

DIN EN 62541-3

ICS 35.200; 35.240.50

**OPC Unified Architecture –
Teil 3: Adressraummodell (IEC 62541-3:2010);
Englische Fassung EN 62541-3:2010**

OPC unified architecture –
Part 3: Address Space Model (IEC 62541-3:2010);
English version EN 62541-3:2010

Architecture unifiée OPC –
Partie 3: Modèle d'espace d'adressage (CEI 62541-3:2010);
Version anglaise EN 62541-3:2010

Gesamtumfang 118 Seiten

DKE Deutsche Kommission Elektrotechnik Elektronik Informationstechnik im DIN und VDE

Anwendungsbeginn

Anwendungsbeginn für die von CENELEC am 2010-10-01 angenommene Europäische Norm als DIN-Norm ist 2011-09-01.

Nationales Vorwort

Vorausgegangener Norm-Entwurf: E DIN EN 62541-3:2008-12.

Für diese Norm ist das nationale Arbeitsgremium K 931 „Systemaspekte“ der DKE Deutsche Kommission Elektrotechnik Elektronik Informationstechnik im DIN und VDE (www.dke.de) zuständig.

Die enthaltene IEC-Publikation wurde vom SC 65E „Devices and integration in enterprise systems“ erarbeitet.

Das IEC-Komitee hat entschieden, dass der Inhalt dieser Publikation bis zu dem Datum (stability date) unverändert bleiben soll, das auf der IEC-Website unter „<http://webstore.iec.ch>“ zu dieser Publikation angegeben ist. Zu diesem Zeitpunkt wird entsprechend der Entscheidung des Komitees die Publikation

- bestätigt,
- zurückgezogen,
- durch eine Folgeausgabe ersetzt oder
- geändert.

Für den Fall einer undatierten Verweisung im normativen Text (Verweisung auf eine Norm ohne Angabe des Ausgabedatums und ohne Hinweis auf eine Abschnittsnummer, eine Tabelle, ein Bild usw.) bezieht sich die Verweisung auf die jeweils neueste gültige Ausgabe der in Bezug genommenen Norm.

Für den Fall einer datierten Verweisung im normativen Text bezieht sich die Verweisung immer auf die in Bezug genommene Ausgabe der Norm.

Der Zusammenhang der zitierten Normen mit den entsprechenden Deutschen Normen ergibt sich, soweit ein Zusammenhang besteht, grundsätzlich über die Nummer der entsprechenden IEC-Publikation. Beispiel: IEC 60068 ist als EN 60068 als Europäische Norm durch CENELEC übernommen und als DIN EN 60068 ins Deutsche Normenwerk aufgenommen.

Da sich die Benutzer der vorliegenden Norm der englischen Sprache als Fachsprache bedienen, wird die Englische Fassung der DIN EN 62541-3 veröffentlicht. Für die meisten der verwendeten Begriffe existieren keine gebräuchlichen deutschen Benennungen, da sich die deutschen Anwender in der Regel ebenfalls der englischen Benennungen bedienen. Diese Norm steht nicht in unmittelbarem Zusammenhang mit Rechtsvorschriften und ist nicht als Sicherheitsnorm anzusehen.

Das Präsidium des DIN hat mit Präsidialbeschluss 1/2004 festgelegt, dass DIN-Normen, deren Inhalt sich auf internationale Arbeitsergebnisse der Informationsverarbeitung gründet, unter bestimmten Bedingungen allein in englischer Sprache veröffentlicht werden dürfen. Diese Bedingungen sind für die vorliegende Norm erfüllt.

Nationaler Anhang NA (informativ)

NA.3 Begriffe, Symbole, Abkürzungen und Vereinbarungen

NA.3.1 Begriffe

Für die Anwendung dieses Schriftstückes gelten die folgenden sowie die in IEC/TR 62541-1 angegebenen Begriffe.

3.1.1

DataType

(en: **DataType**)

Instanz eines *DataType Node* (Datentypknoten), die gemeinsam mit dem *ValueRank Attribute* (Werterang-attribut) zur Festlegung des Datentyps einer *Variablen* verwendet wird

3.1.2

DataVariable

(en: **DataVariable**)

Variable, die *Werte* von *Objekten*, entweder direkt oder indirekt für komplexe *Variablen*, darstellen; dabei sind die *Variablen* stets der *TargetNode* (Zielknoten) einer *HasCompetent Reference* (Besitzt-Kompetenz-Bezug)

3.1.3

EventType

(en: **EventType**)

ObjectType Node (Objekttypknoten), der die Typdefinition eines *Event* (Ereignis) darstellt

3.1.4

HierarchicalReference

(en: **Hierarchical Reference**)

Bezug, der zum Aufbau von Hierarchien im *AdressSpace* (Adressraum) verwendet wird

ANMERKUNG Alle hierarchischen *ReferenceTypes* (Bezugstypen) werden von *HierarchicalReferences* (hierarchischen Bezügen) abgeleitet.

