary-encoded value. Applications that have no advance knowledge of a particular binary encoding can use the OPC Binary *TypeDescription* to interpret or construct a value.

The OPC Binary Type Description System does not define a standard mechanism to *encode* data in binary. It only provides a standard way to *describe* an existing binary encoding. Many binary encodings will have a mechanism to describe types that could be encoded; however, these descriptions are useful only to applications that have knowledge of the type description system used with each binary encoding. The OPC Binary Type Description System is a generic syntax that can be used by any application to interpret any binary encoding.

The OPC Binary Type Description System was originally defined in the OPC Complex Data Specification. The OPC Binary Type Description System described in this Annex is quite different and is correctly described as the OPC Binary Type Description System Version 2.0.

Each *TypeDescription* is identified by a *TypeName* which shall be unique within the *TypeDictionary* that defines it. Each *TypeDictionary* also has a *TargetNamespace* which should be unique among all OPC Binary *TypeDictionaries*. This mean that the *TypeName* qualified with the *TargetNamespace* for the dictionary should be a globally-unique identifier for a *TypeDescription*.

Figure C.1 below illustrates the structure of an OPC Binary *TypeDictionary*.

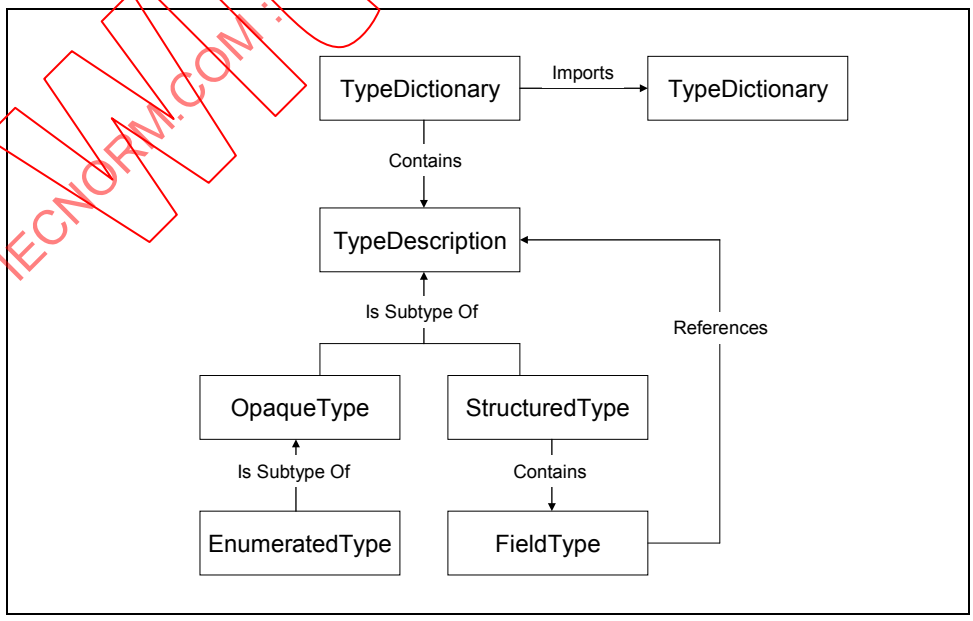


Figure C.1 – OPC Binary Dictionary Structure

Each binary encoding is built from a set of opaque building blocks that are either primitive types with a fixed length or variable-length types with a structure that is too complex to

describe properly in an XML document. These building blocks are described with an *OpaqueType*. An instance of one of these building blocks is a binary-encoded value.

The OPC Binary Type Description System defines a set of standard *OpaqueTypes* that all OPC Binary *TypeDictionaries* should use to build their *TypeDescriptions*. These standard type descriptions are described in C.3.

In some cases, the binary encoding described by an *OpaqueType* may have a fixed size which would allow an application to skip an encoded value that it does not understand. If that is the case, the *LengthInBits* attribute should be specified for the *OpaqueType.* If authors of *TypeDictionaries* need to define new *OpaqueTypes* that do not have a fixed size then they should use the documentation elements to describe how to encode binary values for the type. This description should provide enough detail to allow a human to write a program that can interpret instances of the type.

A *StructuredType* breaks a complex value into a sequence of values that are described by a *FieldType*. Each *FieldType* has a name, type and a number of qualifiers that specify when the field is used and how many instances of the type exist. A *FieldType* is described completely in C.2.6.

An *EnumeratedType* describes a numeric value that has a limited set of possible values, each of which has a descriptive name. *EnumeratedTypes* provide a convenient way to capture semantic information associated with what would otherwise be an opaque numeric value.

C.2 Schema Description

C.2.1 TypeDictionary

The *TypeDictionary* element is the root element of an OPC Binary dictionary. The components of this element are described in Table C.1.

Table C.1 – TypeDictionary Components

Name	Type	Description
Documentation	Documentation	An element that contains human-readable text and XML that provides an overview of what is contained in the dictionary.
Import	ImportDirective[]	Zero or more elements that specify other *TypeDictionaries* that are referenced by *StructuredTypes* defined in the dictionary. Each import element specifies the *NamespaceURI* of the *TypeDictionary* being imported. The *TypeDictionary* element shall declare an XML namespace prefix for each imported namespace.
TargetNamespace	xs:string	Specifies the URI that qualifies all *TypeDescriptions* defined in the dictionary.
DefaultByteOrder	ByteOrder	Specifies the default *ByteOrder* for all *TypeDescriptions* that have the *ByteOrderSignificant* attribute set to “true”. This value overrides the setting in any imported *TypeDictionary*. This value is overridden by the *DefaultByteOrder* specified on a *TypeDescription*.
TypeDescription	TypeDescription[]	One or more elements that describe the structure of a binary encoded value. A TypeDescription is an abstract type. A dictionary may only contain the *OpaqueType*, *EnumeratedType* and *StructuredType* elements.

C.2.2 TypeDescription

A *TypeDescription* describes the structure of a binary encoded value. A *TypeDescription* is an abstract base type and only instances of subtypes may appear in a *TypeDictionary*. The components of a *TypeDescription* are described in Table C.2.

Table C.2 – TypeDescription Components

Name	Type	Description
Documentation	Documentation	An element that contains human readable text and XML that describes the type. This element should capture any semantic information that would help a human to understand what is contained in the value.
Name	xs: NCName	An attribute that specifies a name for the *TypeDescription* that is unique within the dictionary. The fields of structured types reference *TypeDescriptions* by using this name qualified with the dictionary namespace URI.
DefaultByteOrder	ByteOrder	An attribute that specifies the default *ByteOrder* for the type description. This value overrides the setting in any *TypeDictionary* or in any *StructuredType* that references the type description.

C.2.3 OpaqueType

An *OpaqueType* describes a binary encoded value that is either a primitive fixed length type or that has a structure too complex to capture in an OPC Binary type dictionary. Authors of type dictionaries should avoid defining *OpaqueTypes* that do not have a fixed length because it would prevent applications from interpreting values that use these types without having built-in knowledge of the *OpaqueType.* The OPC Binary Type Description System defines many standard *OpaqueTypes* that should allow authors to describe most binary encoded values as *StructuredTypes*.

The components of an *OpaqueTypeDescription* are described in Table C.3.

Table C.3 – OpaqueType Components

Name	Type	Description
TypeDescription	TypeDescription	An *OpaqueType* inherits all elements and attributes defined for a *TypeDescription* in Table C.2.
LengthInBits	xs:string	An attribute which specifies the length of the *OpaqueType* in bits. This value should always be specified. If this value is not specified the *Documentation* element should describe the encoding in a way that a human understands.
ByteOrderSignificant	xs:boolean	An attribute that indicates whether byte order is significant for the type. If byte order is significant then the application shall determine the byte order to use for the current context before interpreting the encoded value. The application determines the byte order by looking for the *DefaultByteOrder* attribute specified for containing *StructuredTypes* or the *TypeDictionary*. If StructuredTypes are nested the inner *StructuredTypes* override the byte order of the outer descriptions. If the *DefaultByteOrder* attribute is specified for the *OpaqueType*, then the *ByteOrder* is fixed and does not change according to context. If this attribute is “true”, then the *LengthInBits* attribute shall be specified and it shall be an integer multiple of 8 bits.

C.2.4 EnumeratedType

An *EnumeratedType* describes a binary-encoded numeric value that has a fixed set of valid values. The encoded binary value described by an *EnumeratedType* is always an unsigned integer with a length specified by the *LengthInBits* attribute.

The names for each of the enumerated values are not required to interpret the binary encoding, however, they form part of the documentation for the type.

The components of an *EnumeratedType* are described in Table C.4.

Table C.4 – EnumeratedType Components

Name	Type	Description
OpaqueType	OpaqueTypeDescription	An *EnumeratedType* inherits all elements and attributes defined for a *TypeDescription* in Table C.2 and for an *OpaqueType* defined in Table C.3. The *LengthInBits* attribute shall always be specified.
EnumeratedValue	EnumeratedValue	One or more elements that describe the possible values for the instances of the type.

C.2.5 StructuredType

A *StructuredType* describes a type as a sequence of binary-encoded values. Each value in the sequence is called a *Field*. Each *Field* references a *TypeDescription* that describes the binary-encoded value that appears in the field. A *Field* may specify that zero, one or multiple instances of the type appear within the sequence described by the *StructuredType*.

Authors of type dictionaries should use *StructuredTypes* to describe a variety of common data constructs including arrays, unions and structures.

Some fields have lengths that are not multiples of 8 bits. Several of these fields may appear in a sequence in a structure, however, the total number of bits used in the sequence shall be fixed and it shall be a multiple of 8 bits. Any field which does not have a fixed length shall be aligned on a byte boundary.

A sequence of fields which do not line up on byte boundaries are specified from the least significant bit to the most significant bit. Sequences which are longer than one byte overflow from the most significant bit of the first byte into the least significant bit of the next byte.

The components of a *StructuredType* are described in Table C.5.

Table C.5 – StructuredType Components

Name	Type	Description
TypeDescription	TypeDescription	A *StructuredType* inherits all elements and attributes defined for a *TypeDescription* in Table C.2.
Field	FieldType	One or more elements that describe the fields of the structure. Each field shall have a name that is unique within the *StructuredType*. Some fields may reference other fields in the *StructuredType* by using this name.
anyAttribute	*	Authors of a *TypeDictionary* may add their own attributes to any *StructuredType* that shall be qualified with a namespace defined by the author. Applications should not be required to understand these attributes in order to interpret a binary encoded instance of the type.

C.2.6 FieldType

A *FieldType* describes a binary encoded value that appears in sequence within a *StructuredType*. Every *FieldType* shall reference a *TypeDescription* that describes the encoded value for the field.

A *FieldType* may specify an array of encoded values.

Fields may be optional and they reference other *FieldTypes*, which indicate if they are present in any specific instance of the type.

The components of a *FieldType* are described in Table C.6.

Table C.6 – FieldType Components

Name	Type	Description
Documentation	Documentation	An element that contains human readable text and XML that describes the field. This element should capture any semantic information that would help a human to understand what is contained in the field.
Name	xs:string	An attribute that specifies a name for the *Field* that is unique within the *StructuredType*. Other fields in the structured type reference a *Field* by using this name.
TypeName	xs:QName	An attribute that specifies the *TypeDescription* that describes the contents of the field. A field may contain zero or more instances of this type depending on the settings for the other attributes and the values in other fields
Length	xs:unsignedInt	An attribute that indicates length of the field. This value may be the total number of encoded bytes or it may be the number of instances of the type referenced by the field. The *IsLengthInBytes* attributes specifies which of these definitions applies.
LengthField	xs:string	An attribute that indicates which other field in the *StructuredType* specifies the length of the field. The length of the field may be in bytes or it may be the number of instances of the type referenced by the field. The *IsLengthInBytes* attributes specifies which of these definitions applies. If this attribute refers to a field that is not present in an encoded value, then the default value for the length is 1. This situation could occur if the field referenced is an optional field (see the *SwitchField* attribute). The length field shall be a fixed length Base-2 representation of an integer. If the length field is one of the standard signed integer types and the value is a negative integer, then the field is not present in the encoded stream. The *FieldType* referenced by this attribute shall precede the field with the *StructuredType*.
IsLengthInBytes	xs:boolean	An attribute that indicates whether the *Length* or *LengthField* attributes specify the length of the field in bytes or in the number of instances of the type referenced by the field.
SwitchField	xs:string	If this attribute is specified, then the field is optional and may not appear in every instance of the encoded value. This attribute specifies the name of another *Field* that controls whether this field is present in the encoded value. The field referenced by this attribute shall be an integer value (see the *LengthField* attribute). The current value of the switch field is compared to the *SwitchValue* attribute using the *SwitchOperand.* If the condition evaluates to true then the field appears in the stream. If the *SwitchValue* attribute is not specified, then this field is present if the value of the switch field is non-zero. The *SwitchOperand* field is ignored if it is present. If the *SwitchOperand* attribute is missing, then the field is present if the value of the switch field is equal to the value of the *SwitchValue* attribute. The *Field* referenced by this attribute shall precede the field with the *StructuredType*.
SwitchValue	xs:unsignedInt	This attribute specifies when the field appears in the encoded value. The value of the field referenced by the *SwitchName* attribute is compared using the SwitchOperand attribute to this value. The field is present if the expression evaluates to true. The field is not present otherwise.
SwitchOperand	xs:string	This attribute specifies how the value of the switch field should be compared to the switch value attribute. This field is an enumeration with the following values: Equal – *SwitchField* is equal to the *SwitchValue*. GreaterThan – *SwitchField* is greater than the *SwitchValue*. LessThan – *SwitchField* is less than the *SwitchValue*. GreaterThanOrEqual – *SwitchField* is greater than or equal to the *SwitchValue*. LessThanOrEqual – *SwitchField* is less than or equal to the *SwitchValue*. NotEqual – *SwitchField* is not equal to the *SwitchValue*. In each case the field is present if the expression is true.

Name	Type	Description
Terminator	xs:hexBinary	This attribute indicates that the field contains one or more instances of *TypeDescription* referenced by this field and that the last value has the binary encoding specified by the value of this attribute. If this attribute is specified then the *TypeDescription* referenced by this field shall either have a fixed byte order (i.e. byte order is not significant or explicitly specified) or the containing StructuredType shall explicitly specify the byte order. EXAMPLES Field Data Type / Terminator / Byte Order / Hexadecimal String Char / tab character / not applicable / 09 WideChar: / tab character / BigEndian / 0009 WideChar: / tab character / LittleEndian / 0900 Int16 / 1 / BigEndian / 0001 Int16 / 1 / LittleEndian / 0100
anyAttribute	*	Authors of a *TypeDictionary* may add their own attributes to any *FieldType* which shall be qualified with a namespace defined by the authors. Applications should not be required to understand these attributes in order to interpret a binary encoded field value.

C.2.7 EnumeratedValue

An *EnumeratedValue* describes a possible value for an *EnumeratedType*.

The components of an *EnumeratedValue* are described in Table C.7.

Table C.7 – EnumeratedValue Components

Name	Type	Description
Name	xs:string	This attribute specifies a descriptive name for the enumerated value.
Value	xs:unsignedInt	This attribute specifies the numeric value that could appear in the binary encoding.

C.2.8 ByteOrder

A *ByteOrder* is an enumeration that describes a possible value byte orders for *TypeDescriptions* that allow different byte orders to be used. There are two possible values: BigEndian and LittleEndian. BigEndian indicates the most significant byte appears first in the binary encoding. LittleEndian indicates that the least significant byte appears first.

C.2.9 ImportDirective

An *ImportDirective* specifies a *TypeDictionary* that is referenced by *FiledDescriptions* defined in the current dictionary.

The components of an *ImportDirective* are described in Table C.8.

Table C.8 – ImportDirective Components

Name	Type	Description
Namespace	xs:string	This attribute specifies the *TargetNamespace* for the *TypeDictionary* being imported. This may be a well-known URI which means applications need not have access to the physical file to recognise types that are referenced.
Location	xs:string	This attribute specifies the physical location of the XML file containing the *TypeDictionary* to import. This value could be a URL for a network resource, a NodeId in an OPC UA server address space or a local file path.

C.3 Standard Type Descriptions

The OPC Binary Type Description System defines a number of standard type descriptions that can be used to describe many common binary encodings using a *StructuredType*. The standard type descriptions are described in Table C.9.

Table C.9 – Standard Type Descriptions

Type name	Description
Bit	A single bit value.
Boolean	A two-state logical value represented as an 8-bit value.
SByte	An 8-bit signed integer.
Byte	An 8-bit unsigned integer.
Int16	A 16-bit signed integer.
UInt16	A 16-bit unsigned integer.
Int32	A 32-bit signed integer.
UInt32	A 32-bit unsigned integer.
Int64	A 64-bit signed integer.
UInt64	A 64-bit unsigned integer.
Float	An IEEE 754-1985 single precision floating point value.
Double	An IEEE 754-1985 double precision floating point value.
Char	An 8-bit UTF-8 character value.
WideChar	A 16-bit UTF-16 character value.
String	A null terminated sequence of UTF-8 characters.
CharArray	A sequence of UTF-8 characters preceded by the number of characters.
WideString	A null terminated sequence of UTF-16 characters.
WideCharArray	A sequence of UTF-16 characters preceded by the number of characters.
DateTime	A 64-bit signed integer representing the number of 100 nanoseconds intervals since 1601-01-01 00:00:00. This is the same as the WIN32 FILETIME type.
ByteString	A sequence of bytes preceded by its length in bytes.
Guid	A 128-bit structured type that represents a WIN32 GUID value.

C.4 Type Description Examples

1. A 128-bit signed integer.

```
<opc:OpaqueType Name="Int128" LengthInBits="128">
  <opc:Documentation>A 128-bit signed integer.</opc:Documentation>
</opc:OpaqueType>
```

2. A 16-bit value divided into several fields.

```
<opc:StructuredType Name="Quality">
  <opc:Documentation>An OPC COM-DA quality value.</opc:Documentation>
  <opc:Field Name="LimitBits" TypeName="opc:Bit" Length="2" />
  <opc:Field Name="QualityBits" TypeName="opc:Bit" Length="6"/>
  <opc:Field Name="VendorBits" TypeName="opc:Byte" />
</opc:StructuredType>
```

When using bit fields, the least significant bits within a byte shall appear first.

3. A structured type with optional fields.

```
<opc:StructuredType Name="DataValue">
  <opc:Documentation>A value with an associated timestamp, and
quality.</opc:Documentation>
  <opc:Field Name="ValueSpecified" TypeName="Bit" />
  <opc:Field Name="StatusCodeSpecified" TypeName="Bit" />
  <opc:Field Name="TimestampSpecified" TypeName="Bit" />
  <opc:Field Name="Reserved1" TypeName="Bit" Length="5" />
  <opc:Field Name="Value" TypeName="Variant" SwitchField="ValueSpecified" />
  <opc:Field Name="Quality" TypeName="Quality" SwitchField="StatusCodeSpecified" />
  <opc:Field Name="Timestamp" TypeName="opc:DateTime"
SwitchField="SourceTimestampSpecified" />
```

```
</opc:StructuredType>
```

It is nesessary to explictly specify any padding bits required to ensure subsequent fields line up on byte boundaries.

4. An array of integers.

```
<opc:StructuredType Name="IntegerArray">
  <opc:Documentation>An array of integers prefixed by its length.</opc:Documentation>
  <opc:Field Name="Size" TypeName="opc:Int32" />
  <opc:Field Name="Array" TypeName="opc:Int32" LengthField="Size" />
</opc:StructuredType>
```

Nothing is encoded for the Array field if the Size field has a value <= 0.

5. An array of integers with a terminator instead of a length prefix.

```
<opc:StructuredType Name="IntegerArray" DefaultByteOrder="LittleEndian">
  <opc:Documentation>An array of integers terminated with a known
value.</opc:Documentation>
  <opc:Field Name="Value" TypeName="opc:Int16" Terminator="FF7F" />
</opc:StructuredType>
```

The terminator is 32,767 converted to hexadecimal with LittleEndian byte order.

6. A simple union.

```
<opc:StructuredType Name="Variant">
  <opc:Documentation>A union of several types.</opc:Documentation>
  <opc:Field Name="ArrayLengthSpecified" TypeName="opc:Bit" Length="1"/>
  <opc:Field Name="VariantType" TypeName="opc:Bit" Length="7" />
  <opc:Field Name="ArrayLength" TypeName="opc:Int32"
      SwitchField="ArrayLengthSpecified" />
  <opc:Field Name="Int32" TypeName="opc:Int32" LengthField="ArrayLength"
      SwitchField="VariantType" SwitchValue="1" />
  <opc:Field Name="String" TypeName="opc:String" LengthField="ArrayLength"
      SwitchField="VariantType" SwitchValue="2" />
  <opc:Field Name="DateTime" TypeName="opc:DateTime" LengthField="ArrayLength"
      SwitchField="VariantType" SwitchValue="3" />
</opc:StructuredType>
```

The *ArrayLength* field is optional. If it is not present in an encoded value, then the length of all fields with *LengthField* set to “ArrayLength” have a length of 1.

It is valid for the the *VariantType* field to have a value that has no matching field defined. This simply means all optional fields are not present in the encoded value.

7. An enumerated type.

```
<opc:EnumeratedType Name="TrafficLight" LengthInBits="32">
  <opc:Documentation>The possible colours for a traffic signal.</opc:Documentation>
  <opc:EnumeratedValue Name="Red" Value="4">
    <opc:Documentation>Red says stop immediately.</opc:Documentation>
  </opc:EnumeratedValue>
  <opc:EnumeratedValue Name="Yellow" Value="3">
    <opc:Documentation>Yellow says prepare to stop.</opc:Documentation>
  </opc:EnumeratedValue>
  <opc:EnumeratedValue Name="Green" Value="2">
    <opc:Documentation>Green says you may proceed.</opc:Documentation>
  </opc:EnumeratedValue>
</opc:EnumeratedType>
```

The documentation element is used to provide human readable description of the type and values.

8. A nillable array.

```
<opc:StructuredTypen Name="NillableArray">
  <opc:Documentation>An array where a length of -1 means null.</opc:Documentation>
  <opc:Field Name="Length" TypeName="opc:Int32" />
  <opc:Field
```

```
      Name="Int32"
      TypeName="opc:Int32"
      LengthField="Length"
      SwitchField="Length"
      SwitchValue="0"
      SwitchOperand="GreaterThanOrEqual" />
</opc:StructuredType>
```

If the length of the array is −1 then the array does not appear in the stream.

C.5 OPC Binary XML Schema

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema
  targetNamespace="http://opcfoundation.org/BinarySchema/"
  elementFormDefault="qualified"
  xmlns="http://opcfoundation.org/BinarySchema/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
>
  <xs:element name="Documentation">
    <xs:complexType mixed="true">
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:any minOccurs="0" maxOccurs="unbounded"/>
      </xs:choice>
      <xs:anyAttribute/>
    </xs:complexType>
  </xs:element>

  <xs:complexType name="ImportDirective">
    <xs:attribute name="Namespace" type="xs:string" use="optional" />
    <xs:attribute name="Location" type="xs:string" use="optional" />
  </xs:complexType>

  <xs:simpleType name="ByteOrder">
    <xs:restriction base="xs:string">
      <xs:enumeration value="BigEndian" />
      <xs:enumeration value="LittleEndian" />
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name="TypeDescription">
    <xs:sequence>
      <xs:element ref="Documentation" minOccurs="0" maxOccurs="1" />
    </xs:sequence>
    <xs:attribute name="Name" type="xs:NCName" use="required" />
    <xs:attribute name="DefaultByteOrder" type="ByteOrder" use="optional" />
  </xs:complexType>

  <xs:complexType name="OpaqueType">
    <xs:complexContent>
      <xs:extension base="TypeDescription">
        <xs:attribute name="LengthInBits" type="xs:int" use="optional" />
        <xs:attribute name="ByteOrderSignificant" type="xs:boolean" default="false" />
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name="EnumeratedValue">
    <xs:sequence>
      <xs:element ref="Documentation"  minOccurs="0" maxOccurs="1" />
    </xs:sequence>
    <xs:attribute name="Name" type="xs:string" use="optional" />
    <xs:attribute name="Value" type="xs:unsignedInt" use="optional" />
  </xs:complexType>

  <xs:complexType name="EnumeratedType">
    <xs:complexContent>
      <xs:extension base="OpaqueTypeDescription">
        <xs:sequence>
       <xs:element name="EnumeratedValue" type="EnumeratedValueDescription"
maxOccurs="unbounded" />
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:simpleType name="SwitchOperand">
```

```
    <xs:restriction base="xs:string">
      <xs:enumeration value="Equals" />
      <xs:enumeration value="GreaterThan" />
      <xs:enumeration value="LessThan" />
      <xs:enumeration value="GreaterThanOrEqual" />
      <xs:enumeration value="LessThanOrEqual" />
      <xs:enumeration value="NotEqual" />
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name="FieldType">
    <xs:sequence>
      <xs:element ref="Documentation" minOccurs="0" maxOccurs="1" />
    </xs:sequence>
    <xs:attribute name="Name" type="xs:string" use="required" />
    <xs:attribute name="TypeName" type="xs:QName" use="optional" />
    <xs:attribute name="Length" type="xs:unsignedInt" use="optional" />
    <xs:attribute name="LengthField" type="xs:string" use="optional" />
    <xs:attribute name="IsLengthInBytes" type="xs:boolean" default="false" />
    <xs:attribute name="SwitchField" type="xs:string" use="optional" />
    <xs:attribute name="SwitchValue" type="xs:unsignedInt" use="optional" />
    <xs:attribute name="SwitchOperand" type="SwitchOperand" use="optional" />
    <xs:attribute name="Terminator" type="xs:hexBinary" use="optional" />
    <xs:anyAttribute processContents="lax" />
  </xs:complexType>

  <xs:complexType name="StructuredType">
    <xs:complexContent>
      <xs:extension base="TypeDescription">
        <xs:sequence>
          <xs:element name="Field" type="FieldType" minOccurs="0" maxOccurs="unbounded"
/>
        </xs:sequence>
        <xs:anyAttribute processContents="lax" />
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:element name="TypeDictionary">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Documentation" minOccurs="0" maxOccurs="1" />
        <xs:element name="Import" type="ImportDirective" minOccurs="0"
maxOccurs="unbounded" />
        <xs:choice minOccurs="0" maxOccurs="unbounded">
          <xs:element name="OpaqueType" type="OpaqueType" />
          <xs:element name="EnumeratedType" type="EnumeratedType" />
          <xs:element name="StructuredType" type="StructuredType" />
        </xs:choice>
      </xs:sequence>
      <xs:attribute name="TargetNamespace" type="xs:string" use="required" />
      <xs:attribute name="DefaultByteOrder" type="ByteOrder" use="optional" />
    </xs:complexType>
  </xs:element>

</xs:schema>
```

C.6 OPC Binary Standard TypeDictionary

```
<?xml version="1.0" encoding="utf-8"?>
<opc:TypeDictionary
  xmlns="http://opcfoundation.org/BinarySchema/"
  xmlns:opc="http://opcfoundation.org/BinarySchema/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  TargetNamespace="http://opcfoundation.org/BinarySchema/"
>
  <opc:Documentation>This dictionary defines the standard types used by the OPC Binary
type description system.</opc:Documentation>

  <opc:OpaqueType Name="Bit" LengthInBits="1">
    <opc:Documentation>A single bit.</opc:Documentation>
  </opc:OpaqueType>

  <opc:OpaqueType Name="Boolean" LengthInBits="8">
    <opc:Documentation>A two state logical value represented as a 8-bit
value.</opc:Documentation>
  </opc:OpaqueType>
```

```
  <opc:OpaqueType Name="SByte" LengthInBits="8">
    <opc:Documentation>An 8-bit signed integer.</opc:Documentation>
  </opc:OpaqueType>

  <opc:OpaqueType Name="Byte" LengthInBits="8">
    <opc:Documentation>A 8-bit unsigned integer.</opc:Documentation>
  </opc:OpaqueType>

  <opc:OpaqueType Name="Int16" LengthInBits="16" ByteOrderSignificant="true">
    <opc:Documentation>A 16-bit signed integer.</opc:Documentation>
  </opc:OpaqueType>

  <opc:OpaqueType Name="UInt16" LengthInBits="16" ByteOrderSignificant="true">
    <opc:Documentation>A 16-bit unsigned integer.</opc:Documentation>
  </opc:OpaqueType>

  <opc:OpaqueType Name="Int32" LengthInBits="32" ByteOrderSignificant="true">
    <opc:Documentation>A 32-bit signed integer.</opc:Documentation>
  </opc:OpaqueType>

  <opc:OpaqueType Name="UInt32" LengthInBits="32" ByteOrderSignificant="true">
    <opc:Documentation>A 32-bit unsigned integer.</opc:Documentation>
  </opc:OpaqueType>

  <opc:OpaqueType Name="Int64" LengthInBits="32" ByteOrderSignificant="true">
    <opc:Documentation>A 64-bit signed integer.</opc:Documentation>
  </opc:OpaqueType>

  <opc:OpaqueType Name="UInt64" LengthInBits="64" ByteOrderSignificant="true">
    <opc:Documentation>A 64-bit unsigned integer.</opc:Documentation>
  </opc:OpaqueType>

  <opc:OpaqueType Name="Float" LengthInBits="32" ByteOrderSignificant="true">
    <opc:Documentation>An IEEE-754 single precision floating point
value.</opc:Documentation>
  </opc:OpaqueType>

  <opc:OpaqueType Name="Double" LengthInBits="64" ByteOrderSignificant="true">
    <opc:Documentation>An IEEE-754 double precision floating point
value.</opc:Documentation>
  </opc:OpaqueType>

  <opc:OpaqueType Name="Char" LengthInBits="8">
    <opc:Documentation>A 8-bit character value.</opc:Documentation>
  </opc:OpaqueType>

  <opc:StructuredType Name="String">
    <opc:Documentation>A UTF-8 null terminated string value.</opc:Documentation>
    <opc:Field Name="Value" TypeName="Char" Terminator="00" />
  </opc:StructuredType>

  <opc:StructuredType Name="CharArray">
    <opc:Documentation>A UTF-8 string prefixed by its length in
characters.</opc:Documentation>
    <opc:Field Name="Length" TypeName="Int32" />
    <opc:Field Name="Value" TypeName="Char" LengthField="Length" />
  </opc:StructuredType>

  <opc:OpaqueType Name="WideChar" LengthInBits="16" ByteOrderSignificant="true">
    <opc:Documentation>A 16-bit character value.</opc:Documentation>
  </opc:OpaqueType>

  <opc:StructuredType Name="WideString">
    <opc:Documentation>A UTF-16 null terminated string value.</opc:Documentation>
    <opc:Field Name="Value" TypeName="WideChar" Terminator="0000" />
  </opc:StructuredType>

  <opc:StructuredType Name="WideCharArray">
    <opc:Documentation>A UTF-16 string prefixed by its length in
characters.</opc:Documentation>
    <opc:Field Name="Length" TypeName="Int32" />
    <opc:Field Name="Value" TypeName="WideChar" LengthField="Length" />
  </opc:StructuredType>

  <opc:StructuredType Name="ByteString">
    <opc:Documentation>An array of bytes prefixed by its length.</opc:Documentation>
    <opc:Field Name="Length" TypeName="Int32" />
    <opc:Field Name="Value" TypeName="Byte" LengthField="Length" />
```

```
  </opc:StructuredType>

  <opc:OpaqueType Name="DateTime" LengthInBits="64" ByteOrderSignificant="true">
    <opc:Documentation>The number of 100 nanosecond intervals since January 01,
1601.</opc:Documentation>
  </opc:OpaqueType>

  <opc:StructuredType Name="Guid">
    <opc:Documentation>A 128-bit globally unique identifier.</opc:Documentation>
    <opc:Field Name="Data1" TypeName="UInt32" />
    <opc:Field Name="Data2" TypeName="UInt16" />
    <opc:Field Name="Data3" TypeName="UInt16" />
    <opc:Field Name="Data4" TypeName="Byte" Length="8" />
  </opc:StructuredType>

</opc:TypeDictionary>
```

Annex D
(normative)

Graphical Notation

D.1 General

This annex defines a graphical notation for OPC UA data. This annex is normative, that is, the notation is used in this standard 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.

D.2 Notation

D.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. The following subclauses describe the simple and the complex notation.

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.

D.2.2 Simple Notation

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

Table D.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 D.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.

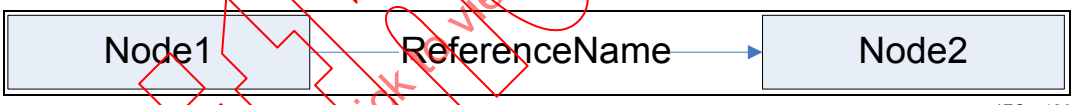


Figure D.1 – Example of a Reference connecting two Nodes

Table D.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 also the generic solution can be used.

Table D.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 to 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 to 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*.

D.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, also "MyName" can 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, also "MyName" can 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* has to be prefixed with "::". Figure D.2 gives an example, where "Node1" uses a *Reference* and "Node2" the shortcut. 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*.

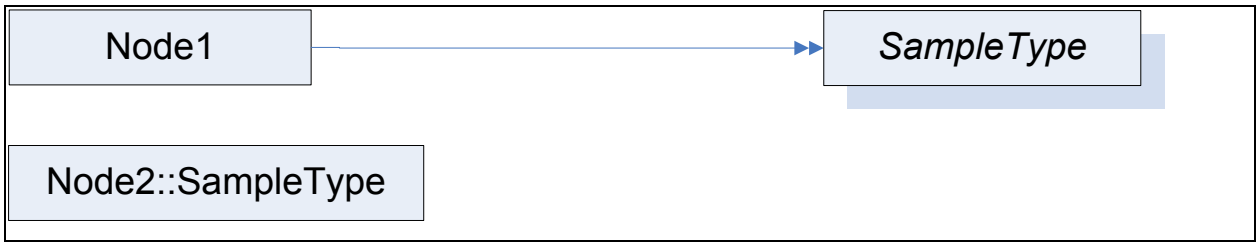


Figure D.2 – Example of using a TypeDefinition inside a Node

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 D.3.

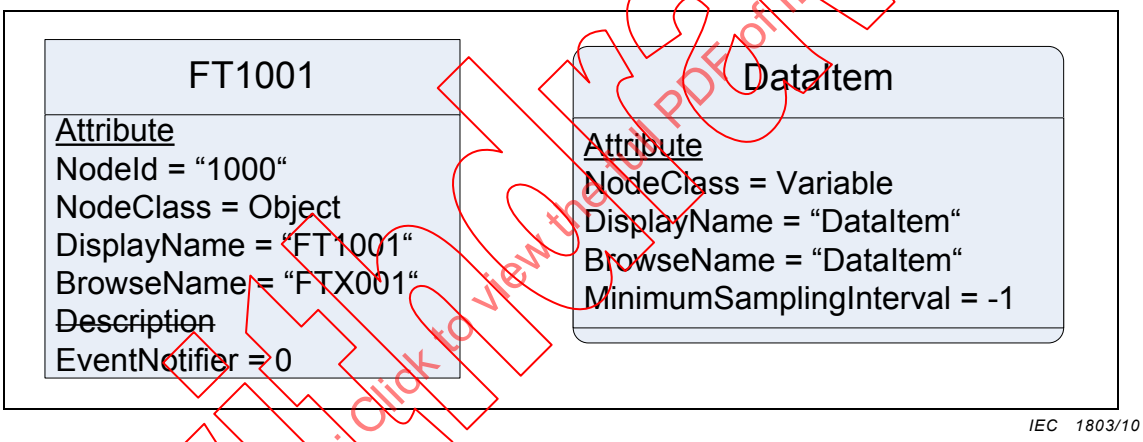


Figure D.3 – Example of exposing Attributes

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 D.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 and as well as an additional *Node*.

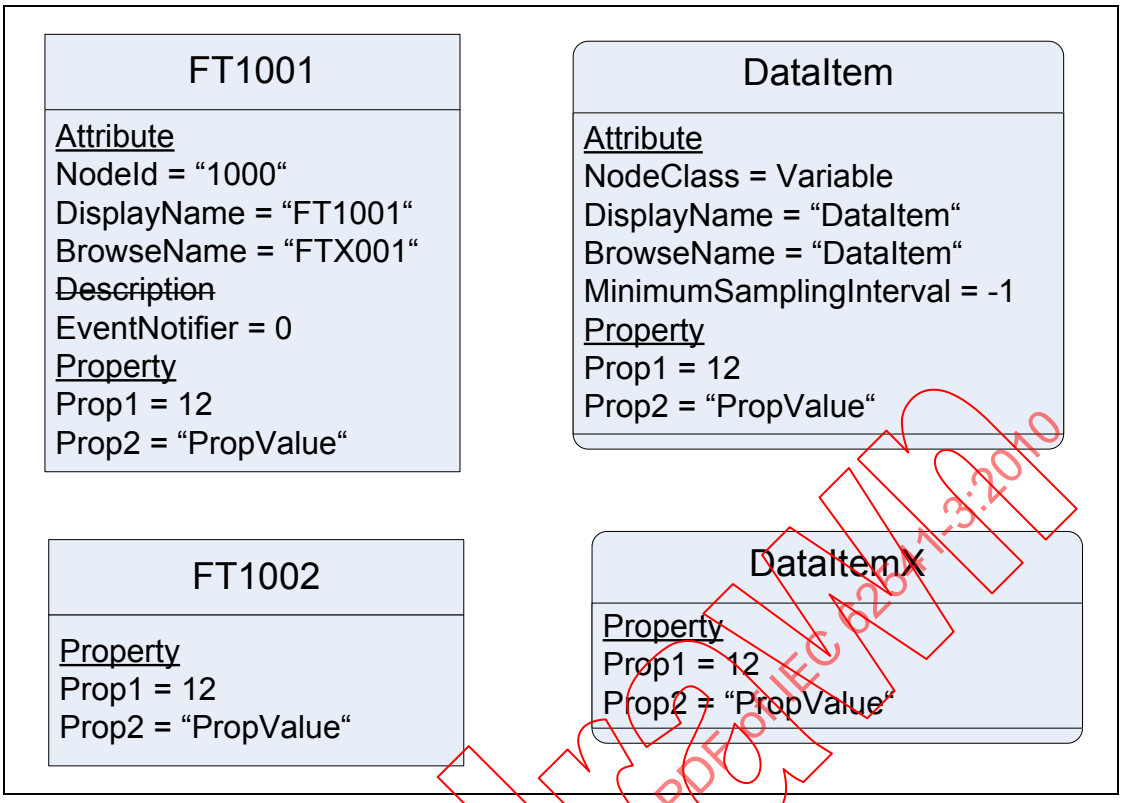


Figure D.4 – Example of exposing Properties inline

It is allowed to add additional information to a figure using the graphical representation, for example callouts.

Bibliography

IEC/TR 62541-2, *OPC Unified Architecture – Part 2: Security model*

IEC 62541-9[3], *OPC Unified Architecture – Part 9: Alarms and conditions*

IEC 62541-11[3], *OPC unified architecture – Part 11: Historical Access*

3 Under consideration.

SOMMAIRE

COMMISSION ÉLECTROTECHNIQUE INTERNATIONALE

ARCHITECTURE UNIFIÉE OPC –

Partie 3: Modèle de l'espace d'adressage

AVANT-PROPOS

1) La Commission Electrotechnique Internationale (CEI) est une organisation mondiale de normalisation composée de l'ensemble des comités électrotechniques nationaux (Comités nationaux de la CEI). La CEI 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, la CEI – 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 la CEI"). 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 la CEI, participent également aux travaux. La CEI 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 la CEI concernant les questions techniques représentent, dans la mesure du possible, un accord international sur les sujets étudiés, étant donné que les Comités nationaux de la CEI intéressés sont représentés dans chaque comité d'études.
3) Les Publications de la CEI se présentent sous la forme de recommandations internationales et sont agréées comme telles par les Comités nationaux de la CEI. Tous les efforts raisonnables sont entrepris afin que la CEI s'assure de l'exactitude du contenu technique de ses publications; la CEI 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 la CEI s'engagent, dans toute la mesure possible, à appliquer de façon transparente les Publications de la CEI dans leurs publications nationales et régionales. Toutes divergences entre toutes Publications de la CEI et toutes publications nationales ou régionales correspondantes doivent être indiquées en termes clairs dans ces dernières.
5) La CEI 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 la CEI. La CEI 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 la CEI, 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 la CEI, 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 la CEI ou de toute autre Publication de la CEI, 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 la CEI peuvent faire l'objet de droits de brevet. La CEI 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 CEI 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 la CEI: Mesure, commande et automation dans les processus industriels.

La présente version bilingue (2012-10) correspond à la version anglaise monolingue publiée en 2010-07.

Le texte anglais de cette norme est issu des documents 65E/160/FDIS et 65E/164/RVD.

Le rapport de vote 65E/164/RVD donne toute information sur le vote ayant abouti à l'approbation de cette norme.

La version française n'a pas été soumise au vote.

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

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

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

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

IMPORTANT – Le logo "colour inside" qui se trouve sur la page de couverture de cette publication indique qu'elle contient des couleurs qui sont considérées comme utiles à une bonne compréhension de son contenu. Il convient, par conséquent, que les utilisateurs impriment ce document au moyen d'une imprimante couleur.

INTRODUCTION

La présente Norme internationale est une spécification destinée aux développeurs d'applications OPC UA. La spécification est le résultat d'un processus d'analyse et de conception visant à développer une interface normalisée qui facilite l'interopérabilité d'applications issues de divers fournisseurs.

ARCHITECTURE UNIFIÉE OPC –

Partie 3: Modèle de l'espace d'adressage

1 Domaine d'application

La présente partie de la CEI 62541 décrit l'*Espace d'Adressage (AddressSpace)* de l'Architecture unifiée OPC (OPC UA) ainsi que les *Objets* correspondants. Il s'agit du métamodèle OPC UA sur lequel se fondent les modèles d'information OPC UA.

2 Références normatives

Les documents de référence suivants sont indispensables à l'application 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).

CEI/TR 62541-1, *Architecture unifiée OPC – Partie 1: Aperçu et Concepts*

CEI 62541-4[1], *Architecture unifiée OPC – Partie 4: Services*

CEI 62541-5[1], *Architecture unifiée OPC – Partie 5: Modèle de l'information*

CEI 62541-6[1], *Architecture unifiée OPC – Partie 6: Correspondances*

CEI 62541-8[1], *Architecture unifiée OPC – Partie 8: Accès aux données*

ISO/CEI 10918-1, *Technologies de l'information – Compression numérique et codage des images fixes de nature photographique: Prescriptions et lignes directrices*

ISO/CEI 15948, *Technologies de l'information – Infographie et traitement d'images – Graphiques de réseau portables (PNG): Spécification fonctionnelle:*

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

Partie 1 du Schéma XML, disponible à l'adresse <http://www.w3.org/TR/xmlschema-1/>

Partie 2 du Schéma XML, disponible à l'adresse <http://www.w3.org/TR/xmlschema-2/>

XPATH, disponible à l'adresse <http://www.w3.org/TR/xpath/>

IETF RFC 3066, *Etiquettes d'Identification des Langues,* disponible à l'adresse <http://tools.ietf.org/html/rfc3066>

IEEE 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic*, disponible à l'adresse <http://ieeexplore.ieee.org/servlet/opac?punumber=2355>

[1] A publier.

3 Termes, définitions, abréviations et conventions

3.1 Termes et définitions

Pour les besoins du présent document, les termes et définitions suivants, ainsi que ceux donnés dans la CEI/TR 62541-1, s'appliquent.

3.1.1
DataType
instance d'un *DataType Node (Nœud* de *Type de Données)* qui est utilisée avec l'*attribut ValueRank* (*Rang de Valeur*) pour définir le type de données d'une *Variable*

3.1.2
DataVariable
variables qui représentent des *valeurs* d'*Objets*, que ce soit directement ou indirectement pour des *Variables* complexes, les *Variables* étant toujours le *TargetNode (Nœud Cible)* d'une *Référence " HasComponent " (Dispose d'un composant)*

3.1.3
EventType
ObjectType (*Nœud de Type d'Objet*) représentant la définition du type d'un *Evénement* donné

3.1.4
référence hiérarchique
Référence qui est utilisée pour construire des hiérarchies dans *AddressSpace* (*Espace d'Adressage)*

NOTE Tous les *Types de Références* hiérarchiques sont issus des *Références Hiérarchiques*.

3.1.5
InstanceDeclaration
Nœud utilisé par une *(TypeDefinitionNode)* complexe pour exposer la complexité de sa structure; cette instance est utilisée par une définition de type

3.1.6
ModellingRule
métadonnée d'une *Déclaration d'Instance* qui définit la manière dont la *Déclaration d'Instance* sera utilisée pour l'instanciation; elle définit également les règles de sous-typage d'une *Déclaration d'Instance*

3.1.7
property
Variables qui constituent le *Nœud Cible* d'une *Référence "Dispose d'une Propriété";* les *propriétés* décrivent les caractéristiques d'un *Nœud*

3.1.8
SourceNode
nœud qui sert de *référence* à un autre *Nœud*; par exemple, dans la *Référence* “A contient B”, “A” est le *Nœud Source*

3.1.9
TargetNode
nœud qui est référencé par un autre *Nœud;* par exemple, dans la *Référence* “A contient B”, “B” est le *Nœud Cible*

3.1.10
TypeDefinitionNode
nœud utilisé pour définir le type d'un autre *Nœud*; les *Nœuds Type d'Objet* et *Type de Variable* sont des *Nœuds de Définition de Type*

3.1.11
VariableType
nœud représentant la définition du type d'une *Variable*

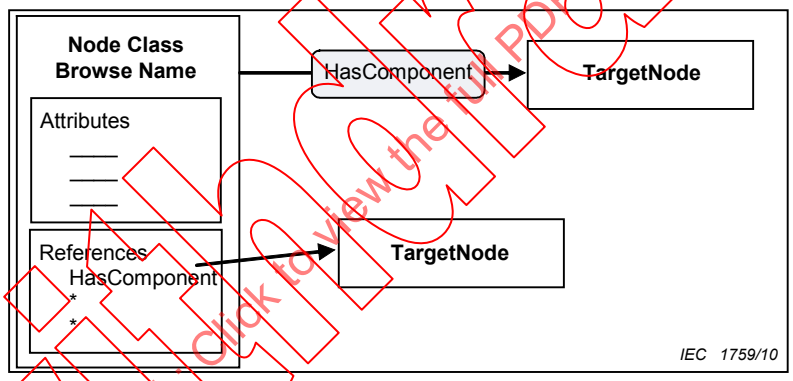
3.2 Abréviations

UA Architecture unifiée (de l'anglais Unified Architecture)
UML Langage de modélisation unifié (de l'anglais Unified Modeling Language)
URI Identificateur de ressources uniformes (de l'anglais Uniform Resource Identifier)
W3C World Wide Web Consortium (consortium chargé de la normalisation du web mondial)
XML Langage de balisage extensible (de l'anglais eXtensible Markup Language)

3.3 Conventions

3.3.1 Conventions pour les figures Espace d'Adressage

Les *Nœuds* et leurs *Références* respectives sont illustrés par des figures. La Figure 1 représente les conventions utilisées dans ces figures.



Légende

Anglais	Français
NodeClass Browse Name	Nom d'exploration de classe de nœud
HasComponent	Dispose d'un composant
TargetNode	Nœud cible
Attributes	Attributs
References	Références

Figure 1 – Diagrammes des Nœuds de l'Espace d'Adressage

Sur ces figures, les rectangles représentent les *Nœuds*. L'intitulé des rectangles représentant les *Nœuds* peut être donné par une ou deux lignes de texte. Lorsque deux lignes sont utilisées dans le rectangle, la première identifie la *Classe* et la seconde donne le *Nom d'Exploration.* Lorsqu'une seule ligne est utilisée, elle contient le *Nom d'Exploration*.

Les *rectangles représentant les Nœuds* peuvent inclure des cases qui définissent leurs *Attributs* et les *Références*. Des dénominations spécifiques dans ces cases identifient des *Attributs* et des *Références* spécifiques.

Des rectangles grisés aux bords arrondis et traversés par des flèches représentent des *Références*. Cette flèche part du *SourceNode* et pointe vers le *Nœud Cible*. Les *Références* peuvent également être illustrées en traçant une flèche partant de la dénomination de la *Référence* dans la case “Références” et aboutissant au *Nœud Cible*.

3.3.2 Conventions pour la définition des Classes de Nœuds

L'Article 5 définit les *Classes de Nœuds* de l'*Espace d'Adressage*. Le Tableau 1 décrit le format des tableaux utilisés pour définir les *Classes de Nœuds*.

Tableau 1 – Conventions utilisées dans les Tableaux de Classes de Nœuds

Dénomination	Usage	Type de Données	Description
Attributs			
“Dénomination de l'attribut”	“M” ou “O”	Type de Données de l'*Attribut*	Définit l'*Attribut*.
Références			
“Dénomination de la référence”	“1”, “0..1” ou “0..*”	Non utilisé	Décrit l'utilisation de la *Référence* par la *Classe de Nœud*.
Propriétés normalisées			
“Dénomination de la Propriété”	“M” ou “O”	Type de Données de la *Propriété*	Définit la *Propriété*.

La colonne Dénomination donne le nom de l'*Attribut*, le nom du *Type de Référence* utilisé pour créer une *Référence* ou le nom d'une *Propriété* référencée par la *Référence " HasProperty " (Dispose d'une Propriété)*.

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 être présent pour tout *Nœud* de la *Classe de Nœud*. Pour les *Références,* cette colonne précise la cardinalité. Les valeurs suivantes peuvent s'appliquer:

- “0..*” indique une absence totale de restrictions; en d'autres termes il n'est pas nécessaire de fournir la *Référence,* mais il n'y a par ailleurs aucune limite quant au nombre de fois où elle est 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 donne la dénomination du *Type de Données* 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*.

Seule la présente norme peut définir des *Attributs*. De ce fait, tous les *Attributs* de la *Classe de Nœud* sont spécifiés dans le tableau et ne peuvent être étendus que par d'autres parties de la présente série de normes.

La présente norme définit également des *Types de Références;* ces derniers ne peuvent cependant être spécifiés que par un serveur ou par un client au moyen des *Services de Gestion de Nœuds* décrits dans la CEI 62541-4. Ainsi, les tableaux de *Classes de Nœuds* de la présente norme peuvent contenir les *Types de Références* de base, appelés *Références,* indiquant de ce fait que tout *Type de Référence* peut être utilisé pour la *Classe de Nœud,* y compris des *Types de Références* spécifiques au système. Les tableaux de *Classes de Nœuds* spécifient uniquement la manière dont les *Classes de Nœuds* peuvent être utilisées en tant que *Nœuds Sources* de *Références*, et non en tant que *Nœuds Cibles*. Si un tableau de *Classe de Nœud* permet l'utilisation d'un *Type de Référence* de sa *Classe de Nœud*

comme *Nœud Source*, ceci est également vrai pour des sous-types du *Type de Référence*. Des sous-classes du *Type de Référence* peuvent toutefois restreindre ses *Nœuds Sources*.

La présente norme définit des *Propriétés*, mais d'autres organismes de normalisation ou des fournisseurs peuvent aussi définir des *Propriétés*; il est également possible d'avoir des *Nœuds* dont les *Propriétés* ne sont pas normalisés. Les *Propriétés* établies dans la présente norme sont définies par leur dénomination, qui est mise en correspondance avec le *Nom d'Exploration* d'*Indice de l'espace de Nom* 0; celui-ci représente l'*Espace de Nom* pour l'OPC UA.

La colonne "usage" (facultatif ou obligatoire) n'implique pas une *Règle de Modélisation* spécifique pour modéliser les *Propriétés*. Chaque application de serveur particulier choisira d'utiliser les *Règles de Modélisation* qui lui conviennent.

4 Concepts de l'Espace d'Adressage

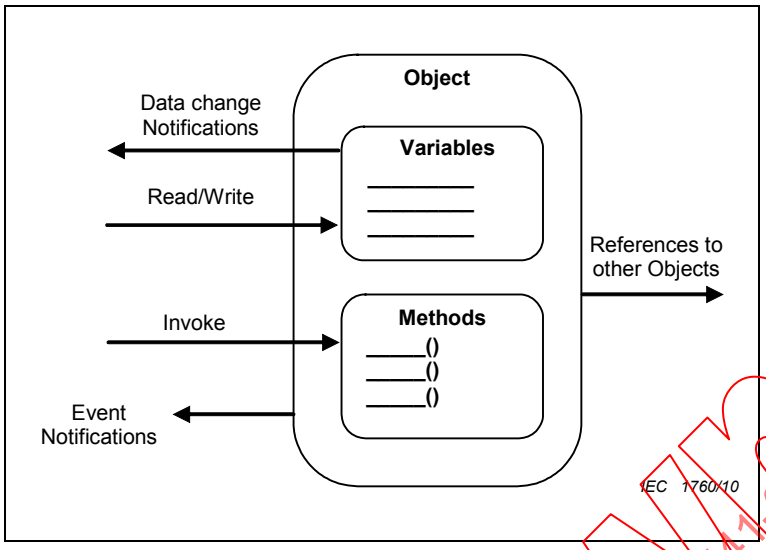
4.1 Aperçu général

Les paragraphes suivants définissent les concepts de l'*Espace d'Adressage*. L'Article 5 définit les *Classes de Nœuds* de l'*Espace d'Adressage* représentant les concepts de l'espace *d'Adressage*. L'Article 6 décrit en détail le modèle type des *Types d'Objets* et des *Types de Variables*. Les Articles 7 à 9 définissent des types normalisés de *Types de Références*, *Types de Données* et *Types d'Evénements*.

L'Annexe A informative donne des considérations d'ordre général quant à la manière d'utiliser le modèle de l'espace d'adressage et l'Annexe B informative fournit un modèle en UML du modèle de l'espace d'adressage. L'Annexe C normative définit le système de description de types OPC binaire comme un format permettant de spécifier des structures de types de données et l'Annexe D normative définit une notation graphique des données d'OPC UA.

4.2 Modèle d'objet

L'objectif premier de l'*Espace d'Adressage* d'OPC UA est de mettre à disposition une méthode normalisée permettant aux serveurs de représenter des *Objets* pour leurs clients. C'est dans ce but qu'a été conçu le Modèle Objet d'OPC UA. Il définit des *Objets* en termes de *Variables* et de *Méthodes*. Il permet également d'exprimer les relations entre *Objets*. Ce modèle est illustré en Figure 2.



Légende

Anglais	Français
Object	Objet
Data change notifications	Notifications de modification de données
Variables	Variables
Read/write	Lecture/écriture
References to other objects	Références à d'autres objets
Invoke	Invocation
Methods	Méthodes
Event notifications	Notifications d'événements

Figure 2 – Modèle Objet d'OPC UA

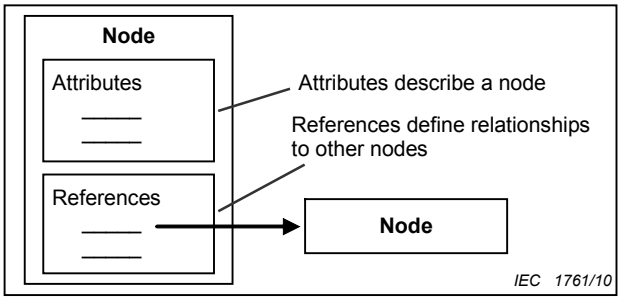
Dans l'*Espace d'Adressage* les éléments de ce modèle sont représentés comme des *Nœuds*. Chaque *Nœud* est attribué à une *Classe de Nœud* et chaque *Classe de Nœud* représente un élément différent du Modèle d'Objet. L'Article 5 définit les *Classes de Nœuds* utilisées pour représenter ce modèle.

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 est appelé son *Espace d'Adressage*. Le modèle utilisé à cet effet est défini par le Modèle d'Objet d'OPC UA (voir 4.2).

Les Objets et leurs composants sont représentés dans l'*Espace d'Adressage* comme un ensemble de *Nœuds* décrits par des *Attributs* et interconnectés par des *Références*. La Figure 3 illustre le modèle d'un *Nœud* et le présent paragraphe en décrit les détails.



Légende

Anglais	Français
Node	Nœud
Attributes	Attributs
Attributes describe a node	Les Attributs décrivent un nœud
References define relationships to other nodes	Les Références définissent des relations avec d'autres nœuds
References	Références

Figure 3 – Modèle de Nœud de l'espace d'Adressage

4.3.2 Classes de Nœuds

Les *Classes de Nœuds* sont définies en termes d'*Attributs* et de *Références* qui doivent être instanciés (recevoir des valeurs) lorsqu'un *Nœud* est défini dans l'*Espace d'Adressage*. Les *Attributs* sont traités en 4.3.3 et les *Références* en 4.3.4.

L'Article 5 définit les *Classes de Nœuds* utilisées pour l'*Espace d'Adressage* d'OPC UA. Ces *Classes de Nœuds* sont collectivement appelées les métadonnées de l'*Espace d'Adressage*. Chaque *Nœud* dans l'*Espace d'Adressage* est une instance de l'une de ces *Classes de Nœuds*. Aucune autre *Classe de Nœud* ne doit être utilisée pour définir des *Nœuds* et, par conséquent, les clients et les serveurs ne sont pas autorisés à définir des *Classes de Nœuds* ou à étendre les définitions de ces *Classes de Nœuds*.

4.3.3 Attributs

Les *Attributs* sont des éléments de données qui décrivent des *Nœuds*. Les clients peuvent accéder à des valeurs d'*Attributs* en utilisant des *Services* de Lecture, d'Ecriture, de Requête, et d'Abonnement/Article surveillé. Ces *Services* sont définis dans la CEI 62541-4.

Les *Attributs* sont des composants élémentaires de *Classes de Nœuds*. Les définitions des *Attributs* font partie intégrante de celles des *Classes de Nœuds* de l'Article 5 et, par conséquent, elles ne sont pas incluses dans l'*Espace d'Adressage*.

Chaque définition d'*Attribut* comprend un identifiant d'attribut (pour les identifiants d'*Attributs*, voir la CEI 62541-6), une dénomination, une description, un type de données et un indicateur obligatoire/facultatif. L'ensemble d'*Attributs* défini pour chaque *Classe de Nœud* ne doit pas être étendu par les clients ou les serveurs.

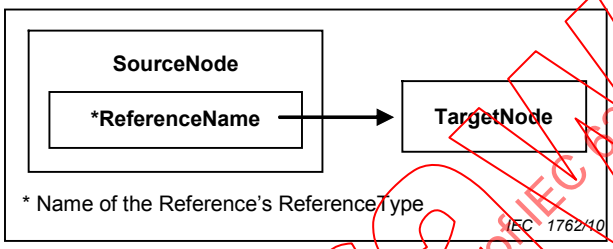
Lorsqu'un *Nœud* est instancié dans l'*Espace d'Adressage*, les valeurs des *Attributs* de la *Classe de Nœud* sont fournies. L'indicateur obligatoire/facultatif de l'*Attribut* indique s'il est nécessaire d'instancier l'*Attribut*.

4.3.4 Références

Les *Références* sont utilisées pour corréler les *Nœuds* entre eux. On peut les accéder en utilisant les *Services* exploration et interrogation définis dans la CEI 62541-4.

De la même manière que les *Attributs*, elles sont définies comme des composants fondamentaux des *Nœuds*. Contrairement aux *Attributs*, les *Références* sont définies comme des instances des *Nœuds de Types de Références*. Les *Nœuds* de *Types de Références* sont visibles dans l'*Espace d'Adressage* et sont définis en utilisant la *Classe de Nœud "Types de Références"* (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 *Type de Référence* 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 *Type de Référence*. Tous les sous-types de *Types de Références* concrets sont considérés égaux aux *Types de Références* concrets de base lorsqu'il s'agit d'identifier des *Références* (voir 5.3 pour les sous-types de *Types de Références*). Ce modèle de *Référence* est illustré en Figure 4.



Légende

Anglais	Français
Sourcenode	Nœud Source
Referencename	Nom de référence
Targetnode	Nœud cible
Name of the reference's referencetype	Dénomination du Type de Référence de la Référence

Figure 4 – Modèle de référence

Le *TargetNode* d'une *Référence* peut être situé dans le même *Espace d'Adressage* ou dans l'*Espace d'Adressage* d'un autre serveur OPC UA. Les *Nœuds Cibles* 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'identifiant 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*. Il est défini deux types de *Variables:* les *Propriétés* et les *Variables de données*. Elles diffèrent par le type de données qu'elles représentent et par le fait qu'elles contiennent ou non d'autres *Variables*.

4.4.2 Propriétés

Les *Propriétés* sont des caractéristiques d'*Objets*, de *Variables de données* et d'autres *Nœuds* définis par le serveur. Les *Propriétés* sont différentes des *Attributs* du fait qu'elles caractérisent *ce que* représente le *Nœud*, par exemple un équipement ou un bon de commande. Les *Attributs* définissent des métadonnées supplémentaires qui sont instanciées pour tous les *Nœuds* à partir d'une *Classe de Nœud.* Les *Attributs* sont communs à tous les *Nœuds* d'une *Classe de Nœud* donnée et ne sont définis que par cette spécification tandis que les *Propriétés* peuvent être définies par le serveur.

Par exemple, un *Attribut* définit le *Type de Données* des *Variables* tandis qu'une *Propriété* peut être utilisée pour spécifier l'unité technique de certaines *Variables*.

Pour éviter les répétitions, il n'est pas admis de définir des *Propriétés* par d'autres *Propriétés*. Pour identifier facilement des *Propriétés*, le *Nom d'Exploration* d'une *Propriété* doit être unique dans le contexte d'un *Nœud* contenant les *Propriétés* (pour plus de détails, voir 5.6.3).

Un *Nœud* et ses *Propriétés* doivent toujours résider dans le même serveur.

4.4.3 Variables de données

Les *Variables de données* représentent le contenu d'un *Objet*. Il peut être par exemple défini par un *Objet* fichier qui contient un train d'octets. Le train d'octets peut être défini comme une *Variable de Données* qui est une suite 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 “heure de début” et “heure de fin”, elle pourrait disposer d'une *Propriété* spécifique à cette structure de données comme “heure de début au plus tôt”.

Comme autre exemple, des blocs fonctionnels des systèmes de commande pourraient être représentés comme des *Objets*. Les paramètres du bloc fonctionnel tels que ses points de consigne, pourraient être représentés comme des *Variables de Données*. L'*Objet* bloc fonctionnel pourrait également disposer de *Propriétés* décrivant son temps d'exécution et son type.

Les *Variables de Données* peuvent avoir des *Variables de Données* supplémentaires, mais uniquement si elles sont complexes. Dans ce cas, leurs *Variables de Données* doivent toujours être des éléments de définition complexe. En suivant l'exemple donné dans la description des *Propriétés* en 4.4.2, le serveur pourrait présenter l'“heure de début” et l'“heure de fin” en tant que composants séparés de la structure de données.

Comme autre exemple, une *DataVariable* complexe peut définir un ensemble de valeurs de température générées par trois transmetteurs de température séparés mais tous visibles dans l'*Espace d'Adressage*. Dans ce cas, cette *DataVariable* complexe pourrait, à partir de là, définir des *Références "Dispose d'un composant"* pour les différentes valeurs de température dont elle est constituée.

4.5 TypeDefinitionNodes

4.5.1 Généralités

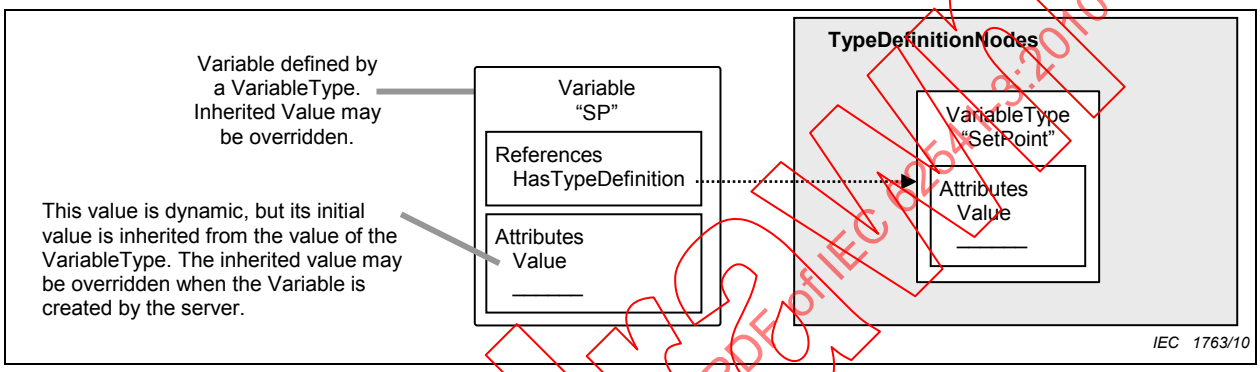
Les serveurs OPC UA doivent fournir des définitions type pour les *Objets* et les *Variables*. La *Référence " HasTypeDefinition "* (*Dispose d'une Définition Type*) doit être utilisée pour relier une instance à sa définition représentée par un *TypeDefinitionNodes*. Les définitions types sont cependant exigées et la CEI 62541-5 définit un *Type d'Objet de Base*, un *Type de Propriété* et un *Type de DataVariable de Base* de sorte qu'un serveur puisse utiliser un type de base si aucune information de type plus spécialisée n'est disponible. Les *Objets* et *Variables* héritent des *Attributs* spécifiés par leur *TypeDefinitionNodes* (pour plus de détails, voir 6.4).

Dans certains cas, l'*Identifiant de Nœud* utilisé par la *Référence "Dispose d'une Définition de Type"* sera bien connu des clients et des serveurs. Les différents organismes peuvent définir des *Nœuds de Définition de Type* bien connus dans l'industrie. Les *Identifiants de Nœuds* des *Nœuds de Définition de Type* bien connus permettent de généraliser ce terme sur l'ensemble des serveurs OPC UA et les clients peuvent ainsi interpréter le *TypeDefinitionNodes* sans avoir à le lire à partir du serveur. Par conséquent, les serveurs peuvent utiliser des *Identifiants de Nœuds* bien connus sans avoir à représenter les *Nœuds de Définition de Type* correspondants dans leur *Espace d'Adressage*. Cependant, pour les clients génériques, les

Nœuds de Définition de Type doivent être fournis. Ces *Nœuds de Définition de Type* peuvent exister dans un autre serveur.

L'exemple suivant, illustré en Figure 5, décrit l'utilisation d'une *Référence "Dispose d'une Définition de Type"*. Dans cet exemple, un paramètre de point de consigne “PC” est représenté par une *DataVariable* dans l'*Espace d'Adresse*. Cette *DataVariable* fait partie d'un *Objet* qui n'est pas illustré sur la figure.

Pour fournir une définition de point de consigne commune qui peut être utilisée par d'autres *Objets*, on emploie un *Type de Variable* spécialisé. Chaque *DataVariable* de point de consigne qui utilise cette définition commune aura une *Référence "Dispose d'une Définition de Type"* qui identifie le *Type de Variable* du “point de consigne” commun.



Légende

Anglais	Français
Typedefinitionnodes	Nœuds de Définition de Type
Variable defined by a variabletype. Inherited value may be overriden	Variable définie par un Type de Variable. La valeur héritée peut être écrasée et remplacée
Variable “SP”	Variable PC
References Hastypedefinition	Références « Dispose d'une Définition de Type »
Variabletype “setpoint”	Type de Variable « Point de Consigne »
This value is dynamic but its initial value is inherited from the value of the variabletype. The inherited value may be overridden when the variable is created by the server	Cette valeur est dynamique mais sa valeur initiale est héritée de la valeur du Type de Variable. La valeur héritée peut être écrasée et remplacée lorsque la variable est créée par le serveur
Attributes value	Attributs Valeur

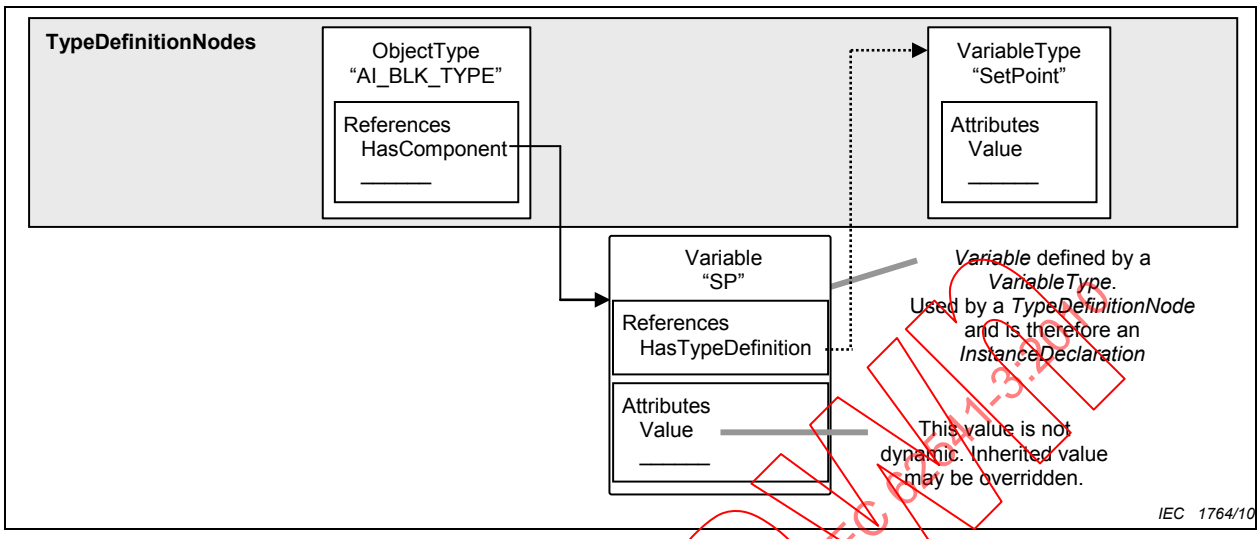
Figure 5 – Exemple d'une Variable définie par un Type de Variable

4.5.2 *TypeDefinitionNodes* de Type complexes et leurs Déclarations d'Instances

Les *TypeDefinitionNodes* peuvent être complexes. Un *TypeDefinitionNodes* complexe définit également des *Références* à d'autres *Nœuds* dans le cadre de la définition de type. Les *Règles de Modélisation* 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 *TypeDefinitionNodes* référence des instances plutôt que d'autres *Nœuds de Définition de Type,* afin que des noms uniques soient utilisés pour plusieurs instances du même type, pour définir des valeurs par défaut et ajouter des *Références* à d'autres instances spécifiques à ce *TypeDefinitionNodes* complexe et non au *TypeDefinitionNodes* de l'instance. Par exemple, dans la Figure 6, le *Type d'Objet* “AI_BLK_TYPE”, qui représente un bloc fonctionnel, dispose d'une *Référence "Dispose d'un composant"* à une *Variable* “PC” du *Type de Variable* “Point de Consigne”. Le “AI_BLK_TYPE” pourrait avoir une *Variable* de point de consigne supplémentaire du même type utilisant un nom différent. Il pourrait être ajouté à la *Variable*

une *Propriété* qui n'était pas définie par son *TypeDefinitionNodes* “Point de Consigne”. Et il pourrait également définir une valeur par défaut pour “PC”, de sorte que chaque instance de “AI_BLK_TYPE” aurait une *Variable* “PC” initialement réglée sur cette valeur.



Légende

Anglais	Français
Typedefinitionnodes	Nœuds de Définition de Type
Objecttype	Type d'Objet
Variabletype	Type de Variable
References Hascomponent	Références « Dispose d'un composant »
Attributes value	Attributs Valeur
Variable “SP”	Variable « PC »
References Hastypedefinition	Références « Dispose d'une Définition de Type »
Variable defined by a variabletype. Used by a typedefinitionnode and is therefore an instancedeclaration	Variable définie par un Type de Variable. Utilisée par un *TypeDefinitionNodes* et est par conséquent une Déclaration d'Instance
This value is not dynamic. Inherited value may be overriden	Cette valeur n'est pas dynamique. La valeur héritée peut être écrasée et remplacée

Figure 6 – Exemple d'une Définition de Type 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 contenant. En d'autres termes, de manière générale, le constructeur de la classe composante vient en premier, suivi par le constructeur de la classe contenant. Il est possible qu'un constructeur de classe contenant prenne le pas sur les valeurs de composants définis par la classe composante.