3.1.5

InstanceDeclaration

(en: **InstanceDeclaration**)

Node (Knoten), der von einem komplexen *TypeDefinitionNode* (Typdefinitions-knoten) zum Zeigen seiner komplexen Struktur verwendet wird; es ist eine Instanz, die von einer Typdefinition verwendet wird

3.1.6

ModellingRule

(en: **ModellingRule**)

Metadaten einer *InstanceDeclaration* (Instanzenbeschreibung), die festlegen, wie die *InstanceDeclaration* für die Instantiierung zu verwenden ist; sie legt auch Untertypenregeln für eine *InstanceDeclaration* fest

3.1.7

Eigenschaft

(en: **Property**)

Variable, die der *TargetNode* für einen *HasProperty Reference* (Besitzt-Eigenschaft-Bezug) ist; *Eigenschaften* beschreiben die Merkmale eines *Node* (Knoten)

3.1.8

SourceNode

(en: **SourceNode**)

Node, der ein *Reference* (Bezug) zu einem weiteren *Node* hat; z. B. in der *Reference* „A enthält B“ ist „A“ der *SourceNode* (Quellknoten)

3.1.9

TargetNode

(en: **TargetNode**)

Node, auf den von einem weiteren *Node* Bezug genommen wird; z. B. in der *Reference* „A enthält B“ ist „B“ der *TargetNode* (Zielknoten)

3.1.10

TypeDefinitionNode

(en: **TypeDefinitionNode**)

Node, der zur Festlegung des Typs eines weiteren *Node* verwendet wird; *ObjectType* (Objektyp-) und *VariableType Nodes* (*Variablentypknoten*) sind *TypDefinitionNodes* (Typdefinitions-knoten)

3.1.11

VariableType

(en: **VariableType**)

Node, der die Typdefinition für eine *Variable* darstellt

**OPC unified architecture -
Part 3: Address Space Model
(IEC 62541-3:2010)**

Architecture unifiée OPC -
Partie 3: Modèle d'espace d'adressage
(CEI 62541-3:2010)

OPC Unified Architecture -
Teil 3: Adressraummodell
(IEC 62541-3:2010)

This European Standard was approved by CENELEC on 2010-10-01. CENELEC members are bound to comply with the CEN/CENELEC Internal Regulations which stipulate the conditions for giving this European Standard the status of a national standard without any alteration.

Up-to-date lists and bibliographical references concerning such national standards may be obtained on application to the Central Secretariat or to any CENELEC member.

This European Standard exists in three official versions (English, French, German). A version in any other language made by translation under the responsibility of a CENELEC member into its own language and notified to the Central Secretariat has the same status as the official versions.

CENELEC members are the national electrotechnical committees of Austria, Belgium, Bulgaria, Croatia, Cyprus, the Czech Republic, Denmark, Estonia, Finland, France, Germany, Greece, Hungary, Iceland, Ireland, Italy, Latvia, Lithuania, Luxembourg, Malta, the Netherlands, Norway, Poland, Portugal, Romania, Slovakia, Slovenia, Spain, Sweden, Switzerland and the United Kingdom.

CENELEC

European Committee for Electrotechnical Standardization
Comité Européen de Normalisation Electrotechnique
Europäisches Komitee für Elektrotechnische Normung

Management Centre: Avenue Marnix 17, B - 1000 Brussels

Foreword

The text of document 64E/160/FDIS, future edition 1 of IEC 62541-3, prepared by SC 65E, Devices and integration in enterprise systems, of IEC TC 65, Industrial-process measurement, control and automation, was submitted to the IEC-CENELEC parallel vote and was approved by CENELEC as EN 62541-3 on 2010-10-01.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. CEN and CENELEC shall not be held responsible for identifying any or all such patent rights.

The following dates were fixed:

- latest date by which the EN has to be implemented
at national level by publication of an identical
national standard or by endorsement (dop) 2011-07-01
- latest date by which the national standards conflicting
with the EN have to be withdrawn (dow) 2013-10-01

Annex ZA has been added by CENELEC.