Afin de faire la différence entre les instances utilisées pour les définitions de types et celles qui représentent des données réelles, ces instances sont appelées *Déclarations d'Instances*. Ce terme est cependant utilisé pour simplifier la présente spécification; on ne peut savoir si une instance est ou n'est pas une *Déclaration d'Instance* que dans l'*Espace d'Adressage,* en suivant ses *Références*. Certaines instances peuvent être partagées et par conséquent référencées par des *Nœuds de Définition de Type*, des *Déclarations d'Instances* et des instances. Cette méthodologie est similaire aux variables de classes dans les langages de programmation orientés objet.

4.5.3 Sous-typage

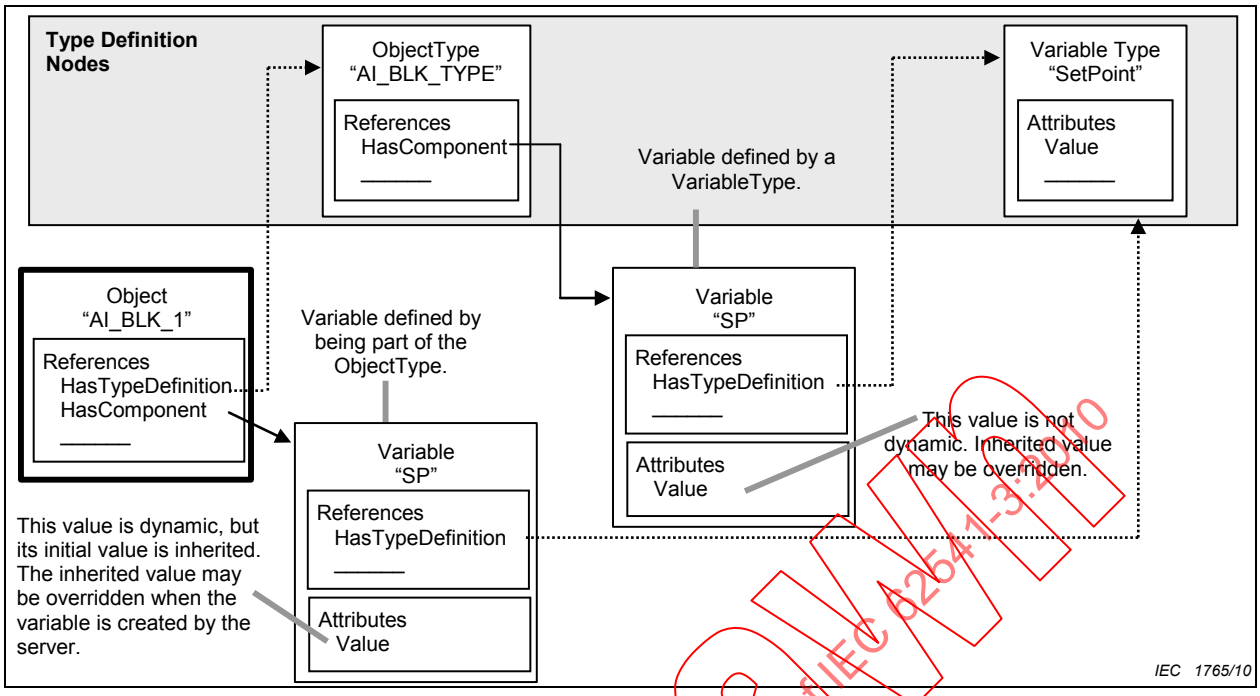
La présente norme permet l'utilisation de sous-types de définitions de types. Les règles de sous-typage sont définies à l'Article 6. Le sous-typage de *Types d'Objets* et *Types de Variables* permet:

- aux clients qui connaissent uniquement le supertype, de traiter une instance du sous-type comme s'il s'agissait d'une instance du supertype;
- de remplacer des instances du supertype 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 supertype mais peuvent apporter des caractéristiques supplémentaires. Par exemple, un fournisseur peut souhaiter étendre un *Type de Variable* général "Capteur de Température" en ajoutant une *Propriété* indiquant la périodicité de maintenance. Pour cela, le fournisseur crée un nouveau *Type de Variable* qui est un *TargetNode* pour une référence *« HasSubtype»* à partir du *Type de Variable* initial et y ajoute la nouvelle *Propriété*.

4.5.4 Instanciation de *TypeDefinitionNodes* de Type complexes

L'instanciation des *Nœuds de Définition de Type* complexes dépend des *Règles de Modélisation* définies en 6.4.4. Il est cependant prévu que des instances d'une définition de type reflètent la structure définie par le *Nœud de Définition de Type*. La Figure 7 présente une instance du *TypeDefinitionNodes* "AI_BLK_TYPE", dans lequel la *Règle de Modélisation Obligatoire*, définie en 6.4.4.5.2, a été appliquée à sa *Variable* contenant. Ainsi, une instance de "AI_BLK_TYPE", appelée AI_BLK_1", dispose d'une *Référence "Dispose d'une Définition de Type"* vers "AI_BLK_TYPE". Elle contient également une *Variable* "PC" ayant le même *Nom d'Exploration* que la *Variable* "PC" utilisée par le *TypeDefinitionNodes* et par conséquent reflète la structure définie par le *Nœud de Définition de Type*.



Légende

Anglais	Français
Typedefinitionnodes	Nœuds de Définition de Type
Objecttype	Type d'Objet
References Hascomponent	Références « Dispose d'un composant »
Variable type "Setpoint"	Type de Variable « Point de Consigne »
Variable defined by a variabletype.	Variable définie par un Type de Variable
Variable "SP"	Variable « PC »
Object	Objet
Variable defined by being part of the objecttype	Variable définie du fait qu'elle fait partie du Type d'Objet
References Hastypedefinition Hascomponent	Références « Dispose d'une Définition de Type » « Dispose d'un composant »
This value is dynamic but its initial value is inherited. The inherited value may be overridden when the variable is created by the server	Cette valeur est dynamique mais sa valeur initiale est héritée. La valeur héritée peut être écrasée et remplacée lorsque la variable est créée par le serveur
Attributes value	Attributs Valeur
This value is not dynamic. Inherited value may be overriden	Cette valeur n'est pas dynamique. La valeur héritée peut être écrasée et remplacée

Figure 7 – L'Objet et ses composants définis par un Type d'Objet

Un client connaissant le *Type d'Objet* "AI_BLK_TYPE" peut utiliser cette information pour explorer directement les *Nœuds* contenants de chaque instance de ce type. Ceci permet une programmation tenant compte du *Nœud de Définition de Type*. Par exemple, un élément graphique peut être programmé dans le client qui traite toutes les instances de "AI_BLK_TYPE" de la même manière qu'en présentant la valeur de "PC".

Une programmation selon le *TypeDefinitionNodes* présente une ou plusieurs contraintes. Un *TypeDefinitionNodes* ou une *Déclaration d'Instance* ne doit jamais référencer deux *Nœuds* ayant un même *Nom d'Exploration* en utilisant les *Références hiérarchiques* dans le sens direct. Les instances fondées sur des *Déclarations d'Instances* doivent toujours conserver le

même *Nom d'Exploration* que la *Déclaration d'Instance* dont elles résultent. Un *Service* spécial défini dans la CEI 62541-4, appelé Traduire les Chemins d'Exploration en Identifiants de Nœuds (TranslateBrowsePathsToNodeIds), peut être utilisé pour identifier les instances sur la base des *Déclarations d'Instances*. L'utilisation du *Service d'exploration* simple pourrait ne pas suffire car l'unicité du *Nom d'Exploration* n'est exigée que pour les *Nœuds de Définition de Type* et les *Déclarations d'Instances*, et non pour d'autres instances. Ainsi, "AI_BLK_1" pourrait disposer d'une autre *Variable* portant le *Nom d'Exploration* "PC", même si celle-ci ne résulte pas d'une *Déclaration d'Instance* du *Nœud de Définition de Type*.

Les instances dérivées d'une *Déclaration d'Instance* doivent avoir le même *TypeDefinitionNodes* ou un sous-type de ce *TypeDefinitionNodes*.

Un *TypeDefinitionNodes* et ses *Déclarations d'Instances* doivent toujours appartenir au même serveur. Cependant, des instances peuvent pointer, avec leur *Référence "Dispose d'une Définition de Type"*, vers un *TypeDefinitionNodes* 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 du système et les erreurs du système sont des exemples d'*Evénements*. Les *Notifications d'Evénements* rendent compte de l'occurrence d'un *Evénement* donné. Les *Evénements* définis dans le présent document ne sont pas directement visibles dans l'*Espace d'Adresse* d'OPC UA. Des *Objets* et des *Vues* peuvent être utilisés pour s'abonner à des *Evénements*. L'*Attribut "Notificateur d'Evénements"* indique si le *Nœud* peut s'abonner à des *Evénements*. Les clients s'abonnent à ces *Nœuds* afin de recevoir des *Notifications* d'occurrences d'*Evénements*.

L' *Abonnement aux Evénements* utilise les *Services* Surveillance et Abonnement définis dans la CEI 62541-4 pour s'abonner aux *Notifications d'Evénements* d'un *Nœud*.

Tout serveur OPC UA qui prend en charge le traitement des événements doit présenter au moins un *Nœud* comme *Notificateur d'Evénements*. L'*Objet Serveur* défini dans la CEI 62541-5 est utilisé à cet effet. Les *Evénements* générés par le serveur sont disponibles par l'intermédiaire de cet *Objet Serveur*. Il n'est pas prévu qu'un serveur produise 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'*Espace d'Adresse*. Ces *Nœuds* (identifiés par l'*Attribut "Notificateur d'Evénements"*) fournissent un sous-ensemble des *Evénements* générés par le serveur. La position dans l'*Espace d'Adresse* indique ce que ce sous-ensemble sera. Par exemple, un *Objet* zone de processus, représentant une zone fonctionnelle du processus, fournirait des *Evénements* provenant uniquement de cette zone du processus. A noter qu'il s'agit seulement d'un exemple et qu'il incombe entièrement au serveur de déterminer les *Evénements* que tel ou tel *Nœud va fournir*.

4.6.2 Types d'Evénements

Chaque *Evénement* correspond à un *Type d'Evénement* spécifique. Un serveur peut prendre en charge plusieurs types. Cette partie définit le *Type d'Evénement de Base* dont dérivent tous les autres *Types d'Evénements*. Il est prévu que d'autres spécifications associées définissent des *Types d'Evénements* supplémentaires dérivant des types de base définis dans la présente partie.

Les *Types d'Evénements* pris en charge par un serveur sont présentés dans l'*Espace d'Adressage* du serveur. Les *Types d'Evénements* sont représentés comme des *Types d'Objets* dans l'*Espace d'Adressage* et il ne leur est pas associé de *Classe de Nœud* spéciale. La CEI 62541-5 définit de manière détaillée la manière dont un serveur présente les *Types d'Evénements.*

Les *Types d'Evénements* définis dans ce document sont spécifiés comme étant abstraits et par conséquent ils ne sont jamais instanciés dans l'*Espace d'Adressage*. Les occurrences d'Evénements de ces *Types d'Evénements* ne sont présentées que par l'intermédiaire d'un *Abonnement*. Il existe dans l'*Espace d'Adressage* des *Types d'Evénements* qui permettent aux clients de découvrir le *Type d'Evénement*. Cette information est utilisée par un client lorsqu'il établit et utilise des *Abonnements* aux *Evénements*. Les *Types d'Evénements* définis par d'autres parties de cette norme ou par des spécifications associées, ainsi que les *Types d'Evénements* spécifiques au serveur, peuvent être définis comme non abstraits et par conséquent les instances de ces *Types d'Evénements* peuvent être visibles dans l'*Espace d'Adressage* même si les *Evénements* de ces *Types d'Evénements* sont également accessibles par l'intermédiaire des mécanismes de *Notification* d'*Evénements*.

Les *Types d'Evénements* normalisés sont décrits à l'Article 9. Leur représentation dans l'*Espace d'Adressage* est spécifiée dans la CEI 62541-5.

4.6.3 Catégorisation des Evénements

Les *Evénements* peuvent être classés par catégories en créant de nouveaux *Types d'Evénements* qui sont des sous-types de *Types d'Evénements* existants mais qui n'étendent pas un type existant. Ils sont uniquement utilisés pour identifier un événement comme appartenant au nouveau *Type d'Evénement*. Par exemple, le *Type d'Evénement* "Type d'Evénement de Défaillance d'un Appareil" pourrait avoir comme sous-type un "Type d'Evénement de Défaillance de Transmetteur" et un "Type d'Evénement de Défaillance d'Ordinateur". Ces nouveaux sous-types ne seraient pas intégrés comme de nouvelles *Propriétés* ou ne modifieraient pas la sémantique héritée du Type d'Evénement de Défaillance d'un Appareil; il s'agit purement et simplement d'une catégorisation des *Evénements*.

Les sources d'*Evénements* peuvent également être organisées en groupes, en utilisant les *Types de Références* d'*Evénements* décrits en 7.18 et 7.20. Par exemple, un serveur peut définir des *Objets* dans l'*Espace d'Adressage* représentant des *Evénements* liés aux appareils proprement dits ou des zones d'*Evénements* d'une installation ou une fonctionnalité contenue dans le serveur. Les *Références d'Evénements* seraient utilisées pour indiquer quelles sont les sources d'*Evénements* qui représentent des appareils concrets et celles qui représentent une certaine fonctionnalité basée dans le serveur. En outre, des *Références* peuvent être utilisées pour grouper les appareils proprement dits ou les fonctionnalités basées dans 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 seraient établies. Pour des exemples supplémentaires, voir la description des *Types de Références d'Evénements*.

Les clients peuvent sélectionner une ou plusieurs catégorie d'*Evénements* en définissant des filtres de contenu qui incluent des termes spécifiant le Type *d'Evénement* de l'*Evénement* ou un groupement de sources d'*Evénements*. Ces deux mécanismes permettent de catégoriser de plusieurs manières un *Evénement* particulier. Un client pourrait obtenir tous les *Evénements* correspondant à un appareil concret 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), comme pour les méthodes d'une classe donnée en programmation orientée objet ou un *Type d'Objet* appartenance, comme dans les 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 *Méthodes*

commence lorsque le client fait appel à la *Méthode* et se termine lorsque le résultat lui est renvoyé.

NOTE L'*Objet* ou le *Type d'Objet* appartenance est spécifié dans l'appel au 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 *Classe de Nœud "Méthodes"*. Ce *Nœud* contient les métadonnées qui identifient les arguments de la *Méthode* et décrit son comportement.

Les *Méthodes* sont invoquées par le *Service* "*Invocation"* défini dans la CEI 62541-4. Chaque *Méthode* est invoquée dans le contexte d'une session existante. Si la session se termine pendant l'exécution de la *Méthode*, les résultats correspondants ne peuvent pas être renvoyés au client et sont rejetés. Dans ce cas, l'exécution de la Méthode est non définie, ce qui veut dire que la *Méthode* peut être exécutée jusqu'à ce qu'elle s'achève ou bien qu'elle soit abandonnée.

Les clients trouvent les *Méthodes* prises en charge par un serveur en explorant les *Références d'Objets* appartenance qui identifient les *Méthodes* prises en charge.

5 Classes de Nœuds normalisées

5.1 Aperçu général

Le présent article définit les *Classes de Nœuds* utilisées pour définir les *Nœuds* dans *l'Espace d'Adressage d'OPC UA*. Les *Classes de Nœuds* sont dérivées d'une *Classe de Nœud de base* commune. On définit en premier lieu cette *Classe de Nœud*, puis celles qui sont utilisées pour organiser l'*Espace d'Adressage* et ensuite les *Classes de Nœuds* utilisées pour représenter les *Objets*.

Les *Classes de Nœuds* 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 type de données. Le paragraphe 6.3 décrit les règles de sous-typage et le paragraphe 6.4 indique les règles d'instanciation des définitions de types.

5.2 Classe de Nœud de base

5.2.1 Généralités

Le modèle de l'espace d'adressage d'OPC UA définit une *Classe de Nœud de Base* dont sont issues toutes les autres *Classes de Nœuds*. Les *Classes de Nœuds* dérivées représentent les divers composants du modèle d'Objet OPC UA (voir 4.2). Les *Attributs* de la *Classe de Nœud* de *Base* sont spécifiés dans le Tableau 2. Aucune *Référence* n'est spécifiée pour la *Classe de Nœud de Base*.

Tableau 2 – Classe de Nœud de base

Dénomination	Usage	Type de Données	Description
Attributs			
NodeId	M	Identifiant de Nœud	Voir 5.2.2
NodeClass	M	Classe de Nœud	Voir 5.2.3
BrowseName	M	Dénomination Qualifiée	Voir 5.2.4
DisplayName	M	Texte Localisé	Voir 5.2.5
Description	O	Texte Localisé	Voir 5.2.6
WriteMask	O	UInt32	Voir 5.2.7
UserWriteMask	O	UInt32	Voir 5.2.8
Références			Aucune *Référence* n'est spécifiée pour cette *Classe de Nœud*

5.2.2 Identifiant de Nœud (NodeId)

Les *Nœuds* sont identifiés sans aucune ambiguïté par un identifiant construit appelé *NodeId* (*Identifiant de Nœud).* Certains serveurs peuvent accepter d'autres *Identifiants de Nœuds* outre l'*Identifiant de Nœud* canonique représenté dans le présent *Attribut*. La structure de l'*Identifiant de Nœud* est définie en 8.2.

5.2.3 Classe de Nœud (NodeClass)

L'*Attribut NodeClass (Classe de Nœud)* identifie la *Classe de Nœud* d'un *Nœud*. Le type de données correspondant est défini en 8.30.

5.2.4 Nom d'Exploration (BrowseName)

Les *Nœuds* disposent d'un *Attribut BrowseName (Nom d'Exploration*) utilisé comme dénomination non localisée et interprétable par l'utilisateur lorsqu'il explore l'*Espace d'Adresse* pour créer des chemins à partir des *Noms d'Exploration*. Le *Service* "Traduire Chemins d'Exploration en Identifiants de Nœuds" défini dans la CEI 62541-4 peut être utilisé pour suivre un chemin constitué de *Noms d'Exploration*.

Il est recommandé de ne jamais utiliser un *Nom d'Exploration* pour afficher le nom d'un *Nœud*. Il convient d'utiliser pour cela le *Nom d'Affichage*.

Contrairement aux *Identifiants de Nœuds*, le *Nom d'Exploration* ne peut pas être utilisé pour identifier de manière univoque un *Nœud*. Différents *Nœuds* peuvent utiliser le même *Nom d'Exploration*.

Le paragraphe 8.3 définit la structure du *Nom d'Exploration*. Il contient un Espace de Nom et une chaîne de caractères. L'Espace de Nom permet d'assurer l'unicité du *Nom d'Exploration* 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 *Noms d'Exploration* pour les *Propriétés*, l'Espace de Nom du *Nom d'Exploration* fourni par l'organisme rend le *Nom d'Exploration* unique, même si différents organismes peuvent utiliser la même chaîne de caractères avec une signification légèrement différente.

Il arrive souvent que les serveurs choisissent d'utiliser le même Espace de Nom pour l'*Identifiant de Nœud* et le *Nom d'Exploration*. Cependant, s'ils souhaitent fournir une *Propriété* normalisée, son *Nom d'Exploration* doit avoir l'Espace de Nom de l'organisme de normalisation, même si l'Espace de Nom de l'*Identifiant de Nœud* reflète quelque chose de différent, par exemple le serveur local.

Il est recommandé que les organismes de normalisation établissant des définitions de type normalisées utilisent leur Espace de Nom pour l'*Identifiant de Nœud* d'un *Nœud de Définition de Type* ainsi que le *Nom d'Exploration* du *Nœud de Définition de Type.*

5.2.5 Nom d'Affichage (DisplayName)

L'*Attribut DisplayName* (*Nom d'Affichage)* contient la dénomination localisée du *Nœud*. Il faut les clients utilisent cet *Attribut* s'ils souhaitent afficher le nom du *Nœud* pour l'utilisateur. Il ne faut pas qu'ils utilisent le *Nom d'Exploration* à cet effet. Le serveur peut maintenir une ou plusieurs représentations localisées pour chaque *Nom d'Affichage*. Les clients négocient les paramètres de localisation géographique à renvoyer lorsqu'ils ouvrent une session sur le serveur. Pour une description de l'établissement de session et des paramètres de localisation géographique, voir la CEI 62541-4. Le paragraphe 8.5 définit la structure du *Nom d'Affichage*. La partie chaîne de caractères du *Nom d'Affichage* est limitée à 512 caractères.

5.2.6 Description

L'*Attribut Description qui est* facultatif doit expliquer la signification du *Nœud* dans un texte localisé en utilisant les mêmes mécanismes de localisation que ceux décrits pour le *Nom d'Affichage* en 5.2.5.

5.2.7 WriteMask (Masque d'Ecriture)

L'*Attribut WriteMask (Masque d'Ecriture)* qui est facultatif présente au client des possibilités d'écriture des *Attributs* du *Nœud*. L'*Attribut "Masque d'Ecriture"* ne tient compte d'aucun droit d'accès utilisateur, ce qui signifie que même si un *Attribut* est modifiable, cette possibilité peut être limitée à un certain utilisateur / groupe d'utilisateurs.

Si le serveur OPC UA n'a pas la possibilité d'obtenir les informations *Masque d'Ecriture* d'un *Attribut* spécifique à partir du système sous-jacent, il faut déclarer qu'il est modifiable. Si une opération d'écriture est invoquée pour l'*Attribut*, il faut que le serveur transfère cette requête et renvoie le *Code d'Etat* correspondant si une cette requête est rejetée. Les *Codes d'Etat* sont définis dans la CEI 62541-4.

L'*Attribut* "*Masque d'Ecriture"* est un entier de 32 bits non signé dont la structure est définie dans le Tableau 3. Si le bit est réglé à 0, ceci veut dire que l'*Attribut* n'est pas modifiable, s'il est réglé à 1, cela signifie qu'il est modifiable. Si un *Nœud* ne prend pas en charge un *Attribut* spécifique, le bit correspondant doit être mis à 0.

Tableau 3 – Masque de bit pour Masque d'Ecriture et Masque d'Ecriture Utilisateur

Champ	Bit	Description
AccessLevel	0	Indique si l'Attribut "Niveau d'Accès" est modifiable.
ArrayDimensions	1	Indique si l'Attribut "Dimensions de Matrice" est modifiable.
BrowseName	2	Indique si l'Attribut "Nom d'Exploration" est modifiable.
ContainsNoLoops	3	Indique si l'Attribut "Ne Contient Aucune Boucle" est modifiable.
DataType	4	Indique si l'Attribut "Type de Données" est modifiable.
Description	5	Indique si l'Attribut "Description" est modifiable.
DisplayName	6	Indique si l'Attribut "Nom d'Affichage" est modifiable.
EventNotifier	7	Indique si l'Attribut "Notificateur d'Evénements" est modifiable.
Executable	8	Indique si l'Attribut "Exécutable" est modifiable.
Historizing	9	Indique si l'Attribut "Historisation" est modifiable.
InverseName	10	Indique si l'Attribut "Nom Inverse" est modifiable.
IsAbstract	11	Indique si l'Attribut "Est Abstrait" est modifiable.
MinimumSamplingInterval	12	Indique si l'Attribut "Intervalle d'Echantillonnage Minimal" est modifiable.
NodeClass	13	Indique si l'Attribut "Classe de Nœud" est modifiable.
NodeId	14	Indique si l'Attribut "Identifiant de Nœud" est modifiable.
Symmetric	15	Indique si l'Attribut "Symétrique" est modifiable.
UserAccessLevel	16	Indique si l'Attribut "Niveau d'Accès Utilisateur" est modifiable.
UserExecutable	17	Indique si l'Attribut "Exécutable par l'Utilisateur" est modifiable.
UserWriteMask	18	Indique si l'Attribut "Masque d'Ecriture Utilisateur" est modifiable.
ValueRank	19	Indique si l'Attribut "Rang de Valeur" est modifiable.
WriteMask	20	Indique si l'Attribut "Masque d'Ecriture" est modifiable.
ValueForVariableType	21	Indique si l'*Attribut "Valeur"* est modifiable pour un *Type de Variable* donné. Il ne s'applique pas aux *Variables* puisqu'il est traité par les *Attributs "Niveau d'Accès"* et *"Niveau d'Accès Utilisateur"* pour la *Variable*. Dans ce cas, pour les *Variables*, ce bit doit être mis à 0.
Reserved	22:32	Réservé pour usage ultérieur. Doit toujours être à zéro.

5.2.8 UserWriteMask (Masque d'Ecriture Utilisateur)

L'*Attribut UserWriteMask (Masque d'Ecriture Utilisateur)* qui est facultatif 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 même masque de bit que celui de l'*Attribut "Masque d'Ecriture"*, défini dans le Tableau 3.

L'*Attribut "Masque d'Ecriture Utilisateur"* étend la restriction de l'*Attribut "Masque d'Ecriture",* lorsqu'il est mis sur non modifiable dans le cas général applicable à chaque utilisateur.

5.3 Classe de Nœud "Types de Références"

5.3.1 Généralités

Les *Références* sont définies comme des instances des *Nœuds de Type de Référence*. Les *Nœuds de Type de Référence* sont visibles dans l'*Espace d'Adresse* et sont définis en utilisant la *Classe de Nœud "Types de Références"* comme spécifié dans le Tableau 4. Par contre, une *Référence* est une partie inhérente d'un *Nœud* et aucune *Classe de Nœud* n'est utilisée pour représenter des *Références*.

La présente norme définit un ensemble de *Types de Références* fournis comme parties inhérentes du Modèle de l'Espace d'Adresse d'OPC UA. Ces *Types de Références* sont définis dans l'Article 7 et leur représentation dans *l'Espace d'Adressage* est définie dans la CEI 62541-5. Les serveurs peuvent également définir des *Types de Références*. Par ailleurs, la CEI 62541-4 définit des *Services de Gestion de Nœuds* qui permettent aux clients d'ajouter des *Types de Références* à l'*Espace d'Adresse*.

Tableau 4 – Classe de Nœud "Types de Références"

Dénomination	Usage	Type de Données	Description
Attributs			
Base NodeClass Attributes	M	--	Hérité de la *Classe de Nœud de Base*. Voir 5.2
IsAbstract	M	Booléen	Un *Attribut* booléen prenant les valeurs suivantes: VRAI Il s'agit d'un *Type de Référence* abstrait, c'est-à-dire qu'aucune *Référence* de ce type ne doit exister, mais uniquement ses sous-types. FAUX Il s'agit d'un *Type de Référence qui n'est pas abstrait*, c'est-à-dire qu'il peut y avoir des *Références* de ce type.
Symmetric	M	Booléen	Un *Attribut* booléen prenant les valeurs suivantes: VRAI La signification du *Type de Référence* est la même que celle qui est perçue à la fois du point de vue du *SourceNode* et du *Nœud Cible.* FAUX La signification du *Type de Référence* telle que perçue du point de vue du *TargetNode* est l'inverse de celle qui est perçue à partir du *Nœud Source*.
InverseName	O	Texte Localisé	La dénomination inverse de la *Référence*, c'est-à-dire la signification du *Type de Référence* telle que perçue à partir du *Nœud Cible*.
Références			
HasProperty	0..*		Utilisé pour identifier les Propriétés (voir 5.3.3.2)
HasSubtype	0..*		Utilisé pour identifier des sous-types (voir 5.3.3.3)
Propriétés normalisées			
NodeVersion	O	Chaîne	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 ou retirée au *Nœud* auquel la *Propriété* appartient. Les modifications de la valeur de l'*Attribut* n'entraînent pas une modification de la *Version de Nœud*. Les clients peuvent consulter la *Propriété «NodeVersion»* ou s'y abonner pour établir le moment où la structure d'un *Nœud* a changé.

5.3.2 Attributs

La *Classe de Nœud "Types de Références"* hérite des *Attributs* de base de la *Classe de Nœud de Base* définie en 5.2. L'*Attribut "Nom d'Exploration"* hérité est utilisé pour préciser la signification du *Type de Référence* telle que perçue à partir du *Nœud Source*. Par exemple, le *Type de Référence* portant le *Nom d'Exploration* "Contains" (Contient) est utilisé dans les *Références* pour indiquer que le *SourceNode* contient le *Nœud Cible*. L'*Attribut "Nom d'Affichage"* hérité contient une traduction du *Nom d'Exploration.*

Le *Nom d'Exploration* d'un *Type de Référence* donné doit être unique dans un serveur. Il n'est pas admis que deux *Types de Références* différents portent le même *Nom d'Exploration*.

L'*Attribut IsAbstract (Est Abstrait)* indique que ce *Type de Référence* est abstrait. Les *Types de Références* 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 héritées par ces sous-types.

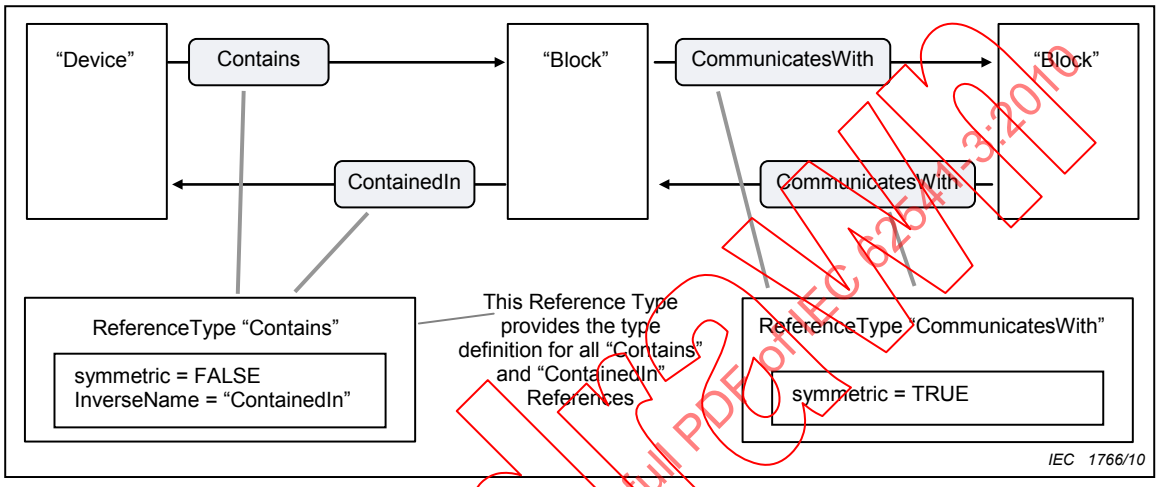
L'*Attribut Symmetric (Symétrique)* est utilisé pour indiquer si la signification du *Type de Référence* est (ou n'est pas) identique à la fois pour le *SourceNode* et le *Nœud Cible*.

Si un *Type de Référence* est symétrique, l'*Attribut InverseName (Nom Inverse)* doit être omis. Les exemples de *Types de Références* symétriques sont "Se Connecte A" et "Communique Avec". Les deux impliquent la même sémantique provenant du *SourceNode* ou du *Nœud Cible*. Par conséquent, les deux directions sont considérées comme des Références directes.

Si le *Type de Référence* est non symétrique et non abstrait, l'*Attribut "Nom Inverse"* doit être établi. L'*Attribut "Nom Inverse"* spécifie la signification du *Type de Référence* telle que perçue à partir du *Nœud Cible*. Les exemples de *Types de Références* non symétriques sont notamment “Contient” et “Contenu Dans”, et “Reçoit De” et “Envoie A”.

Les *Références* qui utilisent le *Nom Inverse*, par exemple les *Références* “Contenu Dans”, sont appelées *Références* inverses.

La Figure 8 donne des exemples de *Références* symétriques et non symétriques ainsi que d'utilisation du *Nom d'Exploration* et du *Nom Inverse*.



Légende

Anglais	Français
Device	Dispositif
Contains	Contient
Block	Bloc
Communicateswith	Communique avec
Containedin	Contenu dans
Referencetype “contains”	Type de référence “contient”
This referencetype provides the type definition for all “contains” and “containedin’ references	Ce Type de Référence fournit la définition de type de toutes les références “contient” et “contenu dans”
Referencetype “communicateswith”	Type de Référence “communique avec”
Symmetric = false Inversename = “containedin”	Symétrique = Faux Nom inverse = “contenu dans”
Symmetric = True	Symétrique = Vrai

Figure 8 – 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 *Types de Références* non symétriques, comme illustré en Figure 8. 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 l'exploration. Une *Référence* bidirectionnelle est modélisée comme deux *Références* séparées.

Un exemple de *Référence unidirectionnelle:* il arrive souvent qu'un abonné connaisse son éditeur, mais que ce dernier ne connaisse pas ses abonnés. L'abonné aurait une *Référence*

“S'Abonne A” vers l'éditeur, sans que l'éditeur n'ait les *Références* inverses correspondantes "Edite Vers" pointant vers ses abonnés.

Le *Nom d'Affichage* et le *Nom Inverse* sont les seuls emplacements normalisés utilisés pour indiquer la sémantique d'un *Type de Référence* donné. Il peut y avoir plusieurs règles sémantiques complexes associées à un *Type de Référence* qui peuvent être exprimées dans ces *Attributs* (par exemple la sémantique de la *Référence «HasSubtype»*). La présente norme ne spécifie pas la manière d'exprimer cette sémantique. Il est cependant possible d'utiliser à cet effet l'*Attribut "Description"*. Cette norme fournit une règle sémantique pour les *Types de Références* spécifiés à l'Article 7.

Un *Type de Référence* peut avoir des contraintes limitant son utilisation. Il peut spécifier par exemple qu'en partant du *Nœud* A et uniquement en suivant les *Références* de ce *Type de Référence* ou d'un de ses sous-types, il ne doit jamais être possible de revenir à A; ce qui signifie une contrainte “Pas de Boucle”.

La présente norme ne spécifie pas la manière dont il est possible de mettre à disposition ces contraintes dans *l'Espace d'Adressage*. Cependant, pour les *Types de Références* normalisés, certaines contraintes sont spécifiées à l'Article 7. La présente norme ne limite pas le type de contraintes utilisables pour un *Type de Référence* donné. La contrainte peut, par exemple, affecter également un *Type d'Objet*. La restriction selon laquelle un *Type de Référence* ne peut être utilisé que par des *Nœuds* de certaines *Classes de Nœuds* ayant une cardinalité définie est une contrainte particulière d'un *Type de Référence*.

5.3.3 Références

5.3.3.1 Généralités

Les *Références HasSubtype (Dispose d'un Sous-type)* et *HasProperty (Dispose d'une Propriété)* sont les seuls *Types de Références* qui peuvent être utilisés avec des *Nœuds de Type de Référence* tels que le *Nœud Source*. Les *Nœuds de Type de Référence* ne doivent pas être le *SourceNode* d'autres types de *Références*.

5.3.3.2 Références "Dispose d'une Propriété"

Les *Références HasProperty (Dispose d'une Propriété)* sont utilisées pour identifier les *Propriétés* d'un *Type de Référence* et ne doivent se référer qu'aux *Nœuds* de la *Classe de Nœud "Variables"*.

La *Propriété NodeVersion (Version de Nœud)* est utilisée pour indiquer la version du *Type de Référence*.

Il n'y a pas d'autres *Propriétés* définies pour les *Types de Références* dans cette norme. D'autres parties de la présente série de normes peuvent définir des *Propriétés* supplémentaires pour les *Types de Références*.

5.3.3.3 Références " *HasSubtype* " (Dispose d'un Sous-type)

Les *Références HasSubtype (Dispose d'un Sous-type)* sont utilisées pour définir des sous-types de *Types de Références*. Il n'est pas exigé que la *Référence «HasSubtype»* soit fournie pour le supertype; il est cependant exigé que ce sous-type fournisse la *Référence* inverse à son supertype. Les règles suivantes s'appliquent au sous-typage.

a) La sémantique d'un *Type de Référence* (par exemple “couvre une hiérarchie”) est héritée par ses sous-types et peut être définie plus finement (par exemple “couvre une hiérarchie spéciale”). Le *Nom d'Affichage*, ainsi que le *Nom Inverse* pour les *Types de Références* non symétriques, reflètent cette spécialisation.

b) Si un *Type de Référence* spécifie certaines contraintes (par exemple “ne permet aucune boucle”), cette restriction est héritée et ne peut qu'être rendue plus sévère (par exemple

hérité “aucune boucle” pourrait être affiné en “doit être une arborescence – un parent uniquement”) mais non réduit (par exemple “autoriser les boucles”).

c) Les contraintes concernant les *Classes de Nœuds* qui peuvent être référencées sont également héritées et ne peuvent être que plus strictes. En d'autres termes, s'il n'est pas admis qu'un *Type de Référence* “A” relie un *Objet* à un *Type d'Objet*, ceci est également vrai pour tous ses sous-types.

d) Un *Type de Référence* doit avoir exactement un supertype, sauf en ce qui concerne les *Type de Référence "Références"* définies en 7.2 comme étant le type racine de la hiérarchie du *Type de Référence*. La hiérarchie du *Type de Référence* ne prend pas en charge les héritages multiples.

5.4 Classe de Nœud "Vues"

Les systèmes sous-jacents sont souvent de grande taille et les clients n'ont souvent besoin que d'un sous-ensemble spécifique de données. Ils ne souhaitent pas ou ne veulent pas être surchargés par des *Nœuds* de Vues dans *l'Espace d'Adressage* qui ne les intéresse aucunement.

Pour résoudre ce problème, cette spécification définit le concept de *Vue (View)*. Chaque *Vue* définit un sous-ensemble de *Nœuds* dans *l'Espace d'Adressage*. L'ensemble *Espace d'Adressage* 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 de Vues* agit comme la racine des *Nœuds* dans la *Vue*. Les *Vues* sont définies au moyen de la *Classe de Nœud "Vues"* spécifiée dans le Tableau 5.

Tous les *Nœuds* contenus dans une *Vue* doivent être accessibles à partir du *Nœud de Vues* lorsqu'une exploration est effectuée dans le contexte de la *Vue*. L'exploration peut effectuer plusieurs sauts, ce qui signifie qu'il n'est pas nécessaire que tous les *Nœuds* contenants puissent être explorés directement à partir du *Nœud de Vues*.

Un *Nœud de Vues* ne peut pas être uniquement utilisé comme entrée supplémentaire dans l'*Espace d'Adressage* mais comme un concept permettant d'organiser l'*Espace d'Adressage* et ainsi comme le seul point d'entrée dans un sous-ensemble de l'*Espace d'Adressage*. Par conséquent, les clients ne doivent pas ignorer les *Nœuds de Vues* lorsqu'ils présentent l'*Espace d'Adressage*. Les clients modestes qui n'utilisent pas les *Vues* à des fins de filtrage peuvent par exemple traiter un *Nœud de Vues* comme un type d'*Objet* de type «*Type de Dossier*» (voir 5.5.3).

Tableau 5 – Classe de Nœud "Vues"

<table>
<tr><th>Dénomination</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>Base NodeClass Attributes</td><td>M</td><td>--</td><td>Hérité de la Classe de Nœud de Base. Voir 5.2</td></tr>
<tr><td>ContainsNoLoops</td><td>M</td><td>Booléen</td><td>S'il est mis sur “vrai”, cet Attribut indique que les Références suivantes dans le contexte de la Vue ne contient 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 de Vues, “A” ne sera jamais atteint de nouveau. Ceci ne signifie pas qu'il y a qu'un chemin à partir du Nœud de Vues pour atteindre un Nœud contenu dans la Vue.
S'il est réglé sur “faux”, 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>Octet</td><td>L'Attribut "Notificateur d'Evénements" est utilisé pour indiquer si le Nœud peut être utilisé pour s'abonner à des Evénements ou pour consulter / modifier des Evénements historiques.
Le Notificateur d'Evénements 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>S'Abonner à des Evénements</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>Réservé</td><td>1</td><td>Réservé pour usage ultérieur. Doit toujours être à zéro.</td></tr>
<tr><td>Consultation de l'Historique</td><td>2</td><td>Indique si l'historique des Evénements est consultable
(0 signifie non consultable, 1 signifie consultable).</td></tr>
<tr><td>Modification de l'Historique</td><td>3</td><td>Indique si l'historique des Evénements est modifiable
(0 signifie non modifiable, 1 signifie modifiable).</td></tr>
<tr><td>Réservé</td><td>4:7</td><td>Réservé pour usage ultérieur. Doit toujours être à 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 "Dispose d'une Propriété" identifient les Propriétés de la Vue.</td></tr>
<tr><td>Propriétés normalisées</td><td></td><td></td><td></td></tr>
<tr><td>NodeVersion</td><td>O</td><td>Chaîne</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 ou retirée du Nœud auquel la Propriété appartient. Les modifications de la valeur de l'Attribut n'entraînent pas une modification de la Version de Nœud. Les clients peuvent consulter la Propriété «NodeVersion» ou s'y abonner pour établir le moment où la structure d'un Nœud a changé.</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 "Version de Vue" 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 Version de la Vue doit toujours être supérieure à 0.</td></tr>
</table>

La *Classe de Nœud "Vues"* hérite des *Attributs* de base de la *Classe de Nœud de Base* définie en 5.2. Deux *Attributs* supplémentaires sont également spécifiés.

L'*Attribut* obligatoire *ContainsNoLoops* (*Ne Contient Aucune Boucle)* est réglé sur faux si le serveur est incapable de savoir si la *Vue* contient ou non des boucles.

L'*Attribut* obligatoire *EventNotifier (Notificateur d'Evénements)* 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 qu'*Evénements de Modification du Modèle* du contenu de la *Vue*; il est également utilisé pour consulter/modifier l'historique des *Evénements*. Une *Vue* prenant en charge des *Evénements* doit prévoir tous les *Evénements* qui ont lieu dans tout *Objet* utilisé comme *Notificateur d'Evénements* faisant partie du contenu de la *Vue*. En outre, il doit tenir compte de tous les *Evénements de Modification du Modèle* 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 la CEI 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* d'exploration et d'interrogation définis dans la CEI 62541-4 attendent de recevoir l'*Identifiant de Nœud* d'un *Nœud de Vues* pour fournir ces *Services* dans le contexte de la *Vue*.

Les *Références "Dispose d'une Propriété"* sont utilisées pour identifier les *Propriétés* d'une *Vue*. La Propriété *«NodeVersion»* est utilisée pour indiquer la version du *Nœud de Vues*. La *Propriété "Version de Vue"* est utilisée pour indiquer la version du contenu de la *Vue*. Contrairement à la *Version de Nœud,* la *Propriété "Version de Vue"* est mise à jour même s'il est ajouté ou retiré de la *Vue* des *Nœuds* qui ne sont pas directement référencés par le *Nœud de Vues*. Cette *Propriété* est facultative car il est possible que certains serveurs ne puissent pas détecter des modifications dans le contenu de la *Vue*. Les serveurs peuvent également générer un *Evénement de Modification du Modèle* (décrit en 9.30) si des *Nœuds* sont ajoutés ou supprimés de la *Vue*. Il n'y a pas d'autres *Propriétés* définies pour les *Vues* dans le présent document. D'autres parties de la présente série de normes peuvent définir des *Propriétés* supplémentaires pour les *Vues*.

Les *Vues* peuvent être le *SourceNode* de toute *Référence hiérarchique*. Elles ne doivent pas être le *SourceNode* d'une quelconque *Référence non hiérarchique*.

5.5 Objets

5.5.1 Classe de Nœud "Objets"

Les *Objets* sont utilisés pour représenter des systèmes, des composants de systèmes, des objets concrets et des objets logiciels. Les *Objets* sont définis en utilisant la *Classe de Nœud "Objets"* spécifiée dans le Tableau 6.

Tableau 6 – Classe de Nœud "Objets"

<table>
<tr><th>Dénomination</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>Base NodeClass Attributes</td><td>M</td><td>--</td><td>Hérité de la Classe de Nœud de Base. Voir 5.2</td></tr>
<tr><td>EventNotifier</td><td>M</td><td>Octet</td><td>L'Attribut "Notificateur d'Evénements" est utilisé pour indiquer si le Nœud peut être utilisé pour s'abonner à des Evénements ou pour consulter / modifier des Evénements historiques.
Le Notificateur d'Evénements 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>S'Abonner à des Evénements</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>Réservé</td><td>1</td><td>Réservé pour usage ultérieur. Doit toujours être à zéro.</td></tr>
<tr><td>Consulta-tion de l'Historique</td><td>2</td><td>Indique si l'historique des Evénements est consultable
(0 signifie non consultable, 1 signifie consultable).</td></tr>
<tr><td>Modifica-tion de l'Historique</td><td>3</td><td>Indique si l'historique des Evénements est modifiable
(0 signifie non modifiable, 1 signifie modifiable).</td></tr>
<tr><td>Réservé</td><td>4:7</td><td>Réservé pour usage ultérieur. Doit toujours être à 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>HasComponent</td><td>0..*</td><td></td><td>Les Références "Dispose d'un composant" identifient les Variables de Données, les Méthodes et les Objets contenus dans l'Objet.</td></tr>
<tr><td>HasProperty</td><td>0..*</td><td></td><td>Les Références "Dispose d'une Propriété" identifient les Propriétés de l'Objet.</td></tr>
<tr><td>HasModellingRule</td><td>0..1</td><td></td><td>Les Objets peuvent pointer vers, au plus, un Objet Règle de Modélisation en utilisant une Référence "Dispose d'une Règle de Modélisation" (pour plus de détails concernant les Règles de Modélisation voir 6.4.4).</td></tr>
<tr><td>HasTypeDefinition</td><td>1</td><td></td><td>La Référence "Dispose d'une Définition de Type" 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 "Dispose d'une Définition de Type" pointant vers un Type d'Objet. Voir 4.5 pour une description des définitions de types.</td></tr>
<tr><td>HasModelParent</td><td>0..1</td><td></td><td>La Référence Dispose d'un Modèle Parent pointe vers le Modèle Parent de l'Objet (voir 6.6 pour plus de détails concernant les Modèles Parents).</td></tr>
<tr><td>HasEventSource</td><td>0..*</td><td></td><td>Les Références "Dispose d'une Source d'Evénement" pointent vers les sources d'événements de l'Objet. Les Références de ce type peuvent uniquement être utilisées pour des Objets dont le bit “S'Abonner à des Evénements” est à Un dans l'Attribut "Notificateur d'Evénements". Une description plus détaillée est fournie en 7.18.</td></tr>
<tr><td>HasNotifier</td><td>0..*</td><td></td><td>Les Références "Dispose d'un Notificateur" pointent vers les Notificateurs de l'Objet. Les Références de ce type peuvent uniquement être utilisées pour des Objets dont le bit “S'Abonner à des Evénements” est à Un dans l'Attribut "Notificateur d'Evénements". Une description plus détaillée est fournie en 7.20.</td></tr>
<tr><td>Organizes</td><td>0..*</td><td></td><td>Il est recommandé que cette Référence soit uniquement utilisée pour des Objets du Type d'Objet "Type de Dossier" (voir 5.5.3).</td></tr>
<tr><td>HasDescription</td><td>0..1</td><td></td><td>Cette Référence doit uniquement être utilisée pour des Objets du Type d'Objet "Type de Codage de Type de Données" (voir 5.8.4).</td></tr>
<tr><td><other References></td><td>0..*</td><td></td><td>Les Objets peuvent contenir d'autres Références.</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>Chaîne</td><td>La Propriété «NodeVersion» est utilisée pour indiquer la version d'un Nœud.</td></tr>
</table>

Dénomination	Usage	Type de Données	Description
			La Propriété «NodeVersion» est mise à jour à chaque fois qu'une *Référence* est ajoutée ou retirée du *Nœud* auquel la *Propriété* appartient. Les modifications de la valeur de l'*Attribut* n'entraînent pas une modification de la *Version de Nœud*. Les clients peuvent consulter la *Propriété «NodeVersion»* ou s'y abonner pour établir le moment où la structure d'un *Nœud* a changé.
Icon	O	Image	La Propriété *"Icône"* fournit une image qui peut être utilisée par les clients lorsqu'ils affichent le *Nœud*. Il est prévu que la *Propriété "Icône"* contienne une image relativement petite.
NamingRule	O	Type de Règle de Nommage	La Propriété "*Règle de Nommage"* définit la *Règle de Nommage* d'une *Règle de Modélisation* (voir 6.4.4.2.1 pour plus de détails). Cette *Propriété* doit uniquement être utilisée pour des *Objets* de type "*Type* de *Règle de Modélisation"* défini en 6.4.4.

La *Classe de Nœud "Objets"* hérite des *Attributs* de base de la *Classe de Nœud de Base* définie en 5.2.

L'*Attribut* obligatoire *EventNotifier* (*Notificateur d'Evénements)* est utilisé pour indiquer si l'*Objet* peut être utilisé pour s'abonner à des *Evénements* ou pour consulter et modifier l'historique des *Evénements*.

La *Classe de Nœud "Objets"* utilise la *Référence HasComponent* (*Dispose d'un composant)* pour définir les *Variables de Données*, les *Objets* et les *Méthodes* d'un *Objet* donné.

Elle utilise la *Référence* HasProperty (*Dispose d'une Propriété)* pour définir les *Propriétés* d'un *Objet*. La *Propriété «NodeVersion»* est utilisée pour indiquer la version de l'*Objet*. La *Propriété Icon (Icône*) fournit une icône de l'*Objet*. La *Propriété NamingRule* (*Règle de Nommage)* définit la *Règle de Nommage* d'une *Règle de Modélisation* et doit uniquement être appliquée à des *Objets* de type "*Type de Règle de Modélisation"*. Il n'y a pas d'autres *Propriétés* définies pour les *Objets* dans le présent document. D'autres parties de la présente série de normes peuvent définir des *Propriétés* supplémentaires pour les *Objets*.

Pour spécifier sa *Règle de Modélisation*, un *Objet* peut utiliser au plus une *Référence "Dispose d'une Règle de Modélisation"* pointant vers un *Objet "Règle de Modélisation"*. Les *Règles de Modélisation* sont définies en 6.4.4.

Un *Objet* doit utiliser au plus une *Référence "Dispose d'un Modèle Parent"* pour spécifier son *Modèle Parent* (voir 6.6 pour plus de détails).

Les *Références HasNotifier* (*Dispose d'un Notificateur)* et *HasEventSource* (*Dispose d'une Source d'Evénement)* sont utilisées pour fournir des informations concernant les événements et ne peuvent être appliquées qu'aux *Objets* utilisés comme notificateurs d'événements. Une description plus détaillée est fournie en 7.18 et 7.20.

La *Référence HasTypeDefinition (Dispose d'une Définition de Type)* pointe vers le *Type d'Objet* 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 *Classes de Nœuds* correspondant aux *Nœuds* pouvant être référencés. Cependant, le *Type de Référence* peut définir des restrictions qui excluent son utilisation pour des *Objets*. Des *Types de Références* normalisés sont décrits à l'Article 7.

Si l'*Objet* est utilisé comme *Déclaration d'Instance* (voir 4.5) tous les *Nœuds* référencés avec des *Références hiérarchiques* dans le sens direct doivent avoir des *Noms d'Exploration* uniques dans le contexte de cet *Objet*.

Si l'*Objet* est créé sur la base d'une *Déclaration d'Instance*, il doit avoir le même *Nom d'Exploration* que sa *Déclaration d'Instance*.

5.5.2 Classe de Nœud "ObjectType"

Les *Types d'Objets* fournissent des définitions pour les *Objets*. Les *Types d'Objets* sont définis en utilisant la *Classe de Nœud* "*ObjectType*", qui est spécifiée dans le Tableau 7.

Tableau 7 – Classe de Nœud "Types d'Objets"

Dénomination	Usage	Type de données	Description
Attributs			
Base NodeClass Attributes	M	--	Hérité de la *Classe de Nœud de Base*. Voir 5.2
IsAbstract	M	Booléen	Un *Attribut* booléen prenant les valeurs suivantes: VRAI Il s'agit d'un *Type d'Objet* abstrait, ce qui signifie qu'aucun *Objet* de ce type ne doit exister, mais uniquement ses sous-types. FAUX Il s'agit d'un *Type d'Objet* qui n'est pas abstrait, c'est-à-dire qu'il peut y avoir des *Objets* de ce type.
Références			
HasComponent	0..*		Les *Références "Dispose d'un composant"* identifient les *Variables de Données*, les *Méthodes* et les *Objets* contenus dans le *Type d'Objet*. Le paragraphe 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 "Dispose d'une Propriété"* identifient les *Propriétés* du *Type d'Objet*. Le paragraphe 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 *Types d'Objets* qui sont des sous-types de ce type. La *Référence* inverse "*Sous-type de"* identifie le type parent de ce type.
GeneratesEvent	0..*		Les *Références* "*Génère un Evénement*" identifient le type d'instances d'*Evénements* que ce type peut générer.
<other References>	0..*		Les *Types d'Objets* peuvent contenir d'autres *Références* qui peuvent être instanciées par des *Objets* définis par ce *Type d'Objet*.
Propriétés normalisées			
NodeVersion	O	Chaîne	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 ou retirée du *Nœud* auquel la *Propriété* appartient. Les modifications de la valeur de l'*Attribut* n'entraînent pas une modification de la *Version de Nœud*. Les clients peuvent consulter la *Propriété «NodeVersion»* ou s'y abonner pour établir le moment où la structure d'un *Nœud* a changé.
Icon	O	Image	La Propriété *"Icône"* fournit une image qui peut être utilisée par les clients lorsqu'ils affichent le *Nœud*. Il est prévu que la *Propriété "Icône"* contienne une image relativement petite.

La *Classe de Nœud* "*ObjectType*" hérite des *Attributs* de base de la *Classe de Nœud de Base* définie en 5.2. L'*Attribut* supplémentaire *IsAbstract (Est Abstrait)* indique si le *Type d'Objet* est ou non abstrait.

La *Classe de Nœud "ObjectType"* utilise les *Références "Dispose d'un composant"* pour définir les *Variables de Données*, les *Objets* et les *Méthodes* d'un *Type d'Objet* donné.

La Référence *HasProperty (Dispose d'une Propriété)* est utilisée pour identifier les *Propriétés*. La *Propriété NodeVersion (Version de Nœud)* est utilisée pour indiquer la version du *Type d'Objet*. La *Propriété Icon* (*Icône)* fournit une icône du *Type d'Objet*. Il n'y a pas d'autres *Propriétés* définies pour les *Types d'Objets* dans le présent document. D'autres parties de la présente série de normes peuvent définir des *Propriétés* supplémentaires pour les *Types d'Objets*.

Les *Références HasSubtype (Dispose d'un Sous-type)* sont utilisées pour définir des sous-types de *Types d'Objets*. Les sous-types de *Type d'Objet* 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 que la *Référence «HasSubtype»* soit fournie pour le supertype; il est cependant exigé que ce sous-type fournisse la *Référence* inverse à son supertype.

Les Références GeneratesEvent (Génère un Evénement) identifient le type d'*Evénements* que les instances de ce *Type d'Objet* peuvent générer. Ces *Objets* peuvent être la source d'un *Evénement* de type spécifié ou d'un de ses sous-types. Il convient que les serveurs fassent en sorte que les *Références "Génère un Evénement"* soient bidirectionnelles. Il est cependant admis qu'elles soient unidirectionnelles lorsque le serveur n'est pas capable de présenter le sens inverse pointant du *Type d'Evénement* vers chaque *Type d'Objet* prenant en charge le *Type d'Evénement.* Il faut noter que l'*Attribut "Notificateur d'Evénements"* d'un *Objet* et les Références *"Génère un Evénement"* de ce *Type d'Objet* n'ont aucune relation. Les *Objets* qui peuvent générer des *Evénements* pourraient ne pas être utilisés comme des *Objets* auxquels s'abonnent les clients pour obtenir les notifications d'*Evénements* correspondantes.

Les Références "Génère un Evénement" sont facultatives, c'est-à-dire que les *Objets* peuvent générer des *Evénements* d'un *Type d'Evénement* qui n'est pas présenté par ce *Type d'Objet*.

Les *Types d'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 *Classes de Nœuds* correspondant aux *Nœuds* pouvant être référencés. Cependant, le *Type de Référence* peut définir des restrictions qui excluent son utilisation pour des *Types d'Objets*. Des *Types de Références* normalisés sont décrits à l'Article 7.

Tous les *Nœuds* référencés avec des *Références hiérarchiques* doivent avoir des *Noms d'Exploration* uniques dans le contexte d'un *Type d'Objet* (voir 4.5).

5.5.3 Type d'Objet normalisé "FolderType" (Type de Dossier)

Le *Type d'Objet FolderType* (*Type de Dossier)* est formellement défini dans la CEI 62541-5. Son objectif est de fournir des *Objets* qui n'ont aucune valeur sémantique autre que l'organisation de l'*Espace d'Adressage*. Il est introduit un *Type de Référence* particulier pour ces *Objets "Dossiers"*, le *Type de Référence* "*Organise"*. Il convient toujours que le *SourceNode* d'une telle Référence soit une *Vue* ou un *Objet* du *Type d'Objet "Type de Dossier"*; le *TargetNode* peut être de n'importe quelle *Classe de Nœud.* Les *Références Organizes* (*Organise*) peuvent être utilisées en toute combinaison avec des *Références "Dispose d'une Référence Enfant"* ("*HasComponent*", *Dispose d'une Propriété*, etc.; voir 7.5) et n'empêchent pas les boucles. Elles peuvent ainsi être utilisées pour couvrir de multiples hiérarchies.

5.5.4 Création du côté client d'Objets d'un Type d'Objet donné

Les *Objets* sont toujours fondés sur un *Type d'Objet;* ce qui signifie qu'ils ont une *Référence "Dispose d'une Définition de Type"* pointant vers son *Type d'Objet*.

Les clients peuvent créer des *Objets* en utilisant le *Service* "*Ajouter des Nœuds"* défini dans la CEI 62541-4. Le *Service* exige que la *Définition Type du Nœud* de l'*Objet* soit spécifié. Un *Objet* créé par le *Service* "*Ajouter des Nœuds"* contient tous les composants définis par son *Type d'Objet* en fonction des *Règles de Modélisation* 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 le *Type d'Objet*. Ce comportement dépend du serveur. Le *Type d'Objet* spécifie uniquement l'ensemble minimal de composants qui doit exister pour chaque *Objet* d'un *Type d'Objet* donné.

Outre le *Service* "*Ajouter des Nœuds"*, les *Types d'Objets* peuvent disposer d'une *Méthode* spéciale avec le *Nom d'Exploration* "*Créer*". Cette Méthode est utilisée pour créer un *Objet* de

ce *Type d'Objet*. Cette *Méthode* peut être utile pour créer des *Objets* lorsque qu'il est nécessaire que les règles sémantiques de la création soient différentes du comportement par défaut attendu dans le contexte du *Service "Ajouter des Nœuds"*. Il convient par exemple que les valeurs soient totalement différentes des valeurs par défaut ou il faut que des *Objets* supplémentaires soient ajoutés, etc. Les arguments d'entrée et de sortie de cette *Méthode* dépendent du *Type d'Objet*; la seule caractéristique commune est le *Nom d'Exploration* qui indique que cette *Méthode* créera un *Objet* sur la base du *Type d'Objet*. Il faut que les serveurs ne fournissent pas une *Méthode* portant sur un *Type d'Objet* avec le *Nom d'Exploration* "*Créer*" pour des raisons autres que la création d'*Objets* du *Type d'Objet*.

5.6 Variables

5.6.1 Généralités

Il est défini deux types de *Variables*: les *Propriétés* et les *Variables de données*. Bien qu'elles soient utilisées de manière différente comme cela est décrit en 4.4 et qu'elles aient des contraintes différentes (décrites dans les paragraphes ci-après), elles utilisent la même *Classe de Nœud (*décrite en 5.6.2). Les contraintes de *Propriétés* fondées sur cette *Classe de Nœud* sont définies en 5.6.3, les contraintes de *Variables de Données* sont indiquées en 5.6.4.

5.6.2 Classe de Nœud "Variables"

Les *Variables* sont utilisées pour représenter des valeurs qui peuvent être simples ou complexes. Les *Variables* sont définies par des *Types de Variables*, spécifiées en 5.6.5.

Les *Variables* sont toujours définies comme des *Propriétés* ou des *Variables de Données* d'autres *Nœuds* de *l'Espace d'Adresse*. 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 *Classe de Nœud "Variables "* spécifiée dans le Tableau 8.

Tableau 8 – Classe de Nœud "Variables"

Dénomination	Usage	Type de données	Description
Attributs			
Base NodeClass Attributes	M	--	Hérité de la *Classe de Nœud de Base*. Voir 5.2
Value	M	Défini par l'*attribut "DataType"*	La 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 "Valeur"*.
DataType	M	Identifiant de Nœud	*Identifiant de Nœud* de la définition de *Type de Données* pour l'*Attribut "Valeur"*. Des *Types de Données* normalisés sont décrits à l'Article 8.
ValueRank	M	Int32	Cet *Attribut* indique si l'*Attribut "Valeur"* 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 ayant le nombre spécifié de dimensions. Unidimensionnelle (1): La valeur est une matrice unidimensionnelle. Une Ou Plusieurs Dimensions (0): La valeur est une matrice à une ou plusieurs dimensions. Scalaire (−1): La valeur n'est pas une matrice. Indifférente (−2): La valeur peut être scalaire ou une matrice ayant n'importe quel nombre de dimensions. Scalaire Ou Unidimensionnelle (−3): La valeur peut être scalaire ou une matrice unidimensionnelle.
ArrayDimensions	O	UInt32[]	Cet *Attribut* spécifie la longueur de chaque dimension d'une valeur matricielle. L'Attribut permet de décrire la capacité de la *Variable* et non la taille courante. Le nombre d'éléments doit être égal à la valeur de l'*Attribut* "*Rang de Valeur*". Doit être nul si le *Rang de Valeur* ≤0. Une valeur 0 pour une dimension particulière indique que la dimension a une longueur variable. Par exemple, si une *Variable* est définie par la matrice C suivante: Int32 ma Matrice[346]; ainsi, ce *Type de Données* de *Variable* pointerait vers une Int32, le *Rang de Valeur* de la *Variable* prend la valeur 1 et l'*Attribut "Dimensions de Matrice"* indique une matrice dont l'une des entrées a la valeur 346.

<table>
<tr><th>Dénomination</th><th>Usage</th><th>Type de données</th><th>Description</th></tr>
<tr><td>AccessLevel</td><td>M</td><td>Octet</td><td>L'Attribut "Niveau d'Accès" est utilisé pour indiquer l'accessibilité (lecture/écriture) de la Valeur d'une Variable et si elle contient des données courantes et/ou historiques. Le Niveau d'Accès ne tient compte d'aucun droit d'accès utilisateur, ce qui signifie que même si la Variable est modifiable, cette possibilité peut être limitée à un certain utilisateur / groupe d'utilisateurs.
Le Niveau d'Accès 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>CurrentRead</td><td>0</td><td>Indique si la valeur courante est consultable
(0 signifie non consultable, 1 signifie consultable).</td></tr>
<tr><td>CurrentWrite</td><td>1</td><td>Indique si la valeur courante est modifiable
(0 signifie non modifiable, 1 signifie modifiable).</td></tr>
<tr><td>HistoryRead</td><td>2</td><td>Indique si l'historique de la valeur est consultable
(0 signifie non consultable, 1 signifie consultable).</td></tr>
<tr><td>HistoryWrite</td><td>3</td><td>Indique si l'historique de la valeur est modifiable (0 signifie non modifiable, 1 signifie modifiable).</td></tr>
<tr><td>SemanticChange</td><td>4</td><td>Indique si la Variable utilisée comme Propriété génère des "Evénements de Modification Sémantique" (voir 9.31).</td></tr>
<tr><td>Reserved</td><td>5:7</td><td>Réservé pour usage ultérieur. Doit toujours être à zéro.</td></tr>
</table>
Les deux premiers bits indiquent également si une valeur courante de cette Variable est disponible et les deux bits suivants indiquent si l'historique de la Variable est disponible via le serveur OPC UA.</td></tr>
<tr><td>UserAccessLevel</td><td>M</td><td>Octet</td><td>L'Attribut "Niveau d'Accès Utilisateur" indique l'accessibilité (lecture/écriture) de la Valeur d'une Variable et si elle contient des données courantes ou historiques en tenant compte des droits d'accès de l'utilisateur.
Le Niveau d'Accès Utilisateur est un entier à 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>CurrentRead</td><td>0</td><td>Indique si la valeur courante est consultable
(0 signifie non consultable, 1 signifie consultable).</td></tr>
<tr><td>CurrentWrite</td><td>1</td><td>Indique si la valeur courante est modifiable
(0 signifie non modifiable, 1 signifie modifiable).</td></tr>
<tr><td>HistoryRead</td><td>2</td><td>Indique si l'historique de la valeur est consultable
(0 signifie non consultable, 1 signifie consultable).</td></tr>
<tr><td>HistoryWrite</td><td>3</td><td>Indique si l'historique de la valeur est modifiable (0 signifie non modifiable, 1 signifie modifiable).</td></tr>
<tr><td>Reserved</td><td>4:7</td><td>Réservé pour usage ultérieur. Doit toujours être à zéro.</td></tr>
</table>
Les deux premiers bits indiquent également si une valeur courante de cette Variable est disponible et les deux bits suivants indiquent si l'historique de la Variable est disponible via le serveur OPC UA.</td></tr>
<tr><td>MinimumSamplingInterval</td><td>O</td><td>Durée</td><td>L'Attribut "Intervalle d'Echantillonnage Minimal" indique la fréquence d'actualisation de la Valeur de la Variable. Il spécifie (en millisecondes) la fréquence raisonnable d'échantillonnage des modifications de la valeur (pour une description détaillée de l'intervalle d'échantillonnage, voir la CEI 62541-4).
Un Intervalle d'Echantillonnage Minimal de 0 indique que le serveur surveille l'élément en continu. Un Intervalle d'Echantillonnage Minimal de -1 signifie qu'il est indéterminé.</td></tr>
</table>

Dénomination	Usage	Type de données	Description
Historizing	M	Booléen	L'Attribut "*Historisation*" indique si le *serveur* recueille activement les données historiques de la *Variable*. Il est différent de l'*Attribut "Niveau d'Accès"* qui indique si la *Variable* dispose d'éventuelles données historiques. Une valeur "VRAI" indique que le *serveur* recueille activement des données. Une valeur "FAUX" indique que le *serveur* ne recueille pas activement des données. La valeur par défaut est FAUX.
Références			
HasModellingRule	0..1		Les *Variables* peuvent pointer vers, au plus, un *Objet Règle de Modélisation* en utilisant une *Référence* "*Dispose d'une Règle de Modélisation*" (pour plus de détails concernant les *Règles de Modélisation* voir 6.4.4).
HasProperty	0..*		Les *Références "Dispose d'une Propriété"* sont utilisées pour identifier les *Propriétés* d'une *Variable de Données*. Il n'est pas admis que les *Propriétés* soient le *SourceNode* des *Références "Dispose d'une Propriété"*.
HasComponent	0..*		Les *Références «HasComponent»* sont utilisées par des *Variables de Données* complexes pour identifier les *Variables de Données* qui *les* composent. Les *Propriétés* ne sont pas autorisées à utiliser cette *Référence*.
HasTypeDefinition	1		La *Référence "Dispose d'une Définition de Type"* 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 "Dispose d'une Définition de Type"* pointant vers un *Type de Variable*. Voir 4.5 pour une description des définitions de types.
HasModelParent	0..1		La *Référence Dispose d'un Modèle Parent* pointe vers le *Modèle Parent* de la *Variable* (voir 6.6 pour plus de détails concernant les *Modèles Parents*).
<other References>	0..*		Les *Variables de Données* peuvent être le *SourceNode* de toute autre *Référence*. Les *Propriétés* ne peuvent être le *SourceNode* que d'une quelconque *Référence non hiérarchique*.
Propriétés normalisées			
NodeVersion	O	Chaîne	La *Propriété* «NodeVersion» est utilisée pour indiquer la version d'une *Variable de Données*. Elle ne s'applique pas aux *Propriétés*. La *Propriété* «NodeVersion» est mise à jour à chaque fois qu'une *Référence* est ajoutée 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 une modification de la *Version de Nœud*. Les clients peuvent consulter la *Propriété «NodeVersion»* ou s'y abonner pour établir le moment où la structure d'un *Nœud* a changé. Bien que la relation entre une *Variable* et son *Type de Données* 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 "Version de Nœud"*.
LocalTime	O	Fuseau Horaire - Type de Données	La *Propriété "Heure Locale"* est uniquement utilisée pour des *Variables de Données*. Elle ne s'applique pas aux *Propriétés*. Cette *Propriété* est une structure contenant le Décalage ainsi que le pointeur "Décalage d'Heure d'Eté". Le Décalage spécifie la différence temporelle (en minutes) entre l'Horodatage Source (UTC) associé à une valeur au lieu où la valeur a été obtenue. L'Horodatage Source est défini dans la CEI 62541-4. Si le "Décalage d'Heure d'Eté" est VRAI, dans ce cas l'Heure d'Eté/normalisée au lieu d'origine est en vigueur et le Décalage inclut la correction de l'heure d'été. S'il est FAUX, dans ce cas le Décalage n'inclut pas la correction de l'heure d'été et celle-ci peut avoir ou ne pas avoir été en vigueur.
DataTypeVersion	O	Chaîne	Uniquement utilisé pour des *Variables* du *Type de Variable "Type de Dictionnaire de DataType"* et "*Type de Description de Type de Données",* comme décrit en 5.8.
DictionaryFragment	O	Chaîne d'Octets	Uniquement utilisé pour des *Variables* du *Type de Variable* "*Type de Description de Type de Données"* comme décrit en 5.8.
AllowNulls	O	Booléen	La *Propriété* "*Permet des Valeurs Nulles"* est uniquement utilisée pour des *Variables de Données*. Elle ne s'applique pas aux *Propriétés*.

Dénomination	Usage	Type de données	Description
			Cette *Propriété* indique si une valeur NULLE est admise pour l'*Attribut "Valeur"* de la *Variable de Données*. Si elle est mise sur vrai, le serveur peut renvoyer des valeurs NULLE et accepter l'écriture de valeurs NULLE. S'il est mis sur faux, 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'admission ou non des valeurs NULLE est spécifique au serveur.

La *Classe de Nœud "Variables"* hérite des *Attributs* de base de la *Classe de Nœud de Base* définie en 5.2.

La *Classe de Nœud "Variables"* définit également un ensemble d'*Attributs* qui décrit la valeur du temps d'exécution de la *Variable*. L'*Attribut "Valeur"* représente la valeur de la *Variable*. Les *Attributs DataType* (*Type de Données*), *ValueRank* (*Rang de Valeur)* et *ArrayDimensions* (*Dimensions de Matrice)* permettent de décrire des valeurs simples et complexes.

L'*Attribut AccessLevel* (*Niveau d'Accès)* 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 de *Niveau d'Accès* à partir du système sous-jacent, il est nécessaire de déclarer qu'il est consultable et modifiable. Si une opération de lecture ou d'écriture est invoquée pour la *Variable*, il convient que le serveur transfère cette requête et renvoie le *Code d'Etat* correspondant si une telle requête est rejetée. Les *Codes d'Etat* sont définis dans la CEI 62541-4.

Le bit *SemanticChange* (*Modification Sémantique)* de l'*Attribut AccessLevel (Niveau d'Accès)* doit être établi lorsque la *Propriété* décrit la sémantique du *Nœud* qui possède la *Propriété* et, en cas de modification de la valeur de la *Propriété*, des "*Evénements de Modification Sémantique"* seront générés. Par exemple, le bit d'une *Propriété* décrivant l'unité technique d'une *DataVariable* sera établi tandis que celui d'une *Propriété* contenant une Icône de la *DataVariable* ne le sera pas. Ce comportement est exactement le même que celui décrit par le bit "*Modifications Sémantiques"* du *Code d'Etat* défini dans la CEI 62541-4. Cependant, si l'utilisateur s'abonne à une *Variable*, il doit examiner le *Code d'Etat* afin d'identifier une éventuelle modification de la sémantique et recevoir ces informations avant de traiter la valeur de la *Variable*.

L'*Attribut UserAccessLevel* (*Niveau d'Accès Utilisateur)* 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 est recommandé d'utiliser le même masque de bit que dans l'*Attribut "Niveau d'Accès"*. L'*Attribut UserAccessLevel* (*Niveau d'Accès Utilisateur)* peut limiter l'accessibilité indiquée par l'*Attribut "Niveau d'Accès"*, mais ne peut l'outrepasser.

L'Attribut *MinimumSamplingInterval* (*Intervalle d'Echantillonnage Minimal)* spécifie la rapidité du serveur pour un échantillonnage raisonnable des modifications de la *valeur*. La précision 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* (*Historisation)* indique si le *serveur* recueille activement les données historiques de la *Variable*. Voir la CEI 62541-11 pour plus de détails sur les *Variables* d'historisation.

Les clients peuvent lire ou écrire des valeurs de *Variables*, ou surveiller la modification des valeurs, comme spécifié dans la CEI 62541-4. La CEI 62541-8 définit des règles supplémentaires d'utilisation des *Services* en automatisation.

Pour spécifier sa *Règle de Modélisation*, une *Variable* peut utiliser au plus une *Référence "Dispose d'une Règle de Modélisation"* pointant vers un *Objet "Règle de Modélisation"*. Les *Règles de Modélisation* sont définies en 6.4.4.

Une *Variable* doit utiliser au plus une *Référence "Dispose d'un Modèle Parent"* pour spécifier son *Modèle Parent* (voir 6.6 pour plus de détails).

Si la *Variable* est créée sur la base d'une *Déclaration d'Instance* (voir 4.5), elle doit avoir le même *Nom d'Exploration* que sa *Déclaration d'Instance*.

Les autres *Références* sont décrites séparément pour des *Propriétés* et *Variables de Données* dans les paragraphes ci-après.

5.6.3 Propriétés

Les *Propriétés* sont utilisées pour définir les caractéristiques de *Nœuds*. Les *Propriétés* sont définies en utilisant la *Classe de Nœud "Variables"* spécifiée dans le Tableau 8. 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 *"Dispose d'une Propriété"*, ce qui 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 *"Dispose d'une Définition de Type"* pointe vers le *Type de Variable* de la *Propriété*. Sachant que les *Propriétés* sont identifiées de manière unique par leur *Nom d'Exploration*, toutes les *Propriétés* doivent pointer vers le *Type de Propriété* défini dans la CEI 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 "Dispose d'une Propriété"*. Pour les différencier des *Variables de Données*, elles ne doivent pas être le *TargetNode* de toute *Référence* "*HasComponent*". Ainsi, une *Référence "Dispose d'une Propriété"* pointant vers une *Variable Nœud* définit ce *Nœud* comme une *Propriété*.

Le *Nom d'Exploration* 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 "Dispose d'une Propriété"* ayant le même *Nom d'Exploration*.

5.6.4 Variable de Données

Les *Variables de données* représentent le contenu d'un *Objet*. Les *Variables de Données* sont définies en utilisant la *Classe de Nœud "Variables"* spécifiée dans le Tableau 8.

Les *Variables de Données* identifient leurs *Propriétés* en utilisant des *Références "Dispose d'une Propriété"*. Les *Variables de Données* complexes utilisent des *Références* "*HasComponent*" pour présenter les *Variables de Données* qui les composent.

La *Propriété NodeVersion (Version de Nœud)* indique la version de la *Variable de Données*. La *Propriété LocalTime* (*Heure Locale)* indique la différence entre l'Horodatage Source de la valeur et l'heure normale au lieu où la valeur a été obtenue. La *Propriété DataTypeVersion* (*Version de Type de Données)* est utilisée uniquement pour les *Dictionnaires de Types de Données* et les *Descriptions de Types de Données,* comme défini en 5.8. La *Propriété* normalisée *DictionaryFragment (Fragment de Dictionnaire)* est utilisée uniquement pour les *Descriptions de Types de Données* comme défini en 5.8. La *Propriété AllowNulls (Permet des Valeurs Nulles)* indique si les valeurs NULLES sont admises pour l'*Attribut "Valeur"*. Il n'y a pas d'autres *Propriétés* définies pour les *Variables de Données* dans le présent document. D'autres parties de la présente série de normes peuvent définir des *Propriétés* supplémentaires pour les *Variables de Données*. La CEI 62541-8 définit un ensemble de *Propriétés* qui peuvent être utilisées pour les *Variables de Données*.

Les *Variables de Données* 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 *Classes de Nœuds* correspondant aux *Nœuds* pouvant être référencés. Cependant, le *Type de Référence* peut définir des restrictions qui excluent son utilisation pour des *Variables de Données*. Des *Types de Références* normalisés sont décrits à l'Article 7.

Il est prévu qu'une *DataVariable* soit définie dans le contexte d'un *Objet*. Cependant, des *Variables de Données* complexes peuvent présenter d'autres *Variables de Données*, et les *Types d'Objets* et *Types de Variables* complexes peuvent également contenir des *Variables de Données*. De ce fait, chaque *DataVariable* doit être le *TargetNode* d'au moins une *Référence* "*HasComponent*" provenant d'un *Objet*, d'un *Type d'Objet*, d'une *DataVariable* ou d'un *Type de Variable*. Les *Variables de Données* ne doivent pas être le *TargetNode* d'une quelconque *Référence "Dispose d'une Propriété"*. Par conséquent, une *Référence* "*HasComponent*" pointant vers une *Variable Nœud* l'identifie comme étant une *Variable de Données*.

La *Référence "Dispose d'une Définition de Type"* pointe vers le *Type de Variable* utilisé comme définition de type de la *Variable de Données*.

Si la *DataVariable* est utilisée comme *Déclaration d'Instance* (voir 4.5), tous les *Nœuds* référencés avec des *Références hiérarchiques* dans le sens direct doivent avoir des *Noms d'Exploration* uniques dans le contexte de cette *Variable de Données*.

5.6.5 VariableType NodeClass

Les *Types de Variables* sont utilisés pour fournir des définitions de type pour les *Variables*. Les *Types de Variables* sont définis en utilisant VariableTypeNodeClass spécifiée dans le Tableau 9.

Tableau 9 – VariableTypeNodeClass

Dénomination	Usage	Type de données	Description
Attributs			
Base NodeClass Attributes	M		Hérité de la *Classe de Nœud de Base*. Voir 5.2
Value	O	Défini par l'*attribut "Data-Type"*	La *valeur* par défaut des instances de ce type.
DataType	M	Identifiant de Nœud	*Identifiant de Nœud* de la définition de type de données pour des instances de ce type.
ValueRank	M	Int32	Cet *Attribut* indique si l'*Attribut "Valeur"* du *Type de 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 ayant le nombre spécifié de dimensions. Unidimensionnelle (1): La valeur est une matrice unidimensionnelle. Une Ou Plusieurs Dimensions (0): La valeur est une matrice à une ou plusieurs dimensions. Scalaire (−1): La valeur n'est pas une matrice. Indifférente (−2): La valeur peut être scalaire ou une matrice ayant n'importe quel nombre de dimensions. Scalaire Ou Unidimensionnelle (−3): La valeur peut être scalaire ou une matrice unidimensionnelle.
ArrayDimensions	O	UInt32[]	Cet *Attribut* spécifie la longueur de chaque dimension d'une valeur matricielle. L'Attribut permet de décrire la capacité du *Type de Variable* et non la taille courante. Le nombre d'éléments doit être égal à la valeur de l'*Attribut* "*Rang de Valeur*". Doit être nul si le *Rang de Valeur* ≤0. Une valeur 0 pour une dimension particulière indique que la dimension a une longueur variable. Par exemple, si un *Type de Variable* est défini par la matrice C suivante: Int32 maMatrice[346];

Dénomination	Usage	Type de données	Description
			ainsi, ce *Type de Données* de *Type de Variable* pointerait vers une Int32, le *Rang de Valeur* du *Type de Variable* prend la valeur 1 et l'*Attribut Dimensions de Matrice* indique une matrice dont l'une des entrées a la valeur 346.
IsAbstract	M	Booléen	Un *Attribut* booléen prenant les valeurs suivantes: VRAI Il s'agit d'un *Type de Variable* abstrait, ce qui signifie qu'aucune *Variable* de ce type ne doit exister, mais uniquement ses sous-types. FAUX Il s'agit d'un *Type de Variable* qui n'est pas abstrait, ce qui signifie qu'il peut y avoir des *Variables* de ce type.
Références			
HasProperty	0..*		Les *Références "Dispose d'une Propriété"* sont utilisées pour identifier les *Propriétés* du *Type de Variable*. Les *Nœuds* référencés peuvent être instanciés par des instances de ce type, en fonction des *Règles de Modélisation* définies en 6.4.4.
HasComponent	0..*		Les *Références "HasComponent"* sont utilisées pour des *Types de Variables* complexes afin d'identifier les *Variables de Données* qui *les* composent. Les *Types de Variables* complexes ne peuvent être utilisés que pour des *Variables de Données*. Les *Nœuds* référencés peuvent être instanciés par des instances de ce type, en fonction des *Règles de Modélisation* définies en 6.4.4.
HasSubtype	0..*		Les *Références «HasSubtype»* indiquent les *Types de Variables* qui sont des sous-types de ce type. La *Référence* inverse "*Sous-type de"* identifie le type parent de ce type.
GeneratesEvent	0..*		Les *Références "Génère un Evénement"* identifient le type d'instances d'*Evénements* que ce type peut générer.
<other References>	0..*		Les *Types de Variables* peuvent contenir d'autres *Références* qui peuvent être instanciées par des *Variables* définies par ce *Type de Variable*. Les *Règles de Modélisation* sont définies en 6.4.4.
Propriétés normalisées			
NodeVersion	O	Chaîne	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 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 une modification de la *Version de Nœud*. Les clients peuvent consulter la *Propriété «NodeVersion»* ou s'y abonner pour établir le moment où la structure d'un *Nœud* a changé. Bien que la relation entre un *Type de Variable* et son *Type de Données* ne soit pas modélisée en utilisant des *Références*, les modifications apportées à l'*Attribut "DataType"* d'un *Type de Variable* entraînent une mise à jour de la *Propriété "Version de Nœud"*.

VariableTypeNodeClass hérite des *Attributs* de la *Classe de Nœud de Base* définie en 5.2. *VariableTypeNodeClass* définit également un ensemble d'*Attributs* qui décrit la valeur par défaut ou la valeur initiale de son instance "*Variables"*. L'*Attribut* "*Valeur"* représente la valeur par défaut. Les *Attributs DataType* (*Type de Données)*, *ValueRank* (*Rang de Valeur)* et *ArrayDimensions* (*Dimensions de Matrice)* permettent de décrire des valeurs simples et complexes. L'*Attribut IsAbstract (Est Abstrait)* indique si le type peut être directement instancié.

VariableTypeNodeClass utilise des *Références "Dispose d'une Propriété"* pour définir les *Propriétés* et utilise les *Références* "*HasComponent*" pour définir les *Variables de Données*. Leur éventuelle instanciation dépend des *Règles de Modélisation* définies en 6.4.4.

La *Propriété «NodeVersion»* indique la version *Type de Variable*. Il n'y a pas d'autres *Propriétés* définies pour les *Types de Variables* dans le présent document. D'autres parties de la présente série de normes peuvent définir des *Propriétés* supplémentaires pour les *Types de Variables*. La CEI 62541-8 définit un jeu de *Propriétés* qui peut être utilisé pour les *Types de Variables*.

Les *Références HasSubtype (Dispose d'un Sous-type)* sont utilisées pour définir des sous-types de *Types de Variables*. Les sous-types de *Type de Variable* 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 que la *Référence «HasSubtype»* soit fournie pour le supertype; il est cependant exigé que ce sous-type fournisse la *Référence* inverse à son sur-type.

Les Références GeneratesEvent (Génère un Evénement) indiquent que les *Variables* du *Type de Variable* peuvent être la source d'un *Evénement* du *Type d'Evénement* spécifié ou d'un de ses sous-types. Il est recommandé que les serveurs fassent en sorte que les *Références "Génère un Evénement"* soient bidirectionnelles. Il est cependant admis qu'elles soient unidirectionnelles lorsque le serveur n'est pas capable de présenter le sens inverse pointant du *Type d'Evénement* vers chaque *Type de Variable* prenant en charge le *Type d'Evénement.*

Les Références "Génère un Evénement" sont facultatives, ce qui signifie que les *Variables* peuvent générer des *Evénements* d'un *Type d'Evénement* qui n'est pas présenté par ce *Type de Variable*.

Les *Types de Variables* 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 *Classes de Nœuds* correspondant aux *Nœuds* pouvant être référencés. Cependant, le *Type de Référence* peut définir des restrictions qui excluent son utilisation pour des *Types de Variables*. Des *Types de Références* normalisés sont décrits à l'Article 7.

Tous les *Nœuds* référencés avec des *Références hiérarchiques* doivent avoir des *Noms d'Exploration* uniques dans le contexte du *Type de Variable* (voir 4.5).

5.6.6 Création du côté client de Variables d'un Type de Variable donné

Les *Variables* sont toujours fondées sur un *Type de Variable*; ce qui signifie qu'elles ont une *Référence "Dispose d'une Définition de Type"* pointant vers son *Type de Variable*.

Les clients peuvent créer des *Variables* en utilisant le *Service "Ajouter des Nœuds"* défini dans la CEI 62541-4. Le *Service* exige que la *Définition Type du Noeud* de la *Variable* soit spécifié. Une *Variable* créée par le *Service "Ajouter des Nœuds"* contient tous les composants définis par son *Type de Variable* en fonction des *Règles de Modélisation* 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 *Type de Variable*. Ce comportement dépend du serveur. Le *Type de Variable* spécifie uniquement l'ensemble minimal de composants qui doit exister pour chaque *Variable* d'un *Type de Variable* donné.

5.7 Classe de Nœud "Méthodes"

Les *Méthodes* définissent des fonctions invocables. Les *Méthodes* sont sollicitées par le *Service "Invocation"* défini dans la CEI 62541-4. Les invocations des Méthodes ne sont pas représentées dans l'*Espace d'Adresse*. Les invocations de *Méthodes* vont toujours jusqu'à achèvement et renvoient toujours des réponses une fois terminées. Les *Méthodes* sont définies en utilisant la *Classe de Nœud "Méthodes"*, spécifiée dans le Tableau 10.

Tableau 10 – Classe de Nœud "Méthodes"

Dénomination	Usage	Type de données	Description
Attributs			
Base NodeClass Attributes	M	--	Hérité de la *Classe de Nœud de Base*. Voir 5.2
Executable	M	Booléen	L'Attribut "Exécutable" indique si la *Méthode* est actuellement exécutable (“Faux” signifie non exécutable, “Vrai” signifie exécutable). L'Attribut "Exécutable" ne tient compte d'aucun droit d'accès utilisateur, ce qui signifie que même si la *Méthode* est exécutable, elle peut être limitée à certains utilisateurs / groupe d'utilisateurs.
UserExecutable	M	Booléen	L'*Attribut "Exécutable par l'Utilisateur"* indique si la *Méthode* est actuellement exécutable en tenant compte des droits d'accès de l'utilisateur (“Faux” signifie non exécutable, “Vrai” signifie exécutable).
Références			
HasProperty	0..*		Les *Références "Dispose d'une Propriété"* identifient les *Propriétés* de la *Méthode*.
HasModellingRule	0..1		Les *Méthodes* peuvent pointer vers, au plus, un *Objet Règle de Modélisation* en utilisant une *Référence "Dispose d'une Règle de Modélisation"* (pour plus de détails concernant les *Règles de Modélisation* voir 6.4.4).
HasModelParent	0..1		La *Référence "Dispose d'un Modèle Parent"* pointe vers le *Modèle Parent* de la *Méthode* (voir 6.6 pour plus de détails concernant les *Modèles Parents*).
GeneratesEvent	0..*		*Les Références "Génère un Evénement"* identifient les types d'*Evénements* qui seront générés à chaque fois que la *Méthode* est invoquée.
AlwaysGenerates-Event	0..*		*Les Références "Génère Toujours un Evénement"* identifient les types d'*Evénements* qui seront générés chaque fois que la *Méthode* est invoquée.
<other References>	0..*		Les *Méthodes* peuvent contenir d'autres *Références*.
Propriétés normalisées			
NodeVersion	O	Chaîne	La *Propriété "Version de Nœud"* est utilisée pour indiquer la version d'un *Nœud*. La *Propriété* «NodeVersion» est mise à jour à chaque fois qu'une *Référence* est ajoutée ou retirée du *Nœud* auquel la *Propriété* appartient. Les modifications de la valeur de l'*Attribut* n'entraînent pas une modification de la *Version de Nœud*. Les clients peuvent consulter la *Propriété «NodeVersion»* ou s'y abonner pour établir le moment où la structure d'un *Nœud* a changé.
InputArguments	O	Argument[]	La *Propriété "Arguments d'entrée"* est utilisée pour spécifier les arguments qui doivent être utilisés par un client lorsqu'il invoque la *Méthode*.
OutputArguments	O	Argument[]	La *Propriété "Arguments de sortie"* spécifie le résultat renvoyé par l'invocation de la *Méthode*.

La *Classe de Nœud "Méthodes"* hérite des *Attributs* de base de la *Classe de Nœud de Base* définie en 5.2. La *Classe de Nœud "Méthodes"* ne définit pas d'*Attributs* supplémentaires.

L'*Attribut Executable* (*Exécutable*) 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 est demandé que le serveur transfère cette requête et renvoie le *Code d'Etat* correspondant si une telle requête est rejetée. Les *Codes d'Etat* sont définis dans la CEI 62541-4.

L'*Attribut UserExecutable (Exécutable par l'Utilisateur)* 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 "Exécutable"*. L'*Attribut "Exécutable par l'Utilisateur"* peut être mis sur “Faux”, même si l'*Attribut "Exécutable"* est mis sur “Vrai”; il doit cependant être mis sur “Faux” si l'*Attribut "Exécutable"* est mis sur “Faux”.

Des *Propriétés* peuvent être définies pour les *Méthodes* en utilisant les *Références "Dispose d'une Propriété"*. Les *Propriétés InputArguments* (*Arguments d'entrée)* et *OutputArguments (Arguments de sortie)* spécifient les arguments d'entrée et les arguments de sortie de la *Méthode*. Les deux contiennent une matrice du *Type de Données Argument* comme spécifié en 8.6. Une matrice vide, pour une *Propriété* qui n'est pas fournie, indique qu'il n'y a pas d'arguments d'entrée ou d'arguments de sortie pour la *Méthode*. La *Propriété «NodeVersion»* indique la version *de* la *Méthode*. Il n'y a pas d'autres *Propriétés* définies pour les *Méthodes* dans le présent document. D'autres parties de la présente série de normes peuvent définir des *Propriétés* supplémentaires pour les *Méthodes*.

Pour spécifier sa *Règle de Modélisation*, une *Variable* peut utiliser au plus une *Méthode "Dispose d'une Règle de Modélisation"* pointant vers un *Objet "Règle de Modélisation"*. Les *Règles de Modélisation* sont définies en 6.4.4.

Une *Méthode* doit utiliser au plus une *Référence "Dispose d'un Modèle Parent"* pour spécifier son *Modèle Parent* (voir 6.6 pour plus de détails).

Les Références "Génère un Evénement" indiquent que les *Méthodes* peuvent générer un *Evénement* du *Type d'Evénement* spécifié ou d'un des sous-types pour chaque invocation de la *Méthode*. Un serveur peut générer un *Evénement* pour chaque *Type d'Evénement* référencé lorsqu'une *Méthode* est invoquée avec succès.

Les Références AlwaysGeneratesEvent (*Génère Toujours un Evénement)* indique que les *Méthodes* généreront un *Evénement* du *Type d'Evénement* spécifié ou de l'un de ses sous-types pour chaque invocation de *Méthode*. Un serveur doit toujours générer un *Evénement* pour chaque *Type d'Evénement* référencé lorsqu'une *Méthode* est invoquée avec succès.

Il faut que les serveurs fassent en sorte que les *Références "Génère un Evénement"* soient bidirectionnelles. Il est cependant admis qu'elles soient unidirectionnelles lorsque le serveur n'est pas capable de présenter le sens inverse, pointant du *Type d'Evénement* vers chaque *Méthode* générant le *Type d'Evénement*.

Les Références "Génère un Evénement" sont facultatives, ce qui signifie que l'invocation d'une *Méthode* peut produire des *Evénements* d'un *Type d'Evénement* qui n'est pas référencé avec une *Référence "Génère un Evénement"* 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 *Classes de Nœuds* correspondant aux *Nœuds* pouvant être référencés. Cependant, le *Type de Référence* peut définir des restrictions qui excluent son utilisation pour des *Méthodes*. Des *Types de Références* 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 *Type d'Objet*. Si une *Méthode* est invoquée, l'*Identifiant de Nœud* de l'un de ces *Nœuds* doit être placé dans le *Service* "Invocation" défini dans la CEI 62541-4, en tant que paramètre, afin de détecter le contexte d'exécution de la *Méthode*.

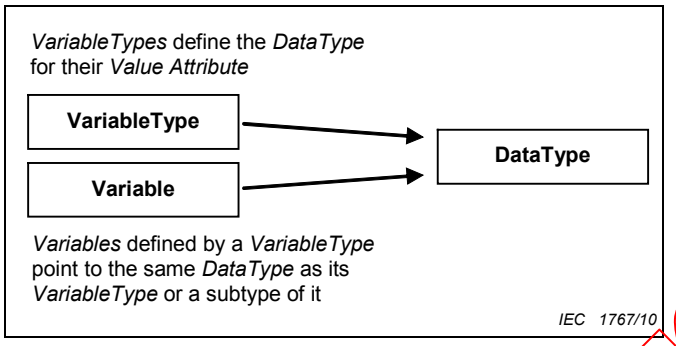
Si la *Méthode* est utilisée comme *Déclaration d'Instance* (voir 4.5), tous les *Nœuds* référencés avec des *Références hiérarchiques* dans le sens direct doivent avoir des *Noms d'Exploration* uniques dans le contexte de cette *Méthode*.

5.8 Types de Données

5.8.1 Modèle de Type de Données

Le Modèle de Type de Données est utilisé pour définir des types de données simples et complexes. Les types de données sont utilisés pour décrire la structure de l'*Attribut "Valeur"* de *Variables* et leurs *Types de Variables*. Par conséquent, chaque *Variable* et *Type de*

Variable pointe au moyen de son *Attribut "DataType"* vers un *Nœud* de la *Classe de Nœud "DataType",* comme illustré en Figure 9.



Légende

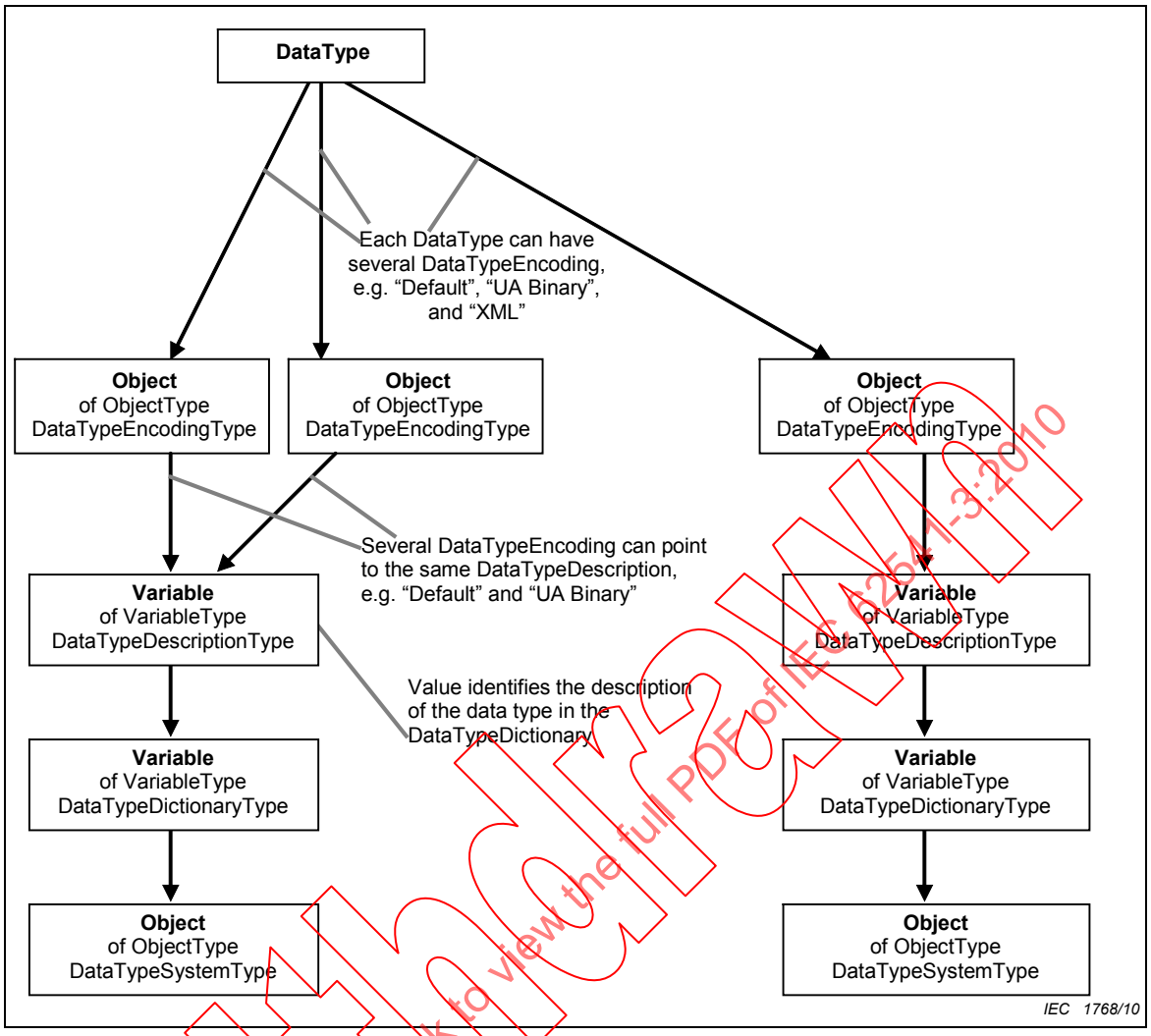
Anglais	Français
Variabletypes define the Datatype for their value attribute	Les Types de Variable définissent le Type de Données pour leur Attribut Valeur
Variabletype	Type de Variable
Variable	Variable
Datatype	Type de Données
Variables defined by a variabletype point to the same datatype as its variabletype or a subtype of it	Les Variables définies par un Type de Variable pointent vers le même Type de Données que leur Type de Variable ou un sous-type de celui-ci

Figure 9 – Variables, Types de Variables et leurs Types de Données

Dans de nombreuses situations, l'*Identifiant de Nœud* du *Nœud* de *Type de Données* – l'*Identifiant de Type de Données* – sera bien connu des clients et des serveurs. L'Article 8 définit les *Types de Données* et la CEI 62541-6 définit leurs *Identifiants de Type de Données*. Par ailleurs, d'autres organismes peuvent définir des *Types de Données* largement connus dans l'industrie. Les *Identifiants de Type de Données* largement connus 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 *Identifiants de Types de Données* largement connus sans avoir à représenter les *Nœuds de Types de Données* correspondants dans leurs *Espaces d'Adresses*.

Dans d'autres situations, les *Types de Données* et leurs *Identifiants de Type de Données* correspondants peuvent être définis par le fournisseur. Il est recommandé que les serveurs tentent de présenter les *Nœuds de Type de Données* et les informations concernant la structure de ces *Types de Données* pour que les clients puissent les lire, même si ces informations peuvent ne pas toujours être disponibles pour le serveur.

La Figure 10 illustre les *Nœuds* utilisés dans *l'Espace d'Adresse* pour décrire la structure d'un *Type de Données.* Le *Type de Données* pointe vers un *Objet* du type "*Type de Codage de DataType".* Chaque *Type de Données* peut avoir plusieurs *Codages de Type de Données*, par exemple codage "Par Défaut", "Binaire UA" et "XML". Les services définis dans la CEI 62541-4 permettent aux clients de demander un codage ou de choisir un codage "Par Défaut". Chaque *Codage de Type de Données* est utilisé précisément par un *Type de Données*, ce qui signifie qu'il n'est pas permis que deux *Types de Données* pointent vers le même *Codage de Type de Données*. L'*Objet* "*Codage de Type de Données"* pointe précisément vers une *Variable* de type "*Type de Description de Type de Données"*. La *Variable* "*Description de Type de Données"* appartient à une *Variable* "*Dictionnaire de Type de Données"*.



Légende

Anglais	Français
Datatype	Type de Données
Each datatype can have several datatypeencoding, e.g. “default”, “UA Binary”, and “XML”	Chaque Type de Données peut avoir plusieurs Codages de Type de Données, par exemple « par défaut », « Binaire UA » et « XML »
Object	Objet
Of objecttype datatypeencodingtype	Du Type d’Objet Type de Codage de Type de Données
Several datatypeencoding can point to the same datatypedescription, e.g. “default” and “UA Binary”	Plusieurs Codages de Type de Données peuvent pointer vers la même Description de Type de Données, par exemple « par défaut » et « Binaire UA »
Variable	Variable
Of variabletype datatypedescriptiontype	Du Type de Description de Type de Données
Value identifies the description of the data type in the datatypedictionary	La valeur identifie la description du type de données dans le Dictionnaire de Type de Données
Of variabletype datatypedictionarytype	Du Type de Variable Type de Dictionnaire de Type de Données
Of objecttype datatypesystemtype	Du Type d’Objet Type de Système de Type de données

Figure 10 – Modèle de Type de Données

Sachant que l'*Identifiant de Nœud* du *Codage de Type de Données* sera utilisé dans certaines correspondances pour identifier le *Type de Données* et son codage, comme défini dans la CEI 62541-6, ces *Identifiants de Nœuds* peuvent également être bien connus lorsqu'ils correspondent à des *Identifiants de Type de Données* bien connus.

Le *Dictionnaire de Type de Données* décrit un jeu de *Types de Données de* manière suffisamment détaillée pour que les clients puissent analyser/interpréter les *valeurs* de *Variables* qu'ils reçoivent et composer les *valeurs* qu'ils envoient. Le *Dictionnaire de Type de Données* est représenté comme une *Variable* de type « *Type de Dictionnaire de Type de Données* » dans *l'Espace d'Adresse*; la description relative aux *Types de Données* est contenue dans son *Attribut "Valeur"*. Tous les *Types de Données* contenants, présentés dans *l'Espace d'Adresse*, sont représentés comme des *Variables* du type « *Type de Description de Type de Données* ». La *Valeur* de ces Variables donne la description d'un *Type de Données* dans l'*Attribut "Valeur"* du *Dictionnaire de Type de Données*.

Le *Type de Données* d'une *Variable "Dictionnaire de Type de Données"* est toujours une Chaîne d'Octets. Le format et les conventions utilisés pour définir les *Types de Données* dans cette Chaîne d'Octets sont définis par des *Systèmes de Type de Données*. Les *Systèmes de Type de Données* sont identifiés par des *Identifiants de Nœuds*. Ils sont représentés dans *l'Espace d'Adresse* comme des *Objets* du *Type d'Objet "Type de Système de Type de Données"*. Chaque *Variable* représentant un *Dictionnaire de Type de Données* référence un Objet "*Système de Type de Données"* pour identifier son *Système de Type de Données*.

Un client doit reconnaître le *Système de Type de Données* pour analyser d'éventuelles informations de description de type. Les clients OPC UA qui ne reconnaissent pas un *Système de Type de Données* seront incapables d'interpréter ses descriptions de type, et par conséquent, les valeurs qu'elles décrivent. Dans ce cas, les clients interprètent ces valeurs comme une Chaîne d'Octets opaque.

Les schémas XML OPC binaire et W3C sont des exemples de *Systèmes de Type de Données*. Le *Système de Type de Données* OPC binaire est défini à l'Annexe C. Le système OPC Binaire utilise le langage XML pour décrire des valeurs de données binaires. Le Schéma XML du W3C est spécifié dans la Partie 1 du Schéma XML et dans la Partie 2 du Schéma XML.

5.8.2 Règles de codage pour différentes sortes de Types de Données

Il existe une distinction nette entre les différentes sortes de *Types de Données;* ces *Types de Données* sont traités de manière différente en termes de codage et selon que ce codage est ou non représenté dans *l'Espace d'Adresse*.

Les *Types de Données Intégrés* sont un jeu fixe de *Types de Données* (voir la CEI 62541-6 pour une liste complète des *Types de Données Intégrés*). Ces types n'ont pas de codages visibles dans *l'Espace d'Adresse* car il faut que le codage soit connu de l'ensemble des produits OPC UA. Des exemples de *Types de Données Intégrés* sont *Int32* (voir 8.26) et *Double* (voir 8.12).

Les *Types de Données Simples* sont des sous-types de *Types de Données Intégrés.* Ils sont traités à la volée comme les *Types de Données Intégrés*, ce qui signifie qu'il n'est pas possible, à l'utilisation, de les distinguer de leurs supertypes *Intégrés*. Etant traités, en termes de codage, comme des *Types de Données Intégrés,* leurs codages ne peuvent pas être définis dans *l'Espace d'Adresse*. Les clients peuvent lire l'*Attribut "DataType"* d'une *Variable* ou d'un *Type de Variable* pour identifier le *Type de Données Simple* de l'*Attribut "Valeur"*. Un exemple de *Type de Données Simple* est la *Durée*. Ce type de données 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 *Durée* (voir 8.13).

Les *Types de Données Structurés* sont des *Types de Données* qui représentent des données structurées et ne sont pas définis comme des *Types de Données Intégrés. Les Types de*

Données Structurés héritent directement ou indirectement de la *Structure du Type de Données* défini en 8.33. Les *Types de Données Structurés* peuvent avoir plusieurs codages et ces derniers sont présentés dans *l'Espace d'Adresse*. La manière dont le codage des *Types de Données Structurés* est traité à la volée est définie dans la CEI 62541-6. Le codage de *Type de Données Structuré* est transmis avec chaque valeur et ainsi les Clients sont informés du *Type de Données* sans avoir à lire l'*Attribut "DataType"*. Le codage doit être transmis, de façon à ce que le Client soit en mesure d'interpréter les données. Un exemple de *Type de Données Structuré* est l'*Argument* (voir 8.6).

Les *Types de Données " Enumération"* sont des *Types de Données* qui représentent des jeux discrets de valeurs définies. Les Enumérations sont toujours codées en Int32 à la volée, comme défini dans la CEI 62541-6. *Les Types de Données Enumération* héritent directement ou indirectement *du Type de Données Enumération* défini en 8.14. Les Enumérations n'ont pas de codages présentés dans *l'Espace d'Adresse*. Pour exposer la représentation interprétable par l'utilisateur d'une valeur énumérée, le *Nœud* de *Type de Données* peut disposer d'une *Propriété* "*Chaîne d'Enumération"* contenant une matrice de "*Texte Localisé"*. La représentation Entier de la valeur d'énumération pointe vers une position de cette matrice. La *Classe de Nœud* définie en 8.30 est un exemple de *Type de Données* d'énumération.

Outre les *Types de Données* décrit ci-dessus, les *Types de Données* abstraits sont également pris en charge; ces derniers n'ont aucun codage et ne peuvent pas être échangés à la volée. Les *Variables* et *Types de Variables* utilisent des *Types de Données* abstraits pour indiquer que leur *Valeur* peut être n'importe lequel des sous-types du *Type de Données* abstrait. Un exemple de *Type de Données* abstrait est l'Entier défini en 8.24.

5.8.3 DataTypeNodeClass

DatatypeNodeClass décrit la syntaxe d'une *Variable* de *Valeur*. Les *Types de Données* peuvent être simples ou complexes, en fonction du *Système de Type de Données*. Les *Types de Données* sont définis en utilisant la *Classe de Nœud "Types de Données"*, spécifiée dans le Tableau 11.

Tableau 11 – Classe de Nœud "Types de Données"

Dénomination	Usage	Type de données	Description
Attributs			
Base NodeClass Attributes	M	--	Hérité de la *Classe de Nœud de Base*. Voir 5.2
IsAbstract	M	Booléen	Un Attribut booléen prenant les valeurs suivantes: VRAI il s'agit d'un *Type de Données* abstrait. FAUX il s'agit d'un *Type de Données* qui n'est pas abstrait.
Références			
HasProperty	0..*		Les *Références "Dispose d'une Propriété"* identifient les *Propriétés* du *Type de Données*.
HasSubtype	0..*		Les *Références «HasSubtype»* peuvent être utilisées pour couvrir une hiérarchie de type de données.
HasEncoding	0..*		Les *Références "Dispose d'un Codage"* indiquent les codages du *Type de Données* représentés comme des *Objets* de type "*Type de Codage de Type de Données"*. Seuls des *Types de Données Structurés* concrets peuvent utiliser les *Références "Dispose d'un Codage"*. Il n'est pas admis que des *Types de Données* Abstraits, *Intégrés, Enumération,* et *Simples* soient le *SourceNode* d'une *Référence "Dispose d'un Codage"*. Chaque *Type de Données Structuré* concret doit pointer vers au moins un *Objet "Codage de Type de Données"*, avec le *Nom d'Exploration* “Binaire par Défaut” ou “XML par Défaut” ayant l'*Indice de l'espace de Nom* 0. Le *Nom d'Exploration* des *Objets "Codage de Type de Données"* doit être unique dans le contexte d'un *Type de Données*, ce qui signifie qu'un *Type de Données* ne doit pas pointer vers deux *Codages de Type de Données* ayant le même *Nom d'Exploration*.

Dénomination	Usage	Type de données	Description
Propriétés normalisées			
NodeVersion	O	Chaîne	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 ou retirée du *Nœud* auquel la *Propriété* appartient. Les modifications de la valeur de l'*Attribut* n'entraînent pas une modification de la *Version de Nœud*. Les clients peuvent consulter la *Propriété «NodeVersion»* ou s'y abonner pour établir le moment où la structure d'un *Nœud* a changé.
EnumStrings	O	Texte Localisé[]	La *Propriété "Chaîne d'Enumération"* s'applique uniquement aux *Types de Données "Enumération"*. Elle ne doit pas être appliquée à d'autres *Types de Données*. Chaque entrée de la matrice de *Texte Localisé* dans cette *Propriété* constitue la représentation interprétable par l'utilisateur d'une valeur énumérée. La représentation Entier de la valeur d'énumération pointe vers une position de la matrice.

DatatypeNodeClass hérite des *Attributs* de base de la *Classe de Nœud de Base* définie en 5.2. L'*Attribut* supplémentaire *IsAbstract (Est Abstrait)* spécifie si le *Type de Données* est ou non abstrait. Les *Types de Données* Abstraits peuvent être utilisés dans *l'Espace d'Adresse*, ce qui signifie que les *Variables* et *Types de Variables* peuvent pointer, au moyen de leur *Attribut «DataType»,* vers un *Type de Données* Abstrait. Cependant, des valeurs concrètes ne peuvent jamais être un *Type de Données* Abstrait et doivent toujours être du sous-type concret du *Type de Données* Abstrait.

Les *Références HasProperty (Dispose d'une Propriété)* sont utilisées pour identifier les *Propriétés* d'un *Type de Données*. La *Propriété «NodeVersion»* est utilisée pour indiquer la version du *Type de Données*. La Version n'est pas affectée par la *Propriété* "*Version de Type de Données"* des "*Dictionnaires de Types de Données"* et des "*Descriptions de Types de Données"*. La *Propriété EnumStrings (Chaînes d'Enumération)* contient des représentations interprétables par l'utilisateur des valeurs d'énumération et est uniquement appliquée aux *Types de Données "Enumération"*. Il n'y a pas d'autres *Propriétés* définies pour les *Types de Données* dans la présente norme. D'autres parties de la présente série de normes peuvent définir des *Propriétés* supplémentaires pour les *Types de Données*.

Les *Références HasSubtype (Dispose d'un Sous-type)* peuvent être utilisées pour présenter une hiérarchie de type de données dans *l'Espace d'Adresse*. Cette hiérarchie doit refléter l'arborescence spécifiée dans le *Dictionnaire de Type de Données*. La sémantique de sous-typage dépend du *Système de Type de Données*. Il n'est pas nécessaire que les serveurs fournissent les *Références «HasSubtype»*, même si leurs *Types de Données* couvrent une hiérarchie de Types de Données. Il faut que les clients ne fassent aucune hypothèse relative à une éventuelle sémantique utilisant ces informations autres que celles fournies dans le *Dictionnaire de Type de Données*. Par exemple, il pourrait s'avérer impossible de relier la valeur d'un type de données à son type de données de base.

Les *Références HasEncoding (Dispose d'un Codage)* pointent du *Type de Données* vers son *Codage de Type de Données*. En suivant cette *Référence*, le client peut explorer le *Dictionnaire de Type de Données* décrivant la structure du *Type de Données* pour le codage utilisé. Chaque T*ype de Données Structuré* concret peut pointer vers plusieurs *Codages de Types de Données*, mais chaque *Codage de Type de Données* doit appartenir à un *Type de Données;* en d'autres termes, il n'est pas admis que deux *Nœuds de Type de Données* pointent vers le même *Objet "Codage de Type de Données"* au moyen de *Références "Dispose d'un Codage"*.

Un *Type de Données* Abstrait n'est pas le *SourceNode* d'une *Référence "Dispose d'un Codage"*. Le *Codage de Type de Données* d'un *Type de Données* Abstrait est fourni par ses sous-types concrets.

Les *Nœuds de Type de Données* ne doivent pas être le *SourceNode* d'autres Types de *Références*. Ils peuvent cependant être le *TargetNode* d'autres *Références*.

5.8.4 Dictionnaire de Type de Données, Description de Type de Données, Codage de Type de Données et Système de Type de Données

Un *Dictionnaire de Type de Données* est une entité qui contient un ensemble de descriptions de type, par exemple un schéma XML. Les *Dictionnaires de Types de Données* sont définis comme des *Variables* du *Type de Variable* "*Type de Dictionnaire de Type de Données"*.

Un *Système de Type de Données* spécifie le format et les conventions utilisés pour définir les *Types de Données* dans des *Dictionnaires de Types de Données.* Les *Systèmes de Type de Données* sont définis comme des *Objets* du *Type d'Objet* "*Type de Système de Type de Données"*.

Le *Type de Référence* utilisé pour relier les *Objets* du *Type d'Objet* "*Type de Système de Type de Données"* aux *Variables* du *Type de Variable* "*Type de Dictionnaire de Type de Données"* est le *Type de Référence "Dispose d'un Composant"*. Ainsi, la *Variable* est toujours le *TargetNode* d'une *Référence "Dispose d'un Composant"*; il s'agit là d'une exigence pour les *Variables*. Cependant, pour les *Dictionnaires de Types de Données,* le serveur doit toujours fournir la *Référence* inverse, sachant qu'il est nécessaire de connaître le *Système de Type de Données* lors du traitement du *Dictionnaire de Type de Données*.

Un exemple de *Dictionnaire de Type de Données* est un document XML contenant un schéma XML. Dans ce cas, le *Système de Type de Données* est le Schéma XML du W3C et les déclarations d'éléments de niveau supérieur dans le document schéma sont des descriptions de type de données. Chacune de ces descriptions est définie dans différentes versions d'un schéma XML en utilisant le même Espace de Nom cible XML. L'Espace de Nom cible est utilisé comme le composant Espace de Nom de l'*Identifiant de Type de Données* dans l'*Espace d'Adresse* du serveur. Etant donné que le même Espace de Nom cible peut être utilisé dans d'autres schémas XML, les clients doivent savoir que deux *Identifiants de Type de Données* ayant le même Espace de Nom ne sont pas nécessairement définis dans le même *Dictionnaire de Type de Données*.

Une modification d'une description de type peut entraîner d'autres modifications mais il est plus probable que des modifications du dictionnaire résultent de la somme ou de la suppression de descriptions de type. Ceci comprend les modifications apportées à un moment où le serveur est hors-ligne, de sorte que la nouvelle version soit disponible que lorsque le serveur redémarre. Les clients peuvent s'abonner à la *Propriété "Version de Type de Données"* pour déterminer une éventuelle modification au *Dictionnaire de Type de Données* depuis la dernière consultation.

Le serveur peut, sans que cela ne soit exigé, mettre le contenu du *Dictionnaire de Type de Données* à la disposition des clients par le biais de l'*Attribut "Valeur"*. Il faut dans ce cas que les clients considèrent que le contenu du *Dictionnaire de Type de Données* est relativement important et qu'ils seront confrontés à des problèmes de performance s'ils consultent automatiquement le contenu du *Dictionnaire de Type de Données* à chaque fois qu'ils rencontrent une instance d'un *Type de Données* spécifique. Il convient que le client utilise la *Propriété Version de Type de Données* pour établir si la copie contenue dans la mémoire cache locale est encore valide. Si le client détecte une modification dans la *Version de Type de Données*, il doit consulter à nouveau le *Dictionnaire de Type de Données*. Ceci implique que la *Version de Type de Données* doit être mise à jour par un serveur, même après redémarrage, étant donné que les clients peuvent avoir gardé une copie persistante dans la mémoire cache locale.

L'*Attribut* "*Valeur"* du *Dictionnaire de Type de Données* qui contient les descriptions de type est une Chaîne d'Octets dont le formatage est défini par le *Système de Type de Données*. Pour le *Système de Type de Données* “schéma XML”, la Chaîne d'Octets contient un document de schéma XML valide. Pour le *Système de Type de Données* “OPC Binaire”, la Chaîne d'Octets contient une chaîne qui est un document XML valide. Le serveur doit assurer

que toute modification au contenu de la Chaîne d'Octets correspond à une modification de la *Propriété* "*Version de Type de Données"*. En d'autres termes, le client peut utiliser en toute sécurité une copie en cache du *Dictionnaire de Type de Données*, tant que la *Version de Type de Données* demeure identique.

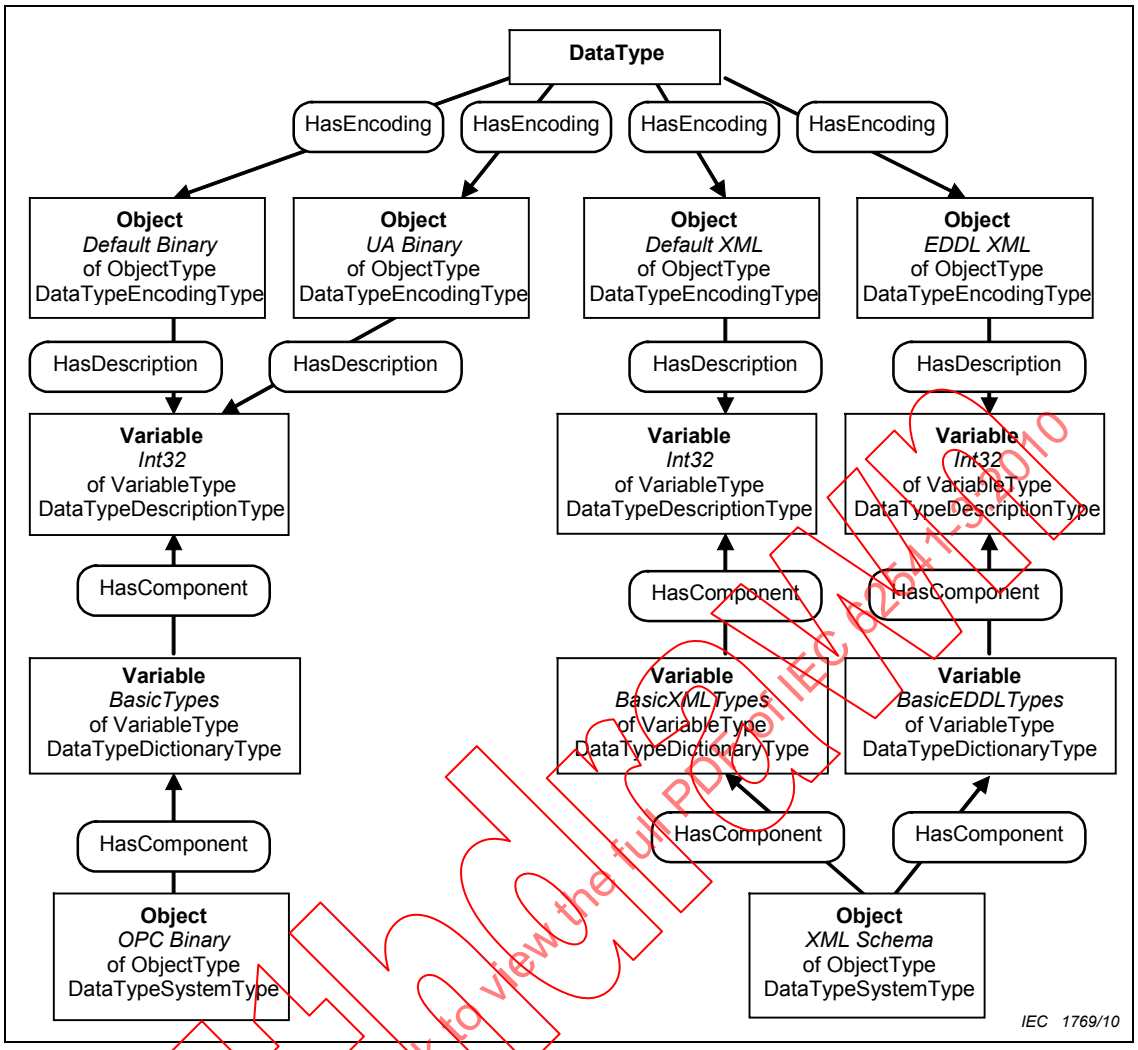
Les *Dictionnaires de Types de Données* sont des *Variables* complexes qui présentent leurs *Descriptions de Types de Données* comme des *Variables* en utilisant des *Références "Dispose d'un Composant"*. Une *Description de Type de Données* fournit les informations nécessaires à l'obtention de la description formelle d'un *Type de Données* dans le *Dictionnaire de Type de Données*. La *Valeur* d'une *Description de Type de Données* dépend du *Système de Type de Données* du *Dictionnaire de Type de Données*. Lorsque des dictionnaires “OPC Binaires" sont utilisés, la *Valeur* doit être le nom de la *Description de Type*. Lorsque des dictionnaires à “schéma XML”sont utilisés, la Valeur doit être une expression XPATH (CHEMIN X) pointant vers un élément XML dans le document du schéma.

Comme les *Dictionnaires de Types de Données,* chaque *Description de Type de Données* fournit la *Propriété Version de Type de Données* indiquant si la description de type du *Type de Données* a changé. Les modifications de la *Version de Type de Données* peuvent avoir un effet sur le fonctionnement des *Abonnements*. Si la *Version de Type de Données* est modifiée en une *Variable* qui fait l'objet d'une surveillance pour un *Abonnement* donné et qui utilise la *Description de Type de Données*, la prochaine *Notification* de modification des données envoyée pour ce qui concerne la *Variable,* contiendra un état qui indique la modification apportée à la *Description de Type de Données*.

Les *Objets "Codage de Type de Données"* des *Types de Données* référencent leurs *Descriptions de Types de Données* des *Dictionnaires de Types de Données* en utilisant des *Références "Dispose d'une Description"*. Cependant, il n'est pas exigé que les serveurs fournissent des *Références* inverses qui relient en retour les *Descriptions de Types de Données* aux *Objets "Codage de Type de Données"*. Si un *Nœud de Type de Données* est présenté dans *l'Espace d'Adresse*, il doit fournir son *Codage de Type de Données* et si un *Dictionnaire de Type de Données* est présenté, il est demandé qu'il présente l'ensemble de ses *Descriptions de Types de Données*. Ces deux *Références* doivent être bidirectionnelles.

Les *Types de Variables* "*Type de Dictionnaire de Type de Données"* et "*Type de Description de Type de Données"* ainsi que les *Types d'Objets* "*Type de Système de Type de Données"* et "*Type de Codage de Type de Données"* sont formellement définis dans la CEI 62541-5.

La Figure 11 donne un exemple de modélisation de *Types de Données* dans *l'Espace d'Adresse*.



Légende

Anglais	Français
Datatype	Type de Données
Hasencoding	Dispose d'un Codage
Object	Objet
Default binary	Binaire par défaut
Of objecttype datatypeencodingtype	Du Type d'Objet Type de Codage de Type de données
UA binary	Binaire UA
Default XML	XML par défaut
EDDL XML	XML EDDL
Hasdescription	Dispose d'une description
Variable	Variable
Int32	Int32 (Entier de 32 bits)
Of variabletype datatypedescriptiontype	Du Type de Variable Type de Description de Type de Données
Hascomponent	Dispose d'un composant
Basictypes	Types de Base
BasicXMLtypes	Types XML de Base
BasicEDDLtypes	Types EDDL de Base
Of variabletype datatypedictionarytype	Du Type de Variable Type de Dictionnaire de Type de Données
Of objecttype datatypesystemtype	Du Type d'Objet Type de Système de Type de Données

Figure 11 – Exemple de modélisation de Type de Données

Dans certains scénarios, un serveur OPC UA peut avoir des ressources limitées et il s'avère peu pratique de présenter des *Dictionnaires de Types de Données* de taille importante. Dans de tels scénarios, il est admis que le serveur puisse fournir un accès à des descriptions de Types de Données particuliers, même si l'ensemble du dictionnaire n'est pas consultable. De ce fait, la présente norme définit une *Propriété* pour la *Description de Type de Données* appelée *Fragment de Dictionnaire* (voir 5.6.2). Cette *Propriété* est une Chaîne d'Octets qui contient un sous-ensemble de *Dictionnaire de Type de Données* qui décrit le format de *Type de Données* associé à la *Description de Type de Données*. Ainsi, le serveur découpe le *Dictionnaire de Type de Données* de grande taille en plusieurs petites parties et les clients peuvent y accéder sans affecter la performance globale du système.

Cependant, il convient que le serveur fournisse l'ensemble du *Dictionnaire de Type de Données* en une seule fois si cela est possible. Il est en général plus efficace de lire l'ensemble du *Dictionnaire de Type de Données* en une seule fois plutôt que de consulter ses différentes parties.

5.9 Synthèse des Attributs de NodeClasses

Le Tableau 12 récapitule tous les *Attributs* définis dans le présent document et précise les *Classes de Nœuds* qui les utilisent, que ce soit de manière facultative (O pour Optional) ou obligatoire (M pour Mandatory).

Tableau 12 – Aperçu des Attributs

Attribut	Variable	Type de variable	Objet	Type d'Objet	Type de Référence	Type de Données	Méthode	Vue
AccessLevel	M							
ArrayDimensions	O	O						
BrowseName	M	M	M	M	M	M	M	M
ContainsNoLoops								M
DataType	M	M						
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
Symmetric					M			
UserAccessLevel	M							
UserExecutable							M	
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 de Types d'Objets et de Types de Variables

6.1 Aperçu général

Dans les articles ci-dessous, le modèle type de *Types d'Objets* et de *Types de Variables* est défini en termes de sous-typage et d'instanciation.

6.2 Définitions

6.2.1 Déclaration d'Instance

Une *Déclaration d'Instance* est un *Objet*, une *Variable* ou une *Méthode* qui référence une *Règle de Modélisation* au moyen d'une *Référence "Dispose d'une Règle de Modélisation"* et qui est le *TargetNode* d'une *Référence hiérarchique* à partir d'une ou d'une autre *Déclaration d'Instance*.

6.2.2 Instances sans Règles de Modélisation

S'il n'existe aucune *Règle de Modélisation*, le *Nœud* n'est pas pris en compte pour l'instanciation de type ou pour le sous-typage.

Si un *Nœud* référencé par un *TypeDefinitionNodes* ne référence pas une *Règle de Modélisation*, cela signifie que ce *Nœud* appartient uniquement au *TypeDefinitionNodes* et non aux instances. Par exemple, un *Nœud de Type d'Objet* peut contenir une *Propriété* qui décrit des scénarios qui pourraient utiliser le type. Cette *Propriété* ne serait pas prise en considération 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'hériteraient pas du *Nœud* référencé.

6.2.3 Hiérarchie de Déclaration d'Instance

La *Hiérarchie de Déclaration d'Instance* d'un *TypeDefinitionNodes* contient le *TypeDefinitionNodes* ainsi que toutes les *Déclarations d'Instances* directement ou indirectement référencées à partir d'un *TypeDefinitionNodes* au moyen des *Références hiérarchiques* dans le sens direct.

6.2.4 Nœud de Déclaration d'Instance similaire

Un *Nœud* similaire d'une *Déclaration d'Instance* est un *Nœud* qui a le même *Nom d'Exploration* et la même *Classe de Nœud* que la *Déclaration d'Instance* et, dans le cas de *Variables* et d'*Objets*, le même *TypeDefinitionNodes* ou un sous-type correspondant.

6.2.5 Chemin d'Exploration

Toutes les cibles de *Références hiérarchiques* directes à partir d'un *TypeDefinitionNodes* doivent avoir un *Nom d'Exploration* qui est unique au sein du *Nœud de Définition de Type*. Cette même restriction s'applique aux cibles de *Références hiérarchiques* dans le sens direct à partir d'une quelconque *Déclaration d'Instance*. Ceci signifie que toute *Déclaration d'Instance* dans la *Hiérarchie de Déclaration d'Instance* peut être identifiée de manière unique par une séquence de *Noms d'Exploration*. Cette séquence de *Noms d'Exploration* est appelée *Chemin d'Exploration*.

6.2.6 Traitement des Attributs de Déclarations d'Instances

Il existe certaines restrictions concernant les *Attributs* de *Déclarations d'Instances* lorsque la *Déclaration d'Instance* est remplacée ou instanciée. Le *Nom d'Exploration* et la *Classe de Nœud* ne doivent jamais changer et toujours demeurer les mêmes que ceux de la *Déclaration d'Instance* initiale.

En outre, les règles définies en 6.2.7 s'appliquent aux *Déclarations d'Instances* de la *Classe de Nœud* "*Variables*".

6.2.7 Traitement des Attributs de Variables et Types de Variables

Il existe certaines restrictions concernant les *Attributs* d'un *Type de Variable* ou d'une *Variable* utilisée comme *Déclaration d'Instance* pour ce qui concerne le type de données de l'*Attribut "Valeur"*.

Lorsqu'une *Variable* utilisée comme *Déclaration d'Instance* ou qu'un *Type de Variable* est remplacé ou instancié, les règles suivantes s'appliquent:

a) L'*Attribut* «DataType» ne peut être modifié qu'en un nouveau *Type de Données* si le nouveau *Type de Données* est un sous-type du *Type de Données* initialement utilisé.

b) L'*Attribut Rang de Valeur* peut seulement être plus strict

   1) 'Toute valeur' peut être mise pour toute autre valeur;

   2) 'Scalaire Ou Unidimensionnelle' peut être mis sur 'Scalaire' ou sur 'Unidimensionnelle';

   3) 'Une Ou Plusieurs Dimensions' peut être mis sur un nombre concret de dimensions (valeur > 0).

   4) Toutes les autres valeurs de cet *Attribut* doivent demeurer inchangées.

c) L'*Attribut "Dimensions de Matrice"* peut être ajouté s'il n'a pas été fourni ou la valeur 0 d'une entrée de la matrice peut être modifiée en une valeur différente. Toutes les autres valeurs de la matrice doivent demeurer inchangées.

6.3 Sous-typage de Types d'Objets et de Types de Variables

6.3.1 Aperçu général

Le *Type de Référence «HasSubtype»* définit des sous-types de types. Le sous-typage ne peut avoir lieu qu'entre *Nœuds* de la même *Classe de Nœud*. Les règles de sous-typage des *Types de Références* sont décrites en 5.3.3.3. Il n'y a aucune définition commune pour le sous-typage des *Types de Données*, comme décrit en 5.8.3. Les paragraphes suivants spécifient les règles de sous-typage pour un héritage simple portant sur les *Types d'Objets* et les *Types de Variables*.

6.3.2 Attributs

Les sous-types héritent des valeurs d'*Attribut* des types de parents, sauf l'*Identifiant de Nœud*. Les valeurs d'*Attribut* héritées peuvent être remplacées par le sous-type; il convient de remplacer les valeurs du *Nom d'Exploration* et du *Nom d'Affichage*. Des règles particulières s'appliquent à certains *Attributs* de *Types de Variables* comme défini en 6.2.7. Des *Attributs* facultatifs, non prévus par le type parent, peuvent être ajoutés au sous-type.

6.3.3 Déclarations d'Instances

6.3.3.1 Aperçu général

Les sous-types héritent intégralement des *Déclarations d'Instances* de type parent héritées.

Tant que ces *Déclarations d'Instances* ne sont pas remplacées, elles ne sont pas référencées par le sous-type. Des *Déclarations d'Instances* peuvent être remplacées en ajoutant des *Références*, en modifiant les *Références* portant sur les différents *Nœuds*, en modifiant les *Références* en sous-types du *Type de Référence* initial, en modifiant les valeurs des *Attributs* ou en ajoutant des *Attributs* facultatifs. Pour obtenir l'ensemble des informations concernant un sous-type, les *Déclarations d'Instances* 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 de *Déclarations d'Instances* est appelé la *Hiérarchie de Déclaration d'Instance* intégralement héritée d'un sous-type donné.

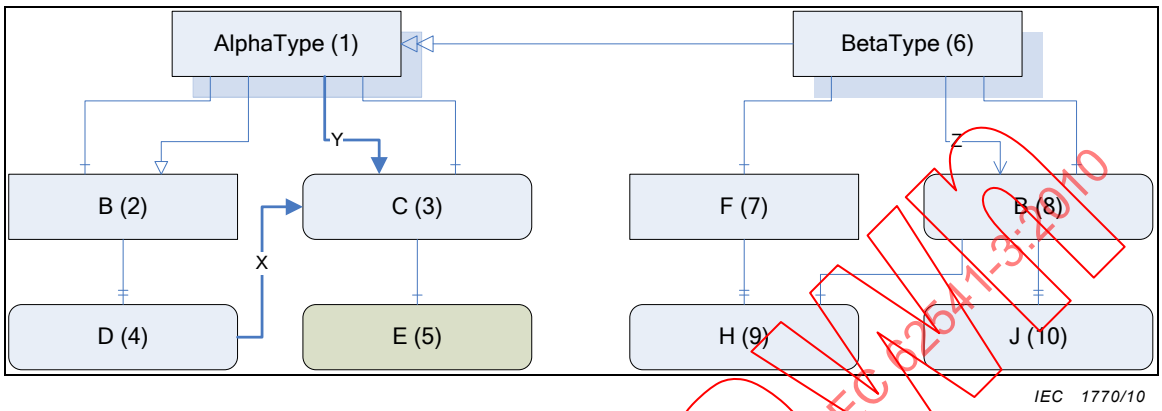
Les paragraphes suivants définissent la méthode de constitution de la *Hiérarchie de Déclaration d'Instance* intégralement héritée et la manière dont les *Déclarations d'Instances* peuvent être remplacées.

6.3.3.2 Hiérarchie de Déclaration d'Instance intégralement héritée

Une instance de *TypeDefinitionNodes* est décrite par la *Hiérarchie de Déclaration d'Instance* intégralement héritée du *Nœud de Définition de Type*. La *Hiérarchie de Déclaration d'Instance* intégralement héritée peut être créée en partant de la *Hiérarchie de Déclaration d'Instance* du

TypeDefinitionNodes et en fusionnant la *Hiérarchie de Déclaration d'Instance* intégralement héritée de son type parent.

Le processus de fusion des *Hiérarchies de Déclaration d'Instance* est direct. Il est illustré par l'exemple présenté en Figure 12 où est spécifié un *TypeDefinitionNodes* "Type Bêta" qui est un sous-type de "Type Alpha". Dans chaque case, la lettre est le *Nom d'Exploration* et le chiffre est l'*Identifiant de Nœud.*



Légende

Anglais	Français
Alphatype	Type Alpha
betatype	Type Bêta

Figure 12 – Sous-typage des *TypeDefinitionNodes*

Une *Hiérarchie de Déclaration d'Instance* peut être pleinement décrite comme un tableau de *Nœuds* identifié par leurs *Chemins d'Exploration* avec un tableau de *Références* correspondant. La *Hiérarchie de Déclaration d'Instance* pour le "Type Bêta" est décrite dans le Tableau 13; la moitié supérieure du tableau est le tableau des *Nœuds* et la moitié inférieure est le tableau des *Références* (les références "*Dispose d'une Règle de Modélisation*" ont été omises du tableau pour plus de clarté; tous les Nœuds, à l'exception des Nœuds 1, 6 et 5 ont des *Règles de Modélisation*). Toutes les *Déclarations d'Instances* de la *Hiérarchie de Déclaration d'Instance* et tous les *Nœuds* référencés avec une *Référence* non hiérarchique à partir d'une telle *Déclaration d'Instance* sont ajoutées au tableau. Les *Références Hiérarchiques* aux *Nœuds* sans *Règle de Modélisation* ne sont pas prises en compte.

Tableau 13 – Hiérarchie de Déclaration d'Instance pour un Type Bêta

Chemin d'Exploration	Identifiant de Nœud
/	6
/F	7
/B	8
/F/H	9
/B/J	10
/B/H	9

Chemin Source	Type de référence	Chemin Cible	Identifiant de Nœud Cible
/	HasComponent	/F	-
/	HasComponent	/B	-
/	Z	/B	-
/	HasTypeDefinition	-	Type Bêta
/F	HasComponent	/F/H	-
/F	HasTypeDefinition	-	Type d'Objet de Base
/B	HasProperty	/B/J	-
/B	HasTypeDefinition	-	Type d'Objet de Base
/F/H	HasTypeDefinition	-	Type de Propriété
/B/J	HasTypeDefinition	-	Type de Propriété
/B	HasComponent	/B/H	-
/B/H	HasTypeDefinition	-	Type de DataVariable de Base

Plusieurs *Chemins d'Exploration* vers le même *Nœud* doivent être traités comme des *Nœuds* séparés. Une *Instance* peut fournir différents *Nœuds* pour chaque *Chemin d'Exploration*.

La *Hiérarchie de Déclaration d'Instance* intégralement héritée pour un “Type Bêta” peut à présent être constituée par fusion avec la *Hiérarchie de Déclaration d'Instance* pour le “Type Alpha”. Le résultat est présenté dans le Tableau 14; les entrées ajoutées à partir du “Type Alpha” sont grisées.

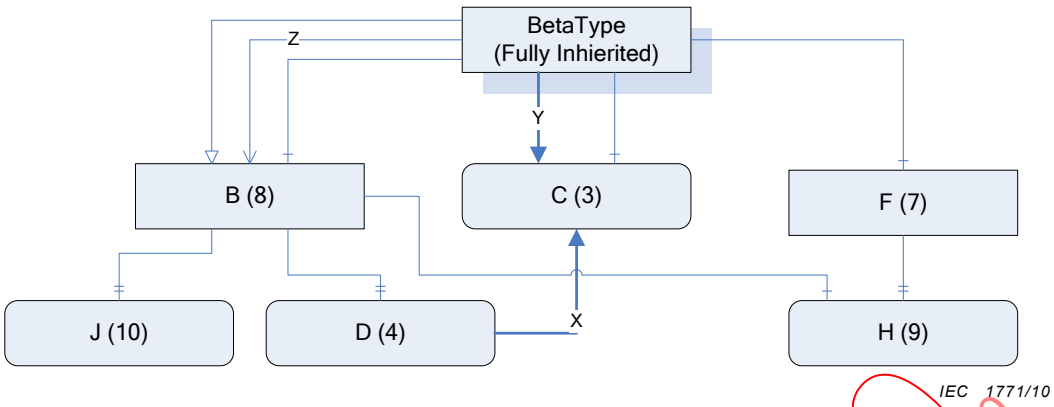
Tableau 14 – Hiérarchie de Déclaration d'Instance intégralement héritée pour un Type Bêta

Chemin d'Exploration	Identifiant de Nœud
/	6
/F	7
/B	8
/F/H	9
/B/J	10
/B/H	9
/B/D	4
/C	3

Chemin Source	Type de référence	Chemin Cible	Identifiant de Nœud Cible
/	HasComponent	/F	-
/	HasComponent	/B	-
/	Z	/B	-
/	HasTypeDefinition	-	Type Bêta
/F	HasComponent	/F/H	-
/F	HasTypeDefinition	-	Type d'Objet de Base
/B	HasProperty	/B/J	-
/B	HasTypeDefinition	-	Type d'Objet de Base
/F/H	HasTypeDefinition	-	Type de Propriété
/B/J	HasTypeDefinition	-	Type de Propriété
/B	HasComponent	/B/H	-
/B/H	HasTypeDefinition	-	Type de DataVariable de Base
/	HasNotifier	/B	-
/B	HasProperty	/B/D	-
/	HasComponent	/C	-
/	Y	/C	-
/C	HasTypeDefinition	-	Type de DataVariable de Base
/B/D	HasTypeDefinition	-	Type de Propriété
/B/D	X	/C	-

Le *Chemin d'Exploration* "/B" existe déjà dans le tableau, de sorte qu'il n'est pas nécessaire de l'ajouter. Cependant, la référence *HasNotifier* (*Dispose d'un Notificateur)* de "/" vers "/B" n'existe pas et a été ajoutée.

Les *Nœuds* et *Références* définis dans le Tableau 14 peuvent être utilisés pour générer la *Hiérarchie de Déclaration d'Instance* intégralement héritée illustrée en Figure 13. *La Hiérarchie de Déclaration d'Instance* intégralement héritée contient toutes les informations nécessaires à un *TypeDefinitionNodes* pour ce qui concerne sa structure complexe, sans qu'il ne soit nécessaire d'ajouter d'éventuelles informations supplémentaires à partir de ses supertypes.



Betatype (fully inherited) Type Bêta (intégralement héritée)

Figure 13 – Hiérarchie de Déclaration d'Instance intégralement héritée pour le Type Bêta

6.3.3.3 Remplacement des Déclarations d'Instances

Un sous-type remplace une *Déclaration d'Instance* en spécifiant une *Déclaration d'Instance* ayant le même *Chemin d'Exploration*. Une *Déclaration d'Instance* remplacée doit avoir la même *Classe de Nœud* et le même *Nom d'Exploration*. Le *TypeDefinitionNodes* de la *Déclaration d'Instance* remplacée doit être le même ou un sous-type du *TypeDefinitionNodes* doit être spécifié dans le supertype.

Lorsqu'une *Déclaration d'Instance* 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 *Chemin d'Exploration* du *Nœud*).

Il n'est possible de remplacer que des *Déclarations d'Instances* directement référencées à partir du *Nœud de Définition de Type*. Si une *Déclaration d'Instance* indirectement référencée, comme "J" dans la Figure 13, doit être remplacée, les *Déclarations d'Instances* directement référencées qui incluent "J", en l'occurrence "B", doivent être remplacées en premier lieu et "J" peut ensuite être remplacé dans une seconde phase.

Une *Référence* est remplacée si elle s'inscrit entre deux *Nœuds* remplacés et qu'elle a le même *Type de Référence* qu'une *Référence* définie dans le supertype. La *Référence* spécifiée dans le sous-type peut être un sous-type du *Type de Référence* utilisé dans le type parent.

Toute *Référence non hiérarchique* spécifiée pour la *Déclaration d'Instance* remplacée est traitée comme une nouvelle *Référence*, à moins que le *Type de Référence* ne permette qu'une seule *Référence* par *Nœud Source*. Dans ce cas, le sous-type peut modifier la cible de la *Référence* mais la nouvelle cible doit avoir la même *Classe de Nœud* 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 *Classe de Nœud* ou le *Nom d'Exploration;* toutefois les restrictions relatives aux *Attributs* spécifiées en 6.2.6 s'appliquent. Tout *Attribut* fourni par la *Déclaration d'Instance* remplacée doit être fourni par la *Déclaration d'Instance* remplaçante, il est admis d'ajouter des *Attributs* facultatifs supplémentaires.

La *Règle de Modélisation* de la *Déclaration d'Instance* remplaçante peut être modifiée comme défini en 6.4.4.3.

Chaque *Déclaration d'Instance* remplaçante nécessite ses propres *Références HasModellingRule* (*Dispose d'une Règle de Modélisation)* et *HasTypeDefinition (Dispose d'une Définition de Type)*, même si elles n'ont pas été modifiées.

Il convient de ne pas remplacer un *Nœud* par un sous-type, à moins que ce sous-type doive le modifier.

La sémantique de certains *Nœuds de Définition de Type* et de *Types de Références* peut imposer des restrictions supplémentaires concernant les *Nœuds* remplaçants.

6.4 Instances de Types d'Objets et de Types de Variables

6.4.1 Aperçu général

Toute *Instance* d'un *TypeDefinitionNodes* peut être la racine d'une hiérarchie qui reflète la *Hiérarchie de Déclaration d'Instance* du *Nœud de Définition de Type*. Chaque *Nœud* de la hiérarchie de l'Instance aura un *Chemin d'Exploration* qui peut être le même que celui de l'une des *Déclarations d'Instances* de la hiérarchie du *Nœud de Définition de Type.* La *Déclaration d'Instance* ayant le même *Chemin d'Exploration* est la *Déclaration d'Instance* invoquée pour le *Nœud*. Si un *Nœud* a une *Déclaration d'Instance*, il doit avoir le même *Nom d'Exploration* et la même *Classe de Nœud* que la *Déclaration d'Instance* et, dans le cas des *Variables* et des *Objets*, le même *TypeDefinitionNodes* ou un sous-type correspondant.

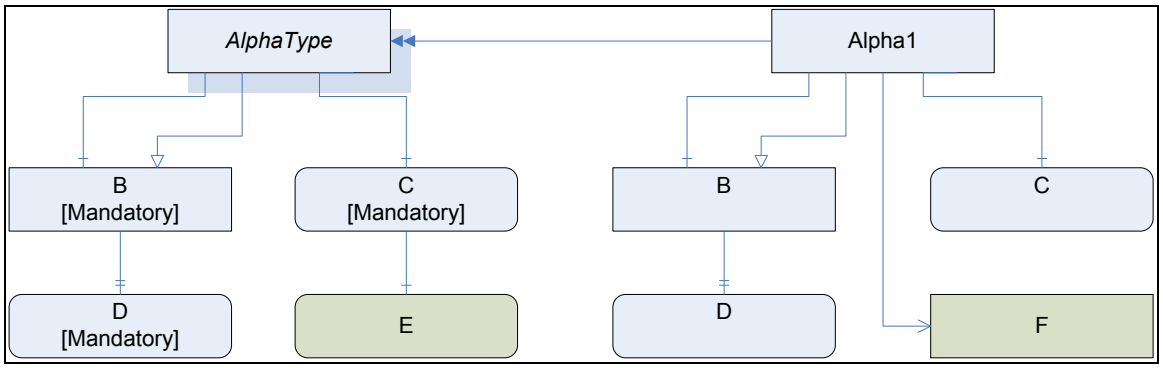
Les Instances peuvent référencer plusieurs *Nœuds* ayant le même *Chemin d'Exploration*. Les clients qui doivent faire la distinction entre les *Nœuds* fondés sur la *Hiérarchie de Déclaration d'Instance* et les *Nœuds* qui ne sont pas fondés sur la *Hiérarchie de Déclaration d'Instance* peuvent effectuer cette distinction en utilisant le service "Traduire Chemins d'Exploration en Identifiants de Nœuds" (TranslateBrowsePathsToNodeIds), défini dans la CEI 62541-4.

6.4.2 Création d'une Instance

Les Instances héritent des valeurs initiales des *Attributs* qu'ils partagent avec le *TypeDefinitionNodes* à partir duquel ils sont instanciés, à l'exception de la *Classe de Nœud* et de l'*Identifiant de Nœud.*

Lorsqu'un *Serveur* crée une instance de *Nœud de Définition de Type,* il doit créer la même hiérarchie de *Nœuds* sous le nouvel *Objet* ou la nouvelle *Variable,* en fonction de la *Règle de Modélisation* de chaque *Déclaration d'Instance*. Des *Règles de Modélisation* normalisées sont définies en 6.4.4.5. Les *Nœuds* dans la hiérarchie nouvellement créée peuvent être des copies des *Déclarations d'Instance*, de la *Déclaration d'Instance* proprement dite ou d'un autre *Nœud* dans l'*Espace d'Adresse* ayant le même *Nœud de Définition de Type* et le même *Nom d'Exploration*. Si de nouvelles copies sont créées, les valeurs d'*Attribut* des *Déclarations d'Instances* sont utilisées comme valeurs initiales.

Un exemple simple de *TypeDefinitionNodes et d'une Instance* est fourni en Figure 14. Les *Nœuds* référencés par le *TypeDefinitionNodes* sans *Règle de Modélisation* n'apparaissent pas dans l'instance. Les *Instances* peuvent avoir des enfants ayant des *Noms d'Exploration* dupliqués; cependant, l'un de ces enfants correspondra à la *Déclaration d'Instance*.



Légende

Anglais	Français
Alphatype	Type Alpha
Alpha1	Alpha 1
mandatory	obligatoire

Figure 14 – Instance et son Nœud de Définition de Type

Il incombe au *Serveur* de décider quelles *Déclarations d'Instances* apparaissent dans toute instance unique. Dans certains cas, le *Serveur* ne définira pas l'ensemble de l'instance et fournira des références distantes aux *Nœuds*, dans un autre *Serveur*. Les *Règles de Modélisation* décrites en 6.4.4.5 permettent aux *Serveurs* d'indiquer 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 de *Nœuds de Définition de Type* pour accéder à des *Nœuds* qui sont dans la hiérarchie de l'instance. Il doit transmettre l'*Identifiant de Nœud* de l'instance et le *Chemin d'Exploration* des *Nœuds* enfants, fondés sur le *Nœud de Définition de Type,* au *service "Traduire Chemins d'Exploration en Identifiants de Nœuds" (TranslateBrowsePathsToNodeIds)* (voir la CEI 62541-4). Le *Service* renvoie l'*Identifiant de Nœud* pour chacun des *Nœuds* enfants. Si un *Nœud* existe, le *Nom d'Exploration* et la *Classe de Nœud* doivent correspondre à la *Déclaration d'Instance*. Dans le cas d'*Objets* ou de *Variables,* le *TypeDefinitionNodes* doit également correspondre au *TypeDefinitionNodes* initial ou en être un sous-type.

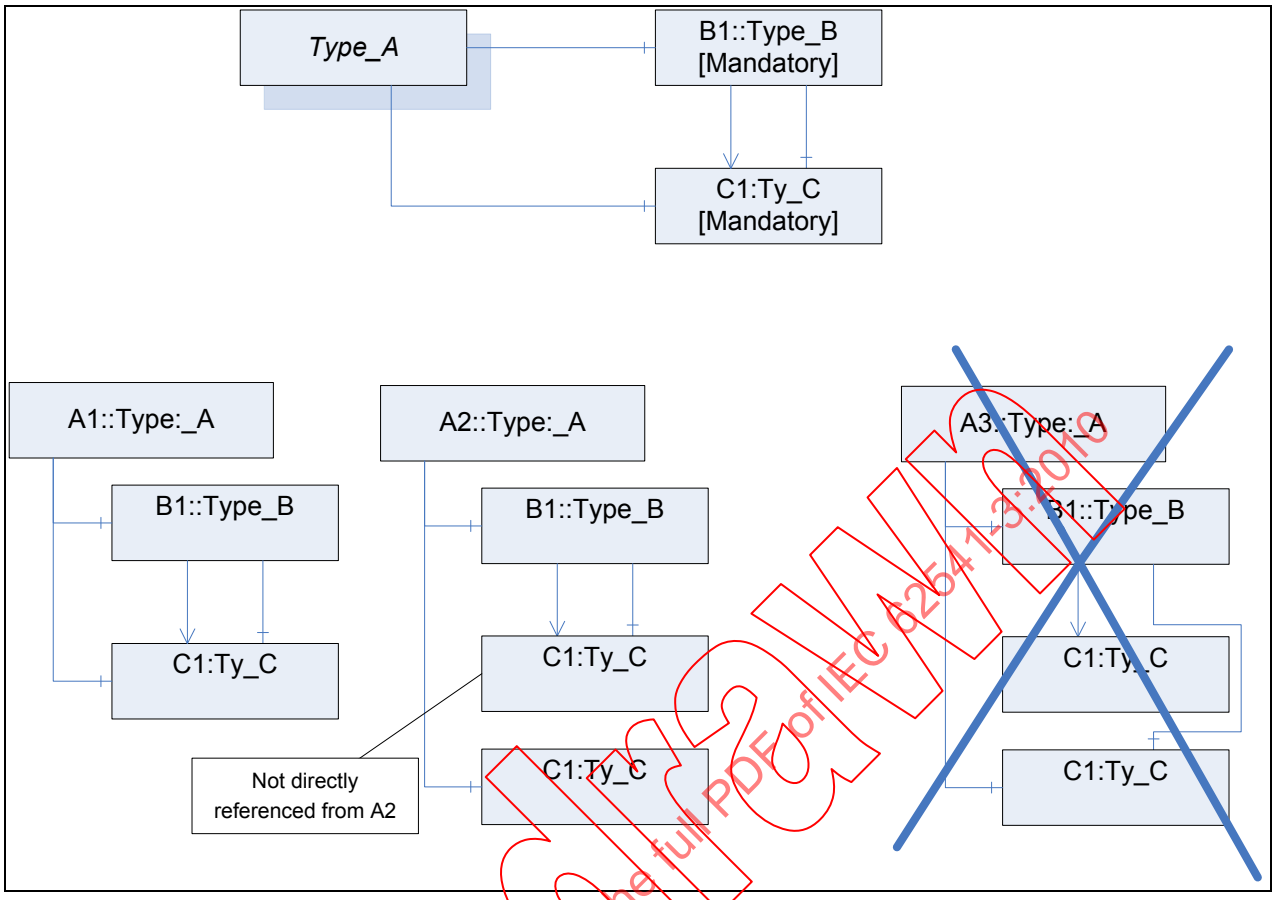
6.4.3 Contraintes applicables à une Instance

Les *Objets* et *Variables* peuvent modifier leurs valeurs d'*Attribut* après création. Des règles particulières s'appliquent pour certains *Attributs* comme défini en 6.2.6.

Des *Références* supplémentaires peuvent être ajoutées aux *Nœuds* et les *Références* peuvent être supprimées tant que les *Règles de Modélisation* définies sur des *Déclarations d'Instances* du *TypeDefinitionNodes* demeurent satisfaites.

Pour les *Variables* et *Objets,* la *Référence HasTypeDefinition (Dispose d'une Définition de Type)* doit toujours pointer vers le même *TypeDefinitionNodes* que la *Déclaration d'Instance* ou un sous-type correspondant.

Si deux *Déclarations d'Instances* de la *Hiérarchie de Déclaration d'Instance* intégralement héritée ont été directement reliées par plusieurs *Références*, toutes ces *Références* doivent relier les mêmes *Nœuds*. Un exemple est fourni en Figure 15. 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. Il est à noter que cette restriction s'applique uniquement aux Nœuds directement reliés. Par exemple, A2 référence une C1 directement et une C1 différente via B1.



Légende

Anglais	Français
Mandatory	Obligatoire
Not directly referenced from A2	Pas directement référencé à partir de A2

Figure 15 – Exemple de plusieurs Références entre Déclarations d'Instances

6.4.4 Règles de Modélisation

6.4.4.1 Généralités

Pour une définition des *Règles de Modélisation*, voir 6.4.4.5. D'autres parties de la présente série de normes peuvent définir des *Règles de Modélisation* supplémentaires. Dans OPC UA les *Règles de Modélisation* sont un concept extensible; par conséquent les fournisseurs peuvent définir leurs propres *Règles de Modélisation*.

Il faut noter que les *Règles de Modélisation* définies dans la présente norme ne spécifient pas la manière de traiter des *Références* non hiérarchiques entre *Déclarations d'Instances*, c'est-à-dire qu'il incombe au serveur d'établir si ces *Références* existent ou non dans une hiérarchie d'instance. D'autres *Règles de Modélisation* peuvent également définir le comportement de *Références* non hiérarchiques entre *Déclarations d'Instance*.

Les *Règles de Modélisation* sont représentées dans *l'Espace d'Adresse* comme des *Objets* du *Type d'Objet "Type de Règle de Modélisation"*. Quelques *Propriétés* définissent la sémantique commune aux *Règles de Modélisation*. Cette version de la norme spécifie uniquement une *Propriété* pour les *Règles de Modélisation*. Il est admis que des versions futures définissent des *Propriétés* supplémentaires pour les *Règles de Modélisation*. La CEI 62541-5 spécifie la représentation des *Objets* "*Règle de Modélisation*", leurs *Propriétés* et leurs types dans

l'Espace d'Adresse. La sémantique des *Propriétés* pour les *Règles de Modélisation* est définie en 6.4.4.2.

Le paragraphe 6.4.4.4 définit la méthode de modification de la *Règle de Modélisation* lorsqu'il s'agit d'instancier des *Déclarations d'Instances* en termes de *Propriétés*. Le paragraphe 6.4.4.3 définit la méthode de modification de la *Règle de Modélisation* lorsqu'il s'agit de remplacer des *Déclarations d'Instances* par des sous-types en termes de *Propriétés*.

6.4.4.2 Propriétés décrivant des Règles de Modélisation

6.4.4.2.1 Règle de Nommage

La *Règle de Nommage* est une *Propriété* obligatoire d'une *Règle de Modélisation*. Elle spécifie le but d'une *Déclaration d'Instance*. Chaque *Déclaration d'Instance* référence une *Règle de Modélisation* et ainsi la *Règle de Nommage* est définie pour chaque *Déclaration d'Instance*.

Il est admis trois valeurs pour la *Règle de Nommage* d'une *Règle de Modélisation*: *Facultative*, *Obligatoire* et *Contrainte*.

La sémantique suivante s'applique pendant toute la durée de vie d'une instance fondée sur un *TypeDefinitionNodes* ayant une *Déclaration d'Instance*.

Pour une instance A1 d'un *TypeDefinitionNodes* "Type Alpha" ayant une *Déclaration d'Instance* B1 disposant d'une *Règle de Modélisation* qui utilise la *Règle de Nommage* "*Facultative",* la règle suivante *s'*applique: Pour chaque *Chemin d'Exploration* du Type Alpha à B1, l'instance A1 peut avoir ou ne pas avoir un *Nœud* similaire (voir 6.2.4) pour B1 avec le même *Chemin d'Exploration*. Si ce *Nœud* existe, le service "Traduire Chemins d'Exploration en Identifiants de Nœuds" (TranslateBrowsePathsToNodeIds) (voir la CEI 62541-4) renvoie ce *Nœud* comme première entrée de la liste.

Pour une instance A1 d'un *TypeDefinitionNodes* "Type Alpha" ayant une *Déclaration d'Instance* B1 disposant d'une *Règle de Modélisation* qui utilise la *Règle de Nommage* "*Obligatoire",* la règle suivante s'applique: Pour chaque *Chemin d'Exploration* du Type Alpha à B1, l'instance A1 doit avoir un *Nœud* similaire (voir 6.2.4) pour B1 avec le même *Chemin d'Exploration* si tous les *Nœuds* du *Chemin d'Exploration* existent. Par exemple, si un *Nœud* dans le *Chemin d'Exploration* a une *Règle de Nommage* "*Facultative"* et qu'il est omis dans l'instance, tous les enfants de ce *Nœud* seraient également omis, indépendamment de leurs *Règles de Modélisation*.

Si une *Déclaration d'Instance* a une *Règle de Modélisation* utilisant la *Règle de Nommage* "*Contrainte"*, elle indique que le *Nom d'Exploration* de la *Déclaration d'Instance* est sans importance mais qu'une autre sémantique est définie avec la *Règle de Modélisation*. Le *service* "Traduire Chemins d'Exploration en Identifiants de Nœuds" (TranslateBrowsePathsToNodeIds) (voir CEI 62541-4) ne peut pas en général être utilisé pour accéder à des instances fondées sur ces *Déclarations d'Instances*.

6.4.4.2.2

Sans objet.

6.4.4.3 Règles de sous-typage pour des Propriétés de Règles de Modélisation

Il est admis que des sous-types remplacent des *Règles de Modélisation* en ce qui concerne leurs *Déclarations d'Instances*. 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* "*Obligatoire"*) et, pour le sous-type, rendre la règle facultative (*Règle de Nommage*

"*Facultative*"). Le Tableau 15 précise les modifications autorisées pour les *Propriétés* lorsque les *Règles de Modélisation* sont changées dans le sous-type.

Tableau 15 – Règles applicables aux Propriétés des Règles de Modélisation en cas de sous-typage

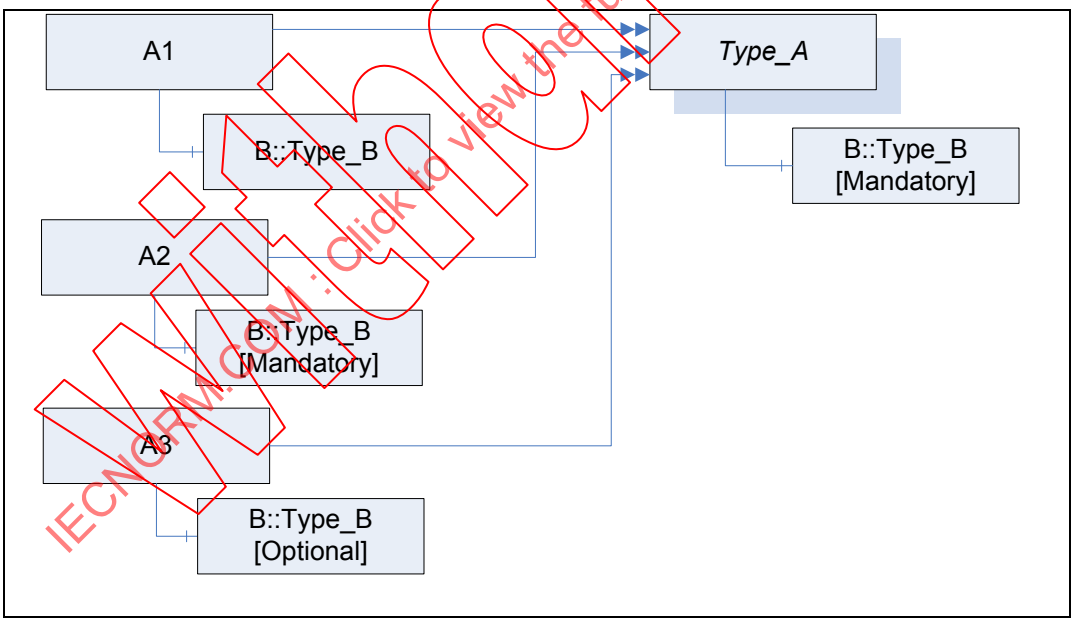
	Valeur pour le supertype	Valeur pour le sous-type
Règle de Nommage	Obligatoire	Obligatoire
Règle de Nommage	Facultative	Obligatoire ou Facultative
Règle de Nommage	Contrainte	Contrainte

6.4.4.4 Règles d'instanciation pour les Propriétés des Règles de Modélisation

Il existe deux différents cas d'utilisation lors de la création d'une instance 'A' fondée sur un *TypeDefinitionNodes* "Type_A". 'A' est utilisé comme une instance normale ou comme *Déclaration d'Instance* d'un autre *Nœud de Définition de Type*.

Dans le premier cas, il n'est pas exigé que des instances nouvellement créées ou référencées, fondées sur des *Déclarations d'Instances*, aient une *Règle de Modélisation*; cependant, il n'est pas permis qu'elles aient une *Règle de Modélisation* indépendante de la *Règle de Modélisation* de leur *Déclaration d'Instance*.

Un exemple est fourni en Figure 16. Les instances A1, A2 et A3 sont des instances valides de Type_A, même si B de A1 n'a aucune *Règle de Modélisation* et B de A3 a une *Règle de Modélisation* différente de B de Type_A.



IEC 1774/10

Légende

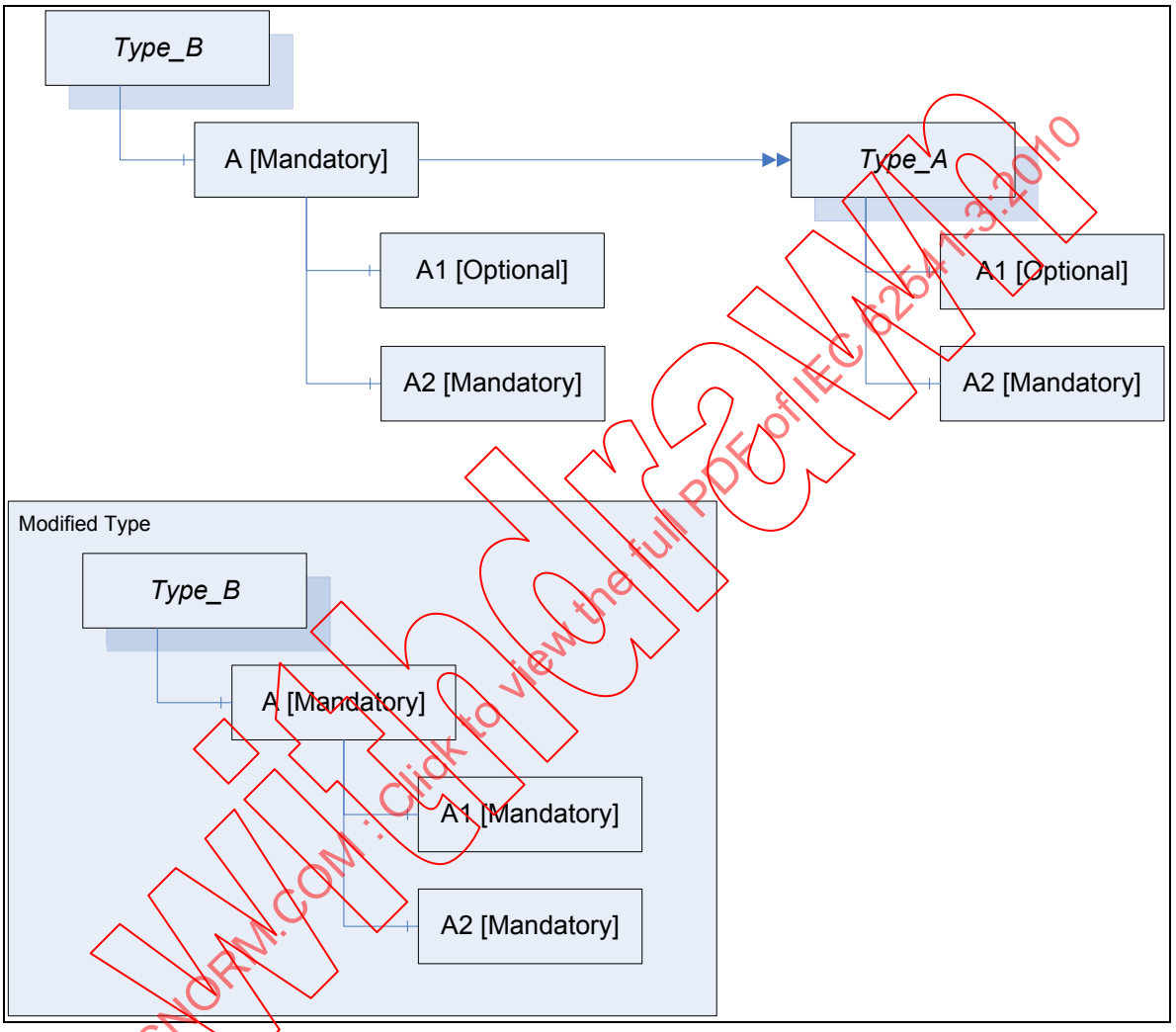
Anglais	Français
Mandatory	Obligatoire
optional	Facultatif

Figure 16 – Exemple de modification d'instances sur la base des Déclarations d'Instances

Dans le second cas, toutes les instances qui sont directement ou indirectement référencées à partir de 'A', fondées sur des *Déclarations d'Instances* de 'Type_A', maintiennent initialement la même *Règle de Modélisation* que leurs *Déclarations d'Instances*. Les *Règles de*

Modélisation peuvent être mises à jour; les modifications autorisées aux *Règles de Modélisation* de ces *Nœuds* sont les mêmes que celles définies pour le sous-typage en 6.4.4.3.

La Figure 17 donne un exemple de ce scénario. Le Type_B utilise une *Déclaration d'Instance* fondée sur le Type_A (partie supérieure de la Figure). Par la suite, les *Règles de Modélisation* de la *Déclaration d'Instance* A1 sont modifiées (partie inférieure de la Figure). La *Règle de Nommage* de A1 est devenue *obligatoire* (modifiée à partir de "*Facultative*").



Légende

Anglais	Français
Mandatory	Obligatoire
optional	Facultative
Modified type	Type modifié

Figure 17 – Exemple de modification de Déclarations d'Instances fondées sur une Déclaration d'Instance

6.4.4.5 Règles de Modélisation normalisées

6.4.4.5.1 Intitulés des Règles de Modélisation normalisées

Les paragraphes suivants définissent les *Règles de Modélisation*. Le Tableau 16 résume les *Propriétés* de ces *Règles de Modélisation*.

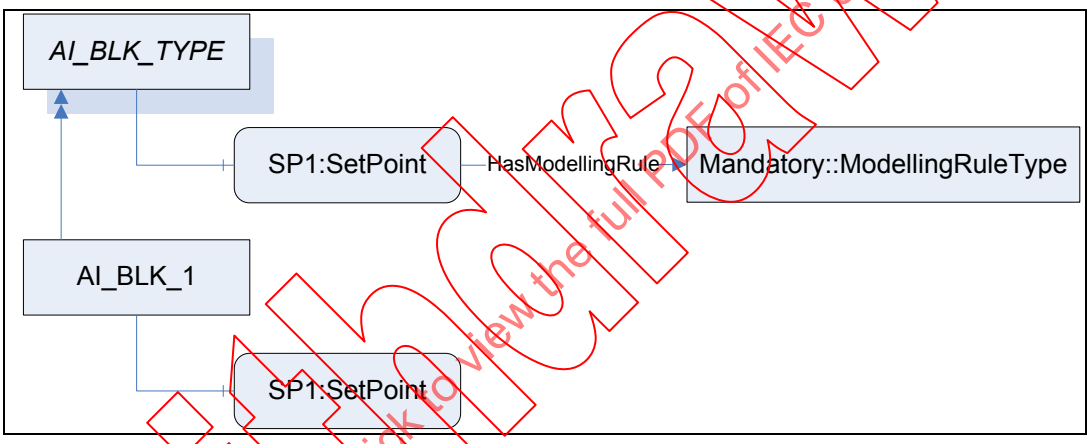
Tableau 16 – Propriétés des Règles de Modélisation

Intitulé	Règle de Nommage
Obligatoire	Obligatoire
Facultative	Facultative
Expose Sa Matrice	Contrainte

6.4.4.5.2 Obligatoire

Une *Déclaration d'Instance* marquée d'une *Règle de Modélisation* "*Obligatoire"* satisfait précisément à la sémantique définie pour la *Règle de Nommage* "*Obligatoire"*. Cela signifie que pour chaque *Chemin d'Exploration* existant sur l'instance, il doit exister un *Nœud* similaire, mais il n'est pas défini si un nouveau *Nœud* est créé ou si un *Nœud* existant est référencé.

Par exemple, le *TypeDefinitionNodes* d'un bloc fonctionnel “AI_BLK_TYPE” aura un point de consigne “PC1”. Une instance de ce type “AI_BLK_1” aura un point de consigne nouvellement créé “PC1" comme *Nœud* similaire à la *Déclaration d'Instance* PC1. Cet exemple est illustré en Figure 18.



Légende

Anglais	Français
SP1 setpoint	Point de consigne PC1
Hasmodellingrule	Dispose de Règle de Modélisation
Mandatory: modellingruletype	Obligatoire: Type de Règle de Modélisation

Figure 18 – Utilisation de la Règle de Modélisation normalisée "Nouveau"

Un exemple complexe, provenant des *Règles de Modélisation* "*Obligatoire"* et *"Facultative",* est donné en 6.4.4.5.3.

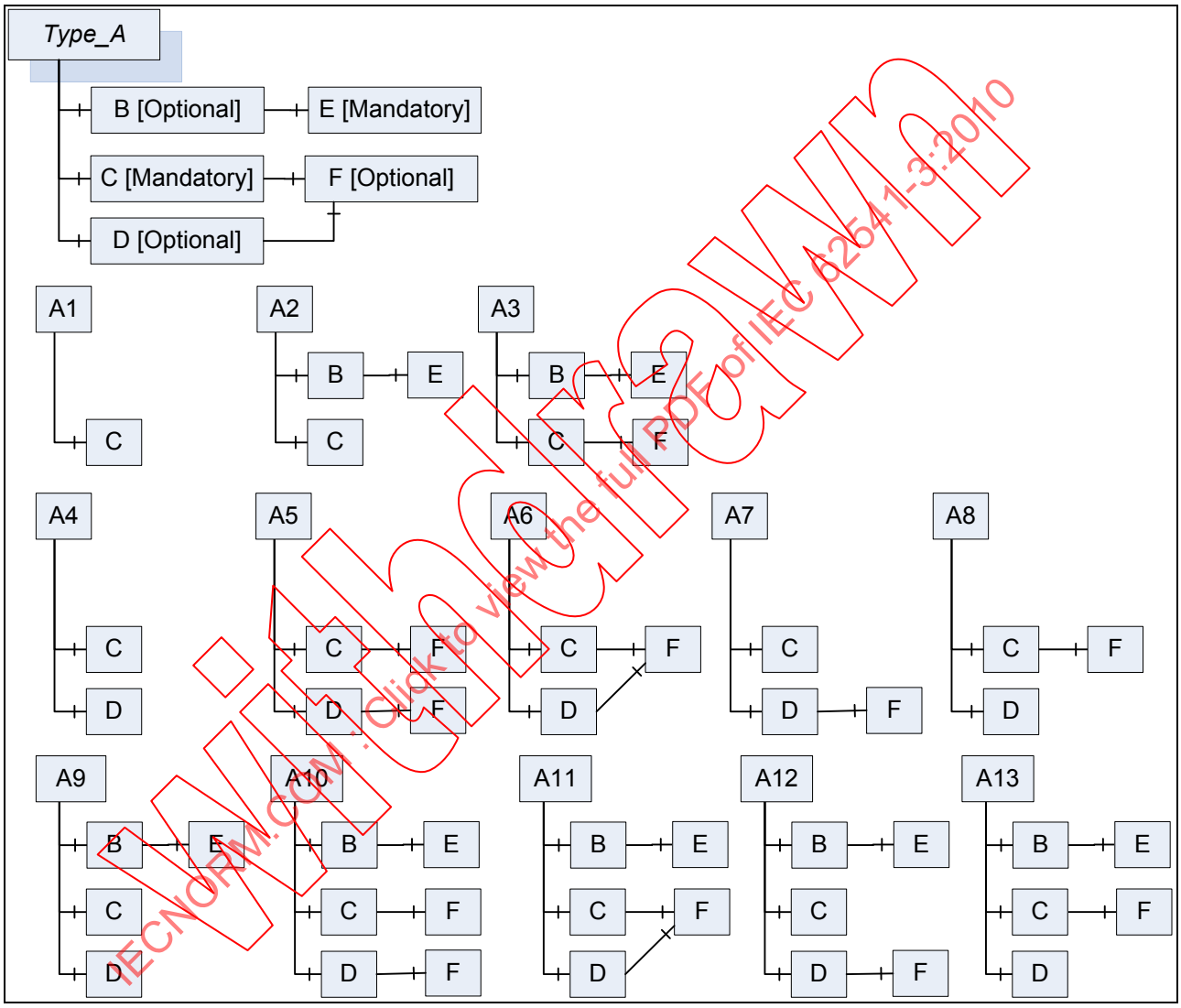
6.4.4.5.3 Facultative

Une *Déclaration d'Instance* marquée d'une *Règle de Modélisation* "*Facultative"* satisfait précisément à la sémantique définie pour la *Règle de Nommage* "*Facultative"*. Cela signifie que pour chaque *Chemin d'Exploration* 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 *Règles de Modélisation* "*Facultative"* et "*Obligatoire"* est illustré en Figure 19. L'exemple contient un *Type d'Objet* "Type_A" et toutes les combinaisons valides d'instances nommées A1 à A13. A noter que si le B "facultatif" est fourni, le "E" obligatoire doit être fourni également; dans le cas contraire il ne l'est pas. F est référencé par C et D.

Dans cette instance, il peut s'agir du même *Nœud* ou de deux *Nœuds* différents ayant le même *Nom d'Exploration* (*Nœud* similaire à la *Déclaration d'Instance* F). L'exemple ne tient pas compte du fait que les instances ont ou n'ont pas des *Règles de Modélisation.* On suppose que chaque F est similaire à la *Déclaration d'Instance* F, etc.

Dans le cas où il y aurait une *Référence* non hiérarchique entre E et F dans la *Hiérarchie de Déclaration d'Instance*, il n'est pas spécifié si elle a lieu ou non dans la hiérarchie d'instance. Dans le cas de A13, il pourrait y avoir une référence à partir de F et non pas à partir de l'autre, ou à partir des deux ou à partir d'aucun d'entre eux.



Légende

Anglais	Français
Mandatory	Obligatoire
optional	Facultative

Figure 19 – Exemple d'utilisation des Règles de Modélisation normalisées "Facultative" et "Obligatoire"

6.4.4.5.4 Expose Sa Matrice (ExposeItsArray)

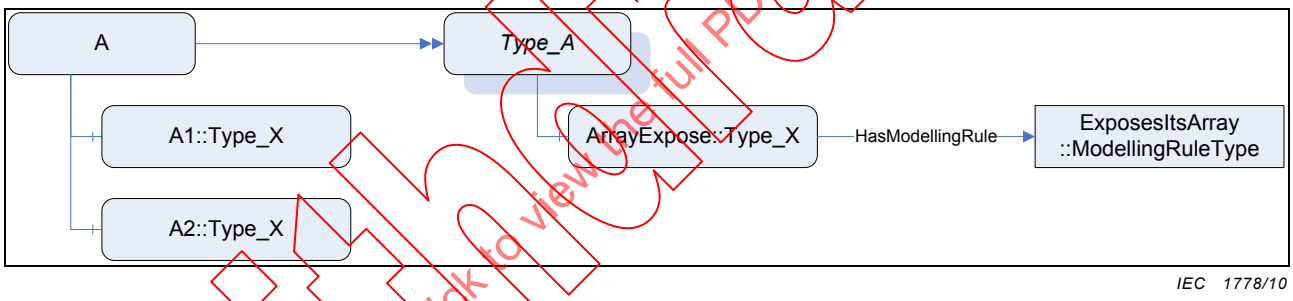
La *Règle de Modélisation ExposesItsArray (Expose Sa Matrice)* présente une sémantique particulière pour les *Types de Variables* ayant comme type de données une matrice

unidimensionnelle ou multidimensionnelle. Elle indique que chaque valeur de la matrice sera également présentée comme une *Variable* dans *l'Espace d'Adresse*.

La *Règle de Modélisation "Expose Sa Matrice"* ne peut être appliquée qu'aux *Déclarations d'Instances* de *Classe de Nœud* "*Variable"* qui font partie d'un *Type de Variable* ayant comme type de données une matrice unidimensionnelle ou multidimensionnelle.

La *Variable* A ayant cette *Règle de Modélisation* doit être référencée par une *Référence hiérarchique* dans le sens direct à partir d'un *Type de Variable* B. B doit avoir une valeur "*Rang de Valeur"* 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 correspondant. S'il y a plusieurs *Références hiérarchiques* dans le sens direct entre A et B, toutes les instances fondées sur B doivent être référencées avec ces *Références*.

Un exemple est donné en Figure 20. 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 la *Déclaration d'Instance* ArrayExpose (Expose Matrice). Les *Noms d'Exploration* de ces instances ne sont pas définis par la *Règle de Modélisation*. En général, il n'est pas possible d'obtenir une *Variable* représentant une entrée spécifique de la matrice (par exemple la seconde). Les clients tenteront d'obtenir la matrice ou d'accéder directement aux *Variables*, de sorte qu'il n'est pas nécessaire que cette information soit fournie.



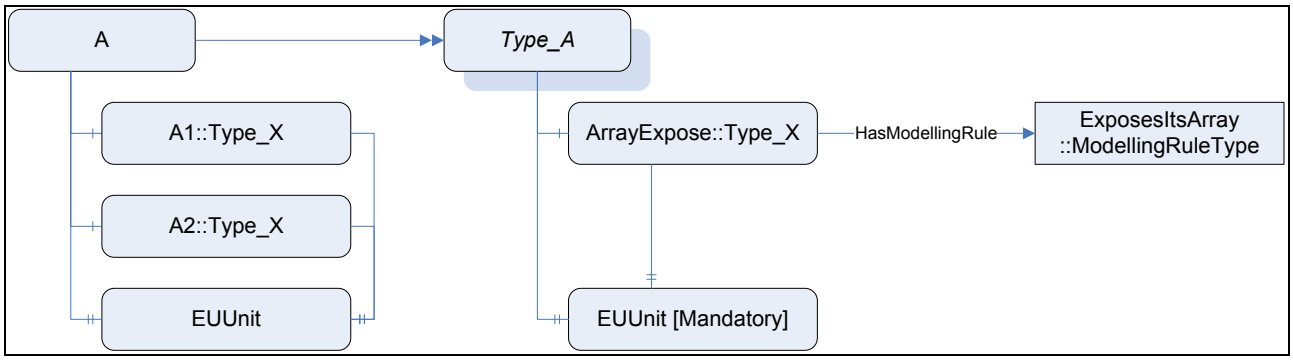
Légende

Anglais	Français
Arrayexpose	Expose Matrice
Hasmodelingtype	Dispose d'un Type de Modélisation
Exposesitsarray: modelingruletype	Expose Sa Matrice: Type de Règle de Modélisation

Figure 20 – Exemple d'utilisation de la Règle de Modélisation "Expose Sa Matrice"

Il est également admis de référencer A par d'autres *Déclarations d'Instances*. Ces *Références* doivent être reflétées en chaque instance fondée sur A.

Un exemple est donné en Figure 21. La *Propriété* EUUnit (Unité EU) est référencée par ArrayExpose (Expose Matrice) et par conséquent chaque instance fondée sur "Expose Matrice" référence l'instance fondée sur la *Déclaration d'Instance* "Unité EU".



IEC 1779/10

Légende

Anglais	Français
Arrayexpose	Expose Matrice
Exposesitsarray: modelingruletype	Expose Sa Matrice: Type de Règle de Modélisation
Hasmodelingrule	Dispose d'une Règle de Modélisation
EUUnit	Unité EU
EUUnit (mandatory)	Unité EU (obligatoire)

Figure 21 – Exemple complexe d'utilisation de la Règle de Modélisation "Expose Sa Matrice"

6.5 Modifications de définitions de type déjà utilisées

Aucun comportement n'est spécifié concernant les sous-types et les instances lorsque les *Types d'Objets* et les *Types de Variables* 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 satisfaites. Par exemple, il n'est pas admis qu'une *Propriété* soit ajoutée en utilisant la *Règle de Modélisation* "*Obligatoire"* sur un type donné s'il existe des instances de ce type sans la *Propriété*. Dans ce cas, le serveur doit soit ajouter la *Propriété* à toutes les instances de type, soit rejeter l'ajout de la *Propriété* ajoutée.

6.6 Modèle Parent

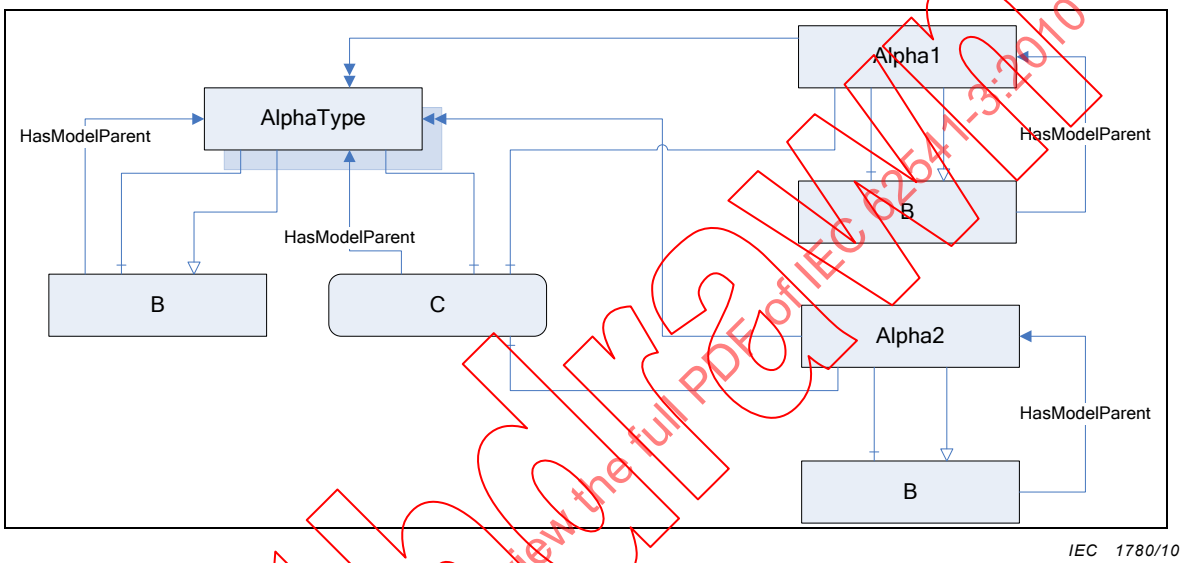
Chaque *Objet*, *Variable* et *Méthode* peut être référencé(e) par plusieurs *Références hiérarchiques*. Pour indiquer l'étendue d'un tel *Nœud* 'A', il peut exposer son *Modèle Parent*. Le *Modèle Parent* de 'A' est l'étendue dans laquelle 'A' est défini. Lorsqu'un client a l'intention de modifier 'A', il peut utiliser le *Modèle Parent* pour déterminer s'il dispose de l'étendue correcte.

Par exemple, un *TypeDefinitionNodes* 'Type_A' peut avoir une *Déclaration d'Instance* appelée 'Icône' qui est partagée par toutes les instances. Ainsi le *Modèle Parent* de 'Icône' est 'Type_A'. Un client peut explorer une instance 'A' qui est de 'Type_A' pour modifier la valeur de 'Icône' pour 'A'. 'A' référence la *Déclaration d'Instance* partagée 'Icône' et, en identifiant son *Modèle Parent*, un client peut conclure que l'étendue du *Nœud* n'était pas 'A' mais 'Type_A'. Ainsi, le client ne peut pas modifier la valeur du *Nœud* mais plutôt générer une copie, référencer la copie au lieu de la *Déclaration d'Instance* et modifier la valeur de la copie.

Pour identifier un *Modèle Parent*, le *Type de Référence HasModelParent* (*Dispose d'un Modèle Parent)* est utilisé en pointant depuis l'*Objet*, la *Variable* ou la *Méthode* vers le *Nœud* qui représente le *Modèle Parent*. Le *Modèle Parent* doit référencer l'*Objet*, la *Variable* ou la *Méthode* au moyen d'une *Référence hiérarchique* dans le sens direct.

Les Références "*Dispose d'un Modèle Parent"* doivent être fournies pour toutes les *Règles de Modélisation* définies dans le présent document. Il n'est pas exigé mais recommandé que les *Références* "*Dispose d'un Modèle Parent"* soient exposées pour d'autres *Règles de Modélisation*. Il est admis de fournir une *Référence "Dispose d'un Modèle Parent",* pour des *Objets*, *Variables*, et *Méthodes* n'ayant aucune référence "*Dispose d'une Règle de Modélisation"*. Cependant, cela n'est pas exigé car il est parfois possible qu'il n'y ait pas de *Modèle Parent* clairement défini.

La Figure 22 illustre les relations "*Dispose d'un Modèle Parent" pour un TypeDefinitionNodes* et deux instances. Dans cet exemple, le modèle parent du *Nœud* "C" est le *TypeDefinitionNodes* et toutes les instances le référencent au moyen de références hiérarchiques.



Légende

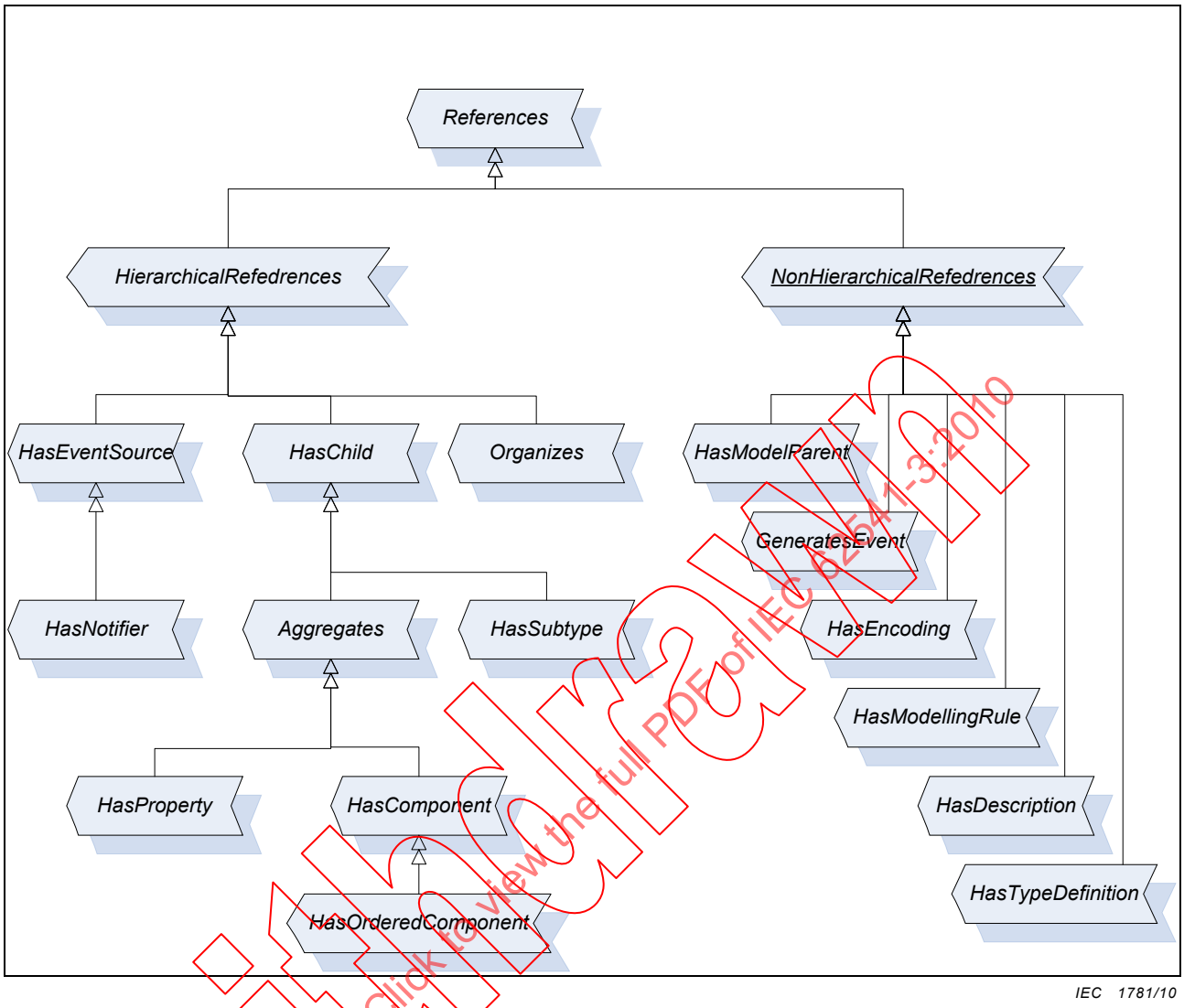
Anglais	Français
Hasmodeparent	Dispose d'un Modèle Parent
Alphatype	Type Alpha

Figure 22 – Exemple de Modèles Parents

7 Types de Références normalisées

7.1 Généralités

Cette norme définit les *Types de Références* comme partie inhérente du Modèle de l'Espace d'Adresse d'OPC UA. La Figure 23 fournit une description informelle de la hiérarchie de ces *Types de Références*. D'autres parties de la présente série de normes peuvent spécifier des *Types de Références* supplémentaires. Les paragraphes suivants définissent les *Types de Références*. La CEI 62541-5 définit leur représentation dans *l'Espace d'Adresse*.



Légende

Anglais	Français
References	Références
Hierarchicalreferences	Références Hiérarchiques
Nonhierarchicalreferences	Références Non Hiérarchiques
Haseventsource	Dispose d'une Source d'Evénements
Haschild	Dispose d'une Référence Enfant
Organizes	Organise
Hasmodelparent	Dispose d'un Modèle Parent
Generatesevent	Génère un Evénement
Hasnotifier	Dispose d'un Notificateur
Aggregates	Regroupe
HasSubtype	Dispose d'un Sous-Type
Hasencoding	Dispose d'un Codage
Hasmodelingrule	Dispose d'une Règle de Modélisation
Hasproperty	Dispose d'une Propriété
Hascomponent	Dispose d'un Composant
Hasdescription	Dispose d'une Description
Hastypedefinition	Dispose d'une Définition de Type
hasorderedcomponent	Dispose de Composants Ordonnés

Figure 23 – Hiérarchie normalisée d'un Type de Référence

7.2 Type de Référence "Références"

Le *Type de Référence "Références"* est un *Type de Référence* abstrait; Seuls ses sous-types peuvent être utilisés.

Aucune sémantique n'est associée à ce *Type de Référence*. Il s'agit du type de base de tous les *Types de Références*. Tous les *Types de Références* doivent être un sous-type de ce *Type de Référence* de base – directement ou indirectement. Le principal objectif de ce *Type de Référence* est de permettre l'utilisation de filtres et de requêtes simples dans les *Services* correspondants de la CEI 62541-5.

Il n'y a pas de contraintes définies pour ce *Type de Référence* abstrait.

7.3 Type de Référence "HierarchicalReferences "

Le *Type de Référence « Références Hiérarchiques »* est un *Type de Référence* abstrait; seuls ses sous-types peuvent être utilisés.

La sémantique des *Références Hiérarchiques* a pour but d'indiquer que les *Références* des *Références Hiérarchiques* 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 *Références Hiérarchiques* n'interdisent pas les boucles. Par exemple, en partant du *Nœud* “A” et en suivant les *Références Hiérarchiques,* on peut être amené à 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 *Type de Référence* abstrait.

Il n'est pas admis que le *SourceNode* et le *TargetNode* d'une *Référence* de *Type de Référence Références Hiérarchiques* soient les mêmes; en d'autres termes, il n'est pas admis d'avoir des auto-références utilisant des *Références Hiérarchiques*.

7.4 Type de Référence "NonHierarchicalReferences "

Le *Type de Référence "Références Non Hiérarchiques"* est un *Type de Référence* abstrait; seuls ses sous-types peuvent être utilisés.

La sémantique des *Références Non Hiérarchiques* a pour but 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 *Types de Références* concrets doivent hériter soit des *Références hiérarchiques*, soit des *Références non hiérarchiques*, directement ou indirectement.

Il n'y a pas de contraintes définies pour ce *Type de Référence* abstrait.

7.5 Type de Référence "HasChild"

Le *Type de Référence HasChild* (*Dispose d'une Référence Enfant)* est un Type de Référence abstrait*;* seuls ses sous-types peuvent être utilisés. C'est un sous-type des *Références Hiérarchiques*.

La sémantique a pour but 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 *Type de Référence "Dispose d'une Référence Enfant",* on ne doit jamais pouvoir revenir à “A”. Il est cependant admis qu'en suivant les *Références,* il peut y avoir plusieurs chemins menant vers un autre *Nœud* “B”.

7.6 Type de Référence "Aggregates"

Le *Type de Référence Aggregates (Regroupe)* est un *Type de Référence* abstrait; seuls ses sous-types peuvent être utilisés. Il s'agit d'un sous-type du type "*Dispose d'une Référence Enfant"*.

La sémantique a pour but d'indiquer qu'une partie (le *Nœud Cible*) appartient au *Nœud Source*. Elle ne spécifie pas le propriétaire du *Nœud Cible*.

Il n'y a pas de contraintes définies pour ce *Type de Référence* abstrait.

7.7 Type de Référence "HasComponent"

Le *Type de Référence HasComponent (Dispose d'un Composant)* est un *Type de Référence* concret qui peut être utilisé directement. Il s'agit d'un sous-type du *Type de Référence Regroupe*.

La sémantique indique une relation "fait partie de". Le *TargetNode* d'une *Référence* de *Type de Référence "Dispose d'un Composant"* fait partie du *Nœud Source*. Ce *Type de Référence* est utilisé pour relier des *Objets* ou des *Types d'Objets* avec des *Objets*, *Variables de Données* et *Méthodes* qu'ils contiennent, ainsi que des *Variables* complexes ou des *Types de Variables* avec leurs *Variables de Données*.

Comme tous les autres *Types de Références*, ce *Type de Référence* ne spécifie rien concernant le propriétaire des parties, même s'il représente une sémantique relationnelle "fait partie de". En d'autres termes, il n'est pas spécifié si le *TargetNode* d'une *Référence* du *Type de Référence "Dispose d'un Composant"* est supprimé lorsque le *SourceNode* est supprimé.

Le *TargetNode* de ce *Type de Référence* doit être une *Variable*, un *Objet* ou une *Méthode*.

Si le *TargetNode* est une *Variable*, le *SourceNode* doit être un *Objet*, un *Type d'Objet*, une *DataVariable* ou un *Type de Variable*. En utilisant la *Référence "Dispose d'un Composant"*, la *Variable* est définie comme *Variable de Données*.

Si le *TargetNode* est un *Objet* ou une *Méthode*, le *SourceNode* doit être un *Objet* ou un *Type d'Objet*.

7.8 Type de Référence "HasProperty"

Le *Type de Référence HasProperty (Dispose d'une Propriété)* est un *Type de Référence* concret qui peut être utilisé directement. Il s'agit d'un sous-type du *Type de Référence Regroupe*.

La sémantique a pour but d'identifier les *Propriétés* d'un *Nœud*. Les *Propriétés* sont décrites en 4.4.2.

Le *SourceNode* de ce *Type de Référence* peut être toute *Classe de Nœud.* Le *TargetNode* doit être *Variable*. En utilisant la *Référence "Dispose d'une Propriété"*, la *Variable* est définie comme *Propriété*. Etant donné que les *Propriétés* ne doivent pas avoir de *Propriétés*, une *Propriété* ne doit jamais être le *SourceNode* d'une *Référence "Dispose d'une Propriété"*.

7.9 Type de Référence "HasOrderedComponent "

Le *Type de Référence HasOrderedComponent (Dispose de Composants Ordonnés)* est un *Type de Référence* concret qui peut être utilisé directement. Il s'agit d'un sous-type du *Type de Référence Dispose d'un Composant*.

La sémantique du *Type de Référence "Dispose de Composants Ordonnés"* – outre celle *Type de Référence "Dispose d'un Composant"* – est d'indiquer lorsque l'exploration est effectuée à partir d'un *Nœud* et suivant des *Références* de ce type ou de son sous-type, toutes les *Références* sont renvoyées, dans un ordre bien défini, au *Service* d'exploration défini dans la CEI 62541-5. L'ordre est fonction du serveur, mais le client peut considérer que le serveur renvoie toujours les composants dans le même ordre.

Il n'y a pas d'autres contraintes définies pour ce *Type de Référence*.

7.10 Type de Référence "HasSubtype"

Le *Type de Référence HasSubtype* (*Dispose d'un Sous-type)* est un *Type de Référence* concret qui peut être utilisé directement. Il s'agit d'un sous-type du *Type de Référence* "*Dispose d'une Référence Enfant"*.

La sémantique de ce *Type de Référence* vise à exprimer une relation entre des sous-types et des types. Elle est utilisée pour couvrir la hiérarchie du *Type de Référence*, dont la sémantique est spécifiée en 5.3.3.3; la hiérarchie de *Type de Données* est spécifiée en 5.8.3, et d'autres hiérarchies du sous-type sont spécifiées à l'Article 6.

Le *SourceNode* des *Références* de ce type doit être un *Type d'Objet*, un *Type de Variable*, un *Type de Données* ou un *Type de Référence* et le *TargetNode* doit être de la même *Classe de Nœud* que le *Nœud Source*. Chaque *Type de Référence* doit être le *TargetNode* d'au plus une *Référence* du type *«HasSubtype»*.

7.11 Type de référence "Organize"

Le Type de référence *Organizes (Organise)* est un *Type de Référence* concret et peut être directement utilisé. C'est un sous-type des *Références Hiérarchiques*.

La sémantique de ce *Type de Référence* vise à organiser les *Nœuds* dans *l'Espace d'Adresse*. Elle peut être utilisée pour couvrir de multiples hiérarchies indépendamment de toute hiérarchie créée avec les *Références Regroupe* sans boucles.

Le *SourceNode* des *Références* de ce type doit être un *Objet* ou une *Vue.* S'il s'agit d'un *Objet*, il est demandé qu'il soit du *Type d'Objet "Type de Dossier"* ou de l'un de ses sous-types (voir 5.5.3).

Le *TargetNode* de ce *Type de Référence* peut être toute *Classe de Nœud.*

7.12 Type de Référence "HasModellingRule"

Le *Type de Référence HasModellingRule* (*Dispose d'une Règle de Modélisation)* est un *Type de Référence* concret et peut être directement utilisé. C'est un sous-type des *Références Non Hiérarchiques*.

La sémantique de ce *Type de Référence* vise à lier la *Règle de Modélisation* à un *Objet*, une *Variable* ou une *Méthode*. Les mécanismes des *Règles de Modélisation* sont décrits en 6.4.4.

Le *SourceNode* de ce *Type de Référence* doit être un *Objet*, une *Variable* ou une *Méthode*. Le *TargetNode* doit être un *Objet* de *Type d'Objet* “Règle de Modélisation” ou l'un de ses sous-types.

Chaque *Nœud* doit être un *SourceNode* d'au plus une *Référence "Dispose d'une Règle de Modélisation"*.

7.13 Type de Référence "HasModelParent"

Le *Type de Référence HasModelParent (Dispose d'un Modèle Parent)* est un *Type de Référence* concret et peut être directement utilisé. C'est un sous-type des *Références Non Hiérarchiques*.

La sémantique de ce *Type de Référence* vise à exposer le *Modèle Parent* d'un *Objet*, d'une *Variable* ou d'une *Méthode*. Les mécanismes du *Modèle Parent* sont décrits en 6.6.

Le *SourceNode* de ce *Type de Référence* doit être un *Objet*, une *Variable* ou une *Méthode*.

Chaque *Nœud* doit être un *SourceNode* d'au plus une *Référence "Dispose d'un Modèle Parent"*.

7.14 Type de Référence "HasTypeDefinition"

Le *Type de Référence HasTypeDefinition (Dispose d'une Définition de Type)* est un *Type de Référence* concret et peut être directement utilisé. C'est un sous-type des *Références Non Hiérarchiques*.

La sémantique de ce *Type de Référence* vise à lier un *Objet* ou une *Variable*, respectivement, à son *Type d'Objet* ou à son *Type de Variable*. Les relations entre types et instances sont décrites en 4.5.

Le *SourceNode* de ce *Type de Référence* doit être un *Objet* ou une *Variable*. Si le *SourceNode* est un *Objet*, le *TargetNode* doit être un *Type d'Objet*. Si le *SourceNode* est une *Variable*, le *TargetNode* doit être un *Type de Variable*.

Chaque *Variable* et chaque *Objet* doit être le *SourceNode* de précisément une *Référence "Dispose d'une Définition de Type"*.

7.15 Type de Référence "HasEncoding"

Le *Type de Référence HasEncoding* (*Dispose d'un codage*) est un *Type de Référence* concret et peut être directement utilisé. C'est un sous-type des *Références Non Hiérarchiques*.

La sémantique de ce *Type de Référence* vise à référencer les *Codages de Type de Données* d'un *Type de Données*.

Le *SourceNode* des *Références* de ce type doit être un *Type de Données*.

Le *TargetNode* de ce *Type de Référence* doit être un *Objet* du *Type d'Objet Type de Codage de Type de Données* ou de l'un de ses sous-types (voir 5.8.4).

7.16 Type de Référence "HasDescription"

Le *Type de Référence HasDescription (Dispose d'une Description)* est un *Type de Référence* concret et peut être directement utilisé. C'est un sous-type des *Références Non Hiérarchiques*.

La sémantique de ce *Type de Référence* vise à référencer la *Description de Type de Données* d'un *Codage de Type de Données*.

Le *SourceNode* des *Références* de ce type doit être un *Objet* du *Type d'Objet Type de Codage de Type de Données* ou de l'un de ses sous-types.

Le *TargetNode* de ce *Type de Référence* doit être une *Variable* du *Type de Variable "Type de Description de Type de Données"* ou de l'un de ses sous-types (voir 5.8.4).

7.17 Type de reference "GeneratesEvent"

Le *Type de Référence GeneratesEvent* (*Génère un Evénement*) est un *Type de Référence* concret et peut être directement utilisé. C'est un sous-type des *Références Non Hiérarchiques*.

La sémantique de ce *Type de Référence* vise à identifier les types d'*Evénements* que des instances de *Types d'Objets* ou de *Types de Variables* peuvent générer ainsi que celles que les *Méthodes* peuvent générer à chaque invocation de *Méthode*.

Le *SourceNode* des *Références* de ce type doit être un *Type d'Objet*, un *Type de Variable* ou une *Méthode*.

Le *TargetNode* de ce *Type de Référence* doit être un *Type d'Objet* représentant les *Types d'Evénements*, c'est-à-dire un *Type d'Evénement de Base* ou l'un de ses sous-types.

7.18 Type de Référence "AlwaysGeneratesEvent"

Le *Type de Référence AlwaysGeneratesEvent* (*Génère Toujours un Evénement*) est un *Type de Référence* concret et peut être directement utilisé. C'est un sous-type du *Type de Référence* "*Génère un Evénement*".

La sémantique de ce *Type de Référence* vise à identifier les types d'*Evénements* que les *Méthodes* doivent générer à chaque invocation de *Méthode*.

Le *SourceNode* des *Références* de ce type doit être une *Méthode*.

Le *TargetNode* de ce *Type de Référence* doit être un *Type d'Objet* représentant les *Types d'Evénements*, c'est-à-dire un *Type d'Evénement de Base* ou l'un de ses sous-types.

7.19 Type de Référence "HasEventSource"

Le *Type de Référence HasEventSource* (*Dispose d'une Source d'Evénement)* est un *Type de Référence* concret et peut être directement utilisé. C'est un sous-type des *Références Hiérarchiques*.

La sémantique de ce *Type de Référence* vise à relier des sources d'événements dans une organisation hiérarchique sans boucles. Ce *Type de Référence* et ses éventuels sous-types sont destinés à être utilisés pour découvrir la génération d'*Evénements* dans un serveur. Il n'est pas exigé qu'ils soient présents pour qu'un serveur génère des *Evénements* à partir de sa source vers ses *Nœuds* notificateurs. En particulier, le notificateur racine d'un serveur, l'*Objet "Serveur"* défini dans la CEI 62541-5, est toujours capable de fournir l'ensemble des *Evénements* à partir du serveur et, en tant que tel, il a des *Références* implicites "*Dispose d'une Source d'Evénement"* vers chaque source d'événement dans un serveur donné.

Le *SourceNode* de ce *Type de Référence* doit être un *Objet* qui est une source d'abonnements à des événements. Une source d'abonnements à des événements est un *Objet* dont le bit SubscribeToEvents (“S'Abonner à des Evénements”) est établi dans l'*Attribut "Notificateur d'Evénements"*.

Le *TargetNode* de ce *Type de Référence* peut être un *Nœud* de toute *Classe de Nœud* qui peut générer des notifications d'événements par le biais d'un abonnement à la source de référence.

En partant du *Nœud* “A” et en suivant uniquement les *Références* du *Type de Référence* "*Dispose d'une Source d'Evénement"* ou de ses sous-types, on ne doit jamais pouvoir revenir à “A”. Il est cependant admis qu'en suivant les *Références,* il peut y avoir plusieurs chemins menant vers un autre *Nœud* “B”.

7.20 Type de Référence "HasNotifier"

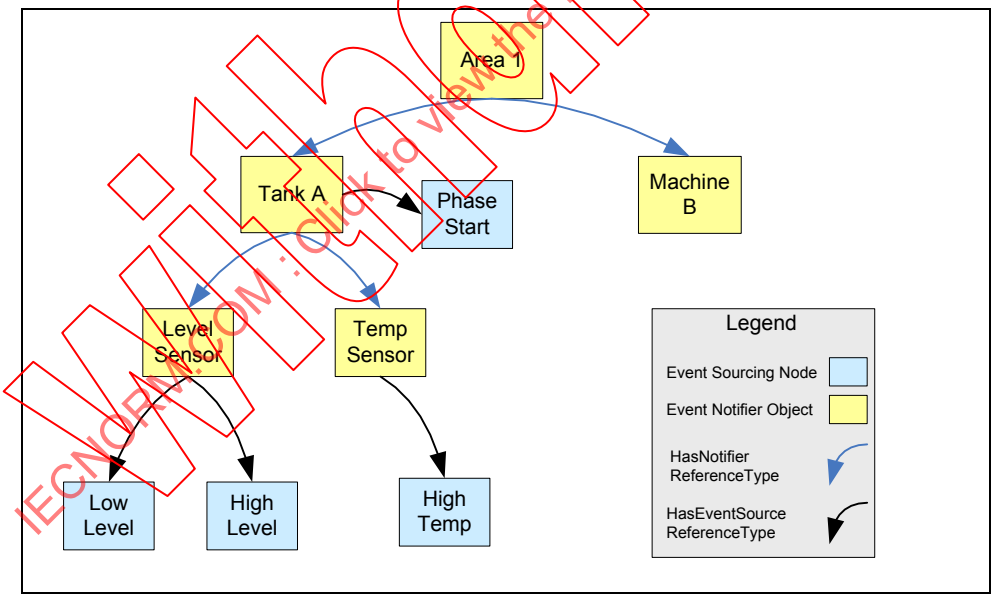
Le Type de Référence *HasNotifier (Dispose d'un Notificateur)* est un *Type de Référence* concret qui peut être directement utilisé. C'est un sous-type du *Type de Référence "Dispose d'une Source d'Evénement"*.

La sémantique de ce *Type de Référence* a pour objectif de relier des *Nœuds Objet* notificateurs entre eux. Le *Type de Référence* est utilisé pour établir une organisation hiérarchique des *Objets* de notification d'événements. Il s'agit d'un sous-type du *Type de Référence "Dispose d'une Source d'Evénement"* défini en 7.18.

Le *SourceNode* de ce *Type de Référence* doit être un *Objet* ou une *Vue* qui est une source d'abonnement à des événements. Le *TargetNode* de ce *Type de Référence* doit être un *Objet* qui est une source d'abonnement à des événements. Une source d'abonnement à des événements est un *Objet* dont le bit « S'Abonner à des Evénements » est à Un établi dans l'*Attribut "Notificateur d'Evénements"*.

Si le *TargetNode* d'une *Référence* de ce type génère un *Evénement*, cet *Evénement* doit également être fourni dans le *SourceNode* de la *Référence*.

La Figure 24 présente un exemple d'organisation possible de *Références d'Evénements*. Dans cet exemple, un abonnement à des *événements* non filtré, dirigé vers l'*Objet* « Capteur de Niveau » fournira les sources d'Evénements « Niveau Bas » et « Niveau Haut » à l'abonné. Un abonnement à des *événements* non filtré, dirigé vers l'*Objet* « Zone 1 », fournira les sources d'Evénements à partir de la « Machine B », du « Réservoir A » et de toutes les sources notificatrices sous le "Réservoir A".



IEC 1782/10

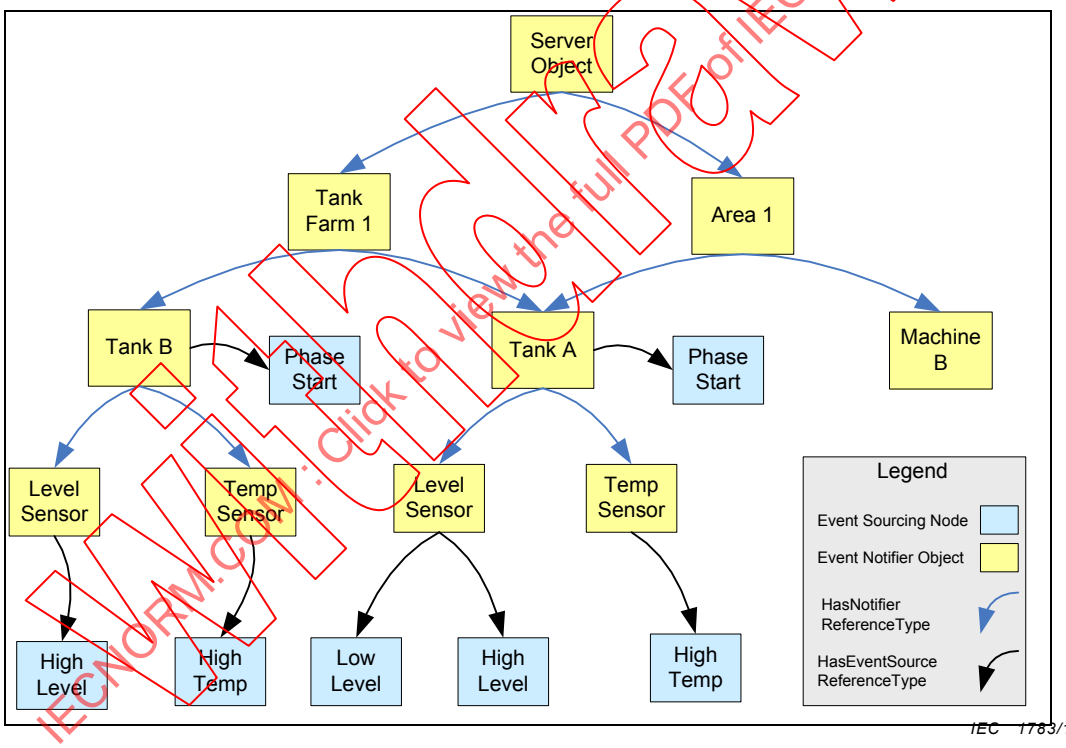
Légende

Anglais	Français
area	Zone
tank	Réservoir
phase start	Début de phase
level sensor	Capteur de niveau
temp sensor	Capteur de temp.
low level	Niveau bas
high level	Niveau haut

Anglais	Français
high temp	Température élevée
legend	Légende
event sourcing node	SourceNode d'Evénement
event notifier object	Objet Notificateur d'Evénement
hasnotifier referencetype	Type de Référence "Dispose d'un notificateur"
haseventsource referencetype	Type de Référence "Dispose d'une Source d'Evénement"

Figure 24 – Exemple de Référence d'Evénement

La Figure 25 présente un autre exemple d'une organisation plus complexe de *Références d'Evénements*. Dans cet exemple, les *Références* explicites sont incluses à partir de l'*Objet* "*Serveur"* du serveur, qui est une source de tous les *Evénements* du serveur. Il a été introduit une seconde organisation des *Evénements* permettant de recueillir les *Evénements* relatifs au « Parc de Stockage 1 ». Un abonnement à des *Evénements* non filtré, dirigé vers l'*Objet* « Parc de Stockage 1 », fournira les sources d'*Evénements* à partir du « Réservoir B », du « Réservoir A » et de toutes les sources notificatives sous le "Réservoir B" et le "Réservoir A".



Légende

Anglais	Français
Server object	Objet Serveur
tank farm	Parc de stockage
tank	Réservoir
area	Zone
phase start	Début de phase
level sensor	Capteur de niveau
temp sensor	Capteur de température.
low level	Niveau bas
high level	Niveau haut

Anglais	Français
high temp	Température élevée
legend	Légende
event sourcing node	SourceNode d'Evénement
event notifier object	Objet Notificateur d'Evénement
hasnotifier referencetype	Type de Référence "Dispose d'un notificateur"
haseventsource referencetype	Type de Référence "Dispose d'une Source d'Evénement"

Figure 25 – Exemple de Référence à des Evénements complexes

8 Types de Données Normalisés

8.1 Généralités

Les paragraphes suivants définissent les *Types de Données*. Leur représentation dans l'*Espace d'Adresse* et la hiérarchie du *Type de Données* sont spécifiées dans la CEI 62541-5. D'autres parties de cette série de normes peuvent spécifier des *Types de Données* supplémentaires.

8.2 Identifiant de Nœud

8.2.1 Généralités

Le *Type de Données intégré* est composé de trois éléments qui identifient un *Nœud* au sein d'un serveur. Ils sont définis dans le Tableau 17.

Tableau 17 – Définition d'un Identifiant de Nœud

Dénomination	Type	Description
Identifiant de Nœud	Structure	
Indice d'Espace de Nom	UInt16 (Entier Non Signé codé sur 16 bits)	L'indice pour un Espace de Nom URI (voir 8.2.2).
Type d'Identifiant	Enum	Le format et le type de données de l'Identifiant (voir 8.2.3).
Identifiant	*	L'identifiant utilisé pour un *Nœud* dans l'*Espace d'Adresse* d'un serveur OPC UA (voir 8.2.4).

Pour une description du codage de l'identifiant dans des Messages OPC UA, voir la CEI 62541-6.

8.2.2 Indice d'Espace de Nom

L'Espace de Nom est un URI qui identifie l'autorité de nommage chargée de l'attribution de l'élément identificateur de l'*Identifiant de Nœud.* Les autorités de nommage comprennent le serveur local, le système sous-jacent ainsi que les organismes et consortiums de normalisation. Il est prévu que la plupart des *Nœuds* utiliseront l'URI du serveur ou du système sous-jacent.

L'utilisation d'un Espace de Nom URI permet de relier plusieurs serveurs OPC UA au même système sous-jacent afin d'utiliser le même identifiant pour désigner le même *Objet*. Ainsi, les clients qui se connectent à ces serveurs peuvent reconnaître les *Objets* qu'ils ont en commun.

Les Espaces de Nom URI, tels que les noms de serveurs, sont identifiés par les valeurs numériques dans les *Services* OPC UA afin de garantir une meilleure efficacité de transfert et de traitement (par exemple les consultations de tables). Les valeurs numériques utilisées pour identifier les espaces de noms correspondent à l'indice dans la *Matrice d'Espace de*

Nom. La *Matrice d'Espace de Nom* est une *Variable* qui fait partie de l'*Objet Serveur* dans l'*Espace d'Adresse* (la définition est donnée dans la CEI 62541-5).

L'URI pour l'Espace de Nom OPC UA est le suivant:

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

son indice correspondant dans la table d'Espace de Nom est 0.

8.2.3 Type d'Identifiant

L'élément "Type d'Identifiant" permet d'identifier le type d'*Identifiant de Nœud*, son format et son étendue. Ses valeurs sont définies dans le Tableau 18.

Tableau 18 – Valeurs du Type d'Identifiant

Valeur	Description
NUMERIC_0	Valeur numérique
STRING_1	Valeur de la Chaîne
GUID_2	Identifiant Globalement Unique
OPAQUE_3	Format spécifique de l'Espace de Nom

En général, le champ d'application des *Identifiants de Nœuds* est le serveur dans lequel ils sont définis. Certains types d'*Identifiants de Nœuds* peuvent identifier de manière unique un *Nœud* dans un système donné ou sur plusieurs systèmes (par exemple des Identifiants Universels Globaux - GUID). Des identifiants valables dans l'ensemble du système et globalement uniques permettent à des clients de suivre des *Nœuds*, tels que des commandes de travaux, au fur et à mesure qu'ils se déplacent entre serveurs OPC UA et qu'ils progressent dans le système.

Les identifiants opaques sont constitués par des chaînes d'octets de format libre qui sont ou non interprétables par l'utilisateur.

8.2.4 Valeur d'Identifiant

L'élément valeur d'identifiant est utilisé dans le contexte des trois premiers éléments pour identifier le *Nœud*. Son type de données et son format sont définis par le Type d'Identifiant

Les valeurs d'Identifiant de Type d'Identifiant STRING_1 sont limitées à 4096 caractères. Les valeurs d'Identifiant de Type d'Identifiant OPAQUE_3 sont limitées à 4096 octets.

Un *Identifiant de Nœud* Nul n'a pas de signification particulière. Par exemple, de nombreux services définis dans la CEI 62541-4 spécifient des comportements particuliers si un *Identifiant de Nœud* Nul est transmis comme paramètre. Chaque Type d'Identifiant dispose d'un jeu de valeurs d'Identifiants qui représentent un *Identifiant de Nœud* Nul. Ces valeurs sont résumées dans le Tableau 19.

Tableau 19 – Valeurs Nulles d'Identifiant de Nœud

Type d'Identifiant	Identifiant
NUMERIC_0	0
STRING_1	Une Chaîne Nulle ou Vide ("")
GUID_2	Un Guid (Global Universal Identifier - Identifant Universel Global) initialisé avec des zéros (par exemple 00000000-0000-0000-0000-000000)
OPAQUE_3	Une Chaîne d'Octets de Longueur = 0

Un *Identifiant de Nœud* Nul a toujours un Indice d'Espace de Nom égal à 0.

Un *Nœud* dans l'*Espace d'Adresse* ne doit pas avoir une valeur Nulle comme *Identifiant de Nœud.*

8.3 Nom Qualifié

Le *Type de Données Intégré* contient un nom qualifié. Il est utilisé par exemple comme *Nom d'Exploration*. Ses éléments sont définis dans le Tableau 20. La partie nom du *Nom Qualifié* est limitée à 512 caractères.

Tableau 20 – Définition de Nom Qualifié

Dénomination	Type	Description
Nom Qualifié	Structure	
Indice d'Espace de Nom	UInt16	Indice identifiant l'Espace de Nom qui définit le nom. L'indice est celui de l'Espace de Nom dans la *Matrice d'Espace de Nom* du serveur local. Le client peut lire la *Variable "Matrice d'Espace de Nom"* pour accéder à la valeur de chaîne de caractères de l'Espace de Nom.
Nom	Chaîne	La partie texte de Nom Qualifié.

8.4 Identifiant de Localisation Géographique

Ce *Type de Données Simple* est spécifié comme une chaîne constituée d'un composant langue et d'un composant pays/région, comme spécifié par la norme IETF RFC 3066. Le composant <pays/région> est toujours précédé d'un tiret. Le format de la chaîne de caractères de l'*Identifiant de Localisation Géographique* est présenté ci-dessous:

<langue>[-<pays/région>], où
<langue> est le code ISO 639 à deux lettres identifiant une langue,
<pays/région> est le code ISO 3166 à deux lettres identifiant le pays/région.

Les règles de construction des *Identifiants de Localisation Géographique* définies par la norme IETF RFC 3066 sont limitées de la manière suivante:

a) la spécification permet uniquement l'utilisation de zéro ou d'un composant <pays/région> à la suite d'un composant <langue>;

b) la spécification permet également les codes <pays/région> à trois lettres « -CHS » et « -CHT » pour les paramètres de localisation chinois « Simplifiés » et « Traditionnels »;

c) la spécification permet également l'utilisation d'autres codes <pays/région> considérés nécessaires par le client ou le serveur.

Des exemples d'*Identifiants de Localisation Géographique* utilisés dans l'OPC UA sont donnés dans le Tableau 21. Les clients et les serveurs fournissent toujours des *Identifiants de Localisation Géographique* qui identifient explicitement la langue et le pays/région.

Tableau 21 – Exemples d'Identifiants de Localisation Géographique

Paramètres de localisation géographique	Identifiant de Localisation Géographique d'OPC UA
Anglais	en
Anglais (Etats-Unis)	en-US
Allemand	de
Allemand (Allemagne)	de-DE
Allemand (Autrichien)	de-AT

Une chaîne de caractères vide ou NULLE indique que l'Identifiant de Localisation Géographique est inconnu.

8.5 Texte Localisé

Ce *Type de Données Intégré* définit une structure contenant une chaîne dans une traduction spécifique à un lieu de géographique donné, défini dans l'Identifiant de localisation géographique. Ses éléments sont définis dans le Tableau 22.

Tableau 22 – Définition de Texte Localisé

Dénomination	Type	Description
Texte Localisé	Structure	
texte	Chaîne	Le texte localisé.
localisation géographique	Identifiant de Localisation Géographique	L'Identifiant du lieu géographique (par exemple « en-US »).

8.6 Argument

Ce *Type de Données Structuré* définit une spécification pour les arguments d'entrée ou de sortie d'une *Méthode*. Il est par exemple utilisé dans l'argument d'entrée et de sortie *Propriétés* pour des *Méthodes*. Ses éléments sont décrits dans le Tableau 23.

Tableau 23 – Définition des Arguments

Dénomination	Type	Description
Argument	Structure	
nom	Chaîne	La dénomination de l'argument
Type de Données	Identifiant de Nœud	L'*Identifiant de Nœud* du *Type de Données* de cet argument
Rang de Valeur	Int32	Indique si le *Type de Données* est une matrice et indique le nombre de dimensions de cette matrice. Il peut prendre les valeurs suivantes: *n* > 1: Le Type de Données est une matrice ayant le nombre spécifié de dimensions. Unidimensionnelle (1): Le Type de Données est une matrice unidimensionnelle. Une Ou Plusieurs Dimensions (0): Le Type de Données est une matrice à une ou plusieurs dimensions. Scalaire (−1): Le Type de Données n'est pas une matrice. Indifférente (−2): Le Type de Données peut être scalaire ou une matrice ayant n'importe quel nombre de dimensions. Scalaire Ou Unidimensionnelle (−3): Le Type de Données peut être scalaire ou une matrice unidimensionnelle.
Dimensions de Matrice	UInt32[]	Spécifie la longueur de chaque dimension d'un Type de Données matriciel. Il permet de décrire la capacité du Type de Données et non la taille courante. Le nombre d'éléments doit être égal à la valeur du "*Rang de Valeur*". Doit être nul si le *Rang de Valeur* ≤ 0. Une valeur 0 pour une dimension particulière indique que la dimension a une longueur variable.
description	Texte Localisé	Une description localisée de l'argument.

8.7 BaseDataType

Ce *Type de Données* abstrait définit une valeur qui peut avoir n'importe quel *Type de Données* valide.

Il définit une valeur spéciale NULLE indiquant qu'une valeur n'est pas présente.

8.8 Boolean

Le *Type de Données Intégré* définit une valeur VRAI ou FAUX.

8.9 Byte

Ce *Type de Données Intégré* définit une valeur dans une plage de 0 à 255.

8.10 ByteString

Ce *Type de Données Intégré* définit une valeur qui est une suite de valeurs d'Octets.

8.11 DateTime

Ce *Type de Données Intégré* définit une date dans le calendrier grégorien. Une description détaillée de ce *Type de Données* est fournie dans la CEI 62541-6.

8.12 Double

Ce *Type de Données Intégré* définit une valeur conforme à la définition de type de données à Double Précision de la norme IEEE 754-1985.

8.13 Duration

Ce *Type de Données Simple* est un *Double* qui définit un intervalle de temps en millisecondes (des fractions peuvent être utilisées pour définir des valeurs sous la milliseconde). Des valeurs négatives sont généralement non valides mais peuvent avoir des significations particulières lorsque la *Durée* est utilisée.

8.14 Enumeration

Ce Type de Données abstrait est le *Type de Données* de base de tous les *Types de Données* "Enumération" comme la *Classe de Nœud* définie en 8.30. Tous les *Types de Données* héritant de ce *Type de Données* subissent un traitement spécial pour le codage comme défini dans la CEI 62541-6. Tous les *Types de Données* "Enumération" doivent hériter de ce *Type de Données.*

8.15 Float

Ce *Type de Données Intégré* définit une valeur conforme à la définition de type de données à simple précision de la norme IEEE 754-1985.

8.16 Guid (de l'anglais Globally Unique IDentifier)

Ce *Type de Données Intégré* définit une valeur qui est un Identifiant Globalement Unique de 128 bits. Une description détaillée de ce *Type de Données* est fournie dans la CEI 62541-6.

8.17 SByte

Ce *Type de Données Intégré* définit une valeur qui est un entier signé compris entre -128 et 127 inclus.

8.18 IdType

Ce *Type de Données* est une énumération qui identifie le Type d'Identifiant d'un *Identifiant de Nœud.* Ses valeurs sont définies dans le Tableau 18. Une description de l'usage de ce *Type de Données* dans les *Identifiants de Nœuds* est donnée en 8.2.3.

8.19 Image

Ce *Type de Données* abstrait définit une *Chaîne d'Octets* représentant une image.

8.20 Image BMP

Ce *Type de Données Simple* définit une *Chaîne d'Octets* représentant une image en format BMP.

8.21 Image GIF

Ce *Type de Données Simple* définit une *Chaîne d'Octets* représentant une image en format GIF.

8.22 Image JPG

Ce *Type de Données Simple* définit une *Chaîne d'Octets* représentant une image en format JPG. Le format JPG est défini dans l'ISO/CEI 10918-1.

8.23 Image PNG

Ce *Type de Données Simple* définit une *Chaîne d'Octets* représentant une image en format PNG. Le format PNG est défini dans l'ISO/CEI 15948.

8.24 Integer

Ce *Type de Données* abstrait définit en entier dont la longueur est définie par ses sous-types.

8.25 Int16

Ce *Type de Données Intégré* définit une valeur qui est un entier signé compris entre -32 768 et 32 767 inclus.

8.26 Int32

Ce *Type de Données Intégré* définit une valeur qui est un entier signé compris entre -2 147 483 648 et 2 147 483 647 inclus.

8.27 Int64

Ce *Type de Données Intégré* définit une valeur qui est un entier signé compris entre -9 223 372 036 854 775 808 et 9 223 372 036 854 775 807 inclus.

8.28 TimeZoneDataType

Ce *Type de Données Structuré* définit l'heure locale qui peut ou non tenir compte de l'heure d'été. Ses éléments sont décrits dans le Tableau 23.

Tableau 24 – Définition du Type de Données de Fuseau Horaire

Dénomination	Type	Description
Type de Données de Fuseau Horaire	Structure	
décalage	UInt16	Le décalage en minutes par rapport à l'heure Utc
Décalage d'heure d'Eté	Booléen	Si VRAI, l'heure d'été (DST) est en vigueur et le *décalage* inclut la correction de l'heure d'été. Si FAUX, le *décalage* n'inclut pas la correction de l'heure d'été et celle-ci peut ou non avoir été en vigueur.

8.29 NamingRuleType

Ce *Type de Données* est une énumération qui identifie la *Règle de Nommage* (voir 6.4.4.2.1). Ses valeurs sont définies dans le Tableau 25.