Endorsement notice

The text of the International Standard IEC 62541-3:2010 was approved by CENELEC as a European Standard without any modification.

CONTENTS

	Page
INTRODUCTION.....	10
1 Scope.....	11
2 Normative references	11
3 Terms, definitions, abbreviations and conventions	12
3.1 Terms and definitions	12
3.2 Abbreviations	13
3.3 Conventions	13
3.3.1 Conventions for AddressSpace figures	13
3.3.2 Conventions for defining NodeClasses	13
4 AddressSpace concepts	15
4.1 Overview	15
4.2 Object Model	15
4.3 Node Model.....	15
4.3.1 General	15
4.3.2 NodeClasses	16
4.3.3 Attributes.....	16
4.3.4 References.....	16
4.4 Variables	17
4.4.1 General	17
4.4.2 Properties.....	17
4.4.3 DataVariables.....	17
4.5 TypeDefinitionNodes	18
4.5.1 General	18
4.5.2 Complex TypeDefinitionNodes and their InstanceDeclarations.....	18
4.5.3 Subtyping	19
4.5.4 Instantiation of complex TypeDefinitionNodes.....	19
4.6 Event Model	21
4.6.1 General	21
4.6.2 EventTypes	21
4.6.3 Event Categorization	21
4.7 Methods	22
5 Standard NodeClasses	22
5.1 Overview	22
5.2 Base NodeClass.....	22
5.2.1 General	22
5.2.2 NodeId	23
5.2.3 NodeClass.....	23
5.2.4 BrowseName	23
5.2.5 DisplayName	23
5.2.6 Description	24
5.2.7 WriteMask	24
5.2.8 UserWriteMask	24

	Page
5.3 ReferenceType NodeClass	25
5.3.1 General	25
5.3.2 Attributes	25
5.3.3 References	27
5.4 View NodeClass	27
5.5 Objects	29
5.5.1 Object NodeClass	29
5.5.2 ObjectType NodeClass	31
5.5.3 Standard ObjectType FolderType	33
5.5.4 Client-side creation of Objects of an ObjectType	33
5.6 Variables	33
5.6.1 General	33
5.6.2 Variable NodeClass	34
5.6.3 Properties	37
5.6.4 DataVariable	38
5.6.5 VariableType NodeClass	38
5.6.6 Client-side creation of Variables of an VariableType	40
5.7 Method NodeClass	40
5.8 DataTypes	42
5.8.1 DataType Model	42
5.8.2 Encoding Rules for different kinds of DataTypes	44
5.8.3 DataType NodeClass	45
5.8.4 DataTypeDictionary, DataTypeDescription, DataTypeEncoding and DataTypeSystem	46
5.9 Summary of Attributes of the NodeClasses	48
6 Type Model for ObjectTypes and VariableTypes	49
6.1 Overview	49
6.2 Definitions	49
6.2.1 InstanceDeclaration	49
6.2.2 Instances without ModellingRules	49
6.2.3 InstanceDeclarationHierarchy	49
6.2.4 Similar Node of InstanceDeclaration	50
6.2.5 BrowsePath	50
6.2.6 Attribute Handling of InstanceDeclarations	50
6.2.7 Attribute Handling of Variable and VariableTypes	50
6.3 Subtyping of ObjectTypes and VariableTypes	50
6.3.1 Overview	50
6.3.2 Attributes	50
6.3.3 InstanceDeclarations	51
6.4 Instances of ObjectTypes and VariableTypes	54
6.4.1 Overview	54
6.4.2 Creating an Instance	54
6.4.3 Constraints on an Instance	55
6.4.4 ModellingRules	56
6.5 Changing Type Definitions that are already used	62
6.6 ModelParent	62

	Page
7 Standard ReferenceTypes	63
7.1 General	63
7.2 References ReferenceType	64
7.3 HierarchicalReferences ReferenceType	64
7.4 NonHierarchicalReferences ReferenceType	65
7.5 HasChild ReferenceType	65
7.6 Aggregates ReferenceType	65
7.7 HasComponent ReferenceType	65
7.8 HasProperty ReferenceType	66
7.9 HasOrderedComponent ReferenceType	66
7.10 HasSubtype ReferenceType	66
7.11 Organizes ReferenceType	66
7.12 HasModellingRule ReferenceType	66
7.13 HasModelParent ReferenceType	67
7.14 HasTypeDefinition ReferenceType	67
7.15 HasEncoding ReferenceType	67
7.16 HasDescription ReferenceType	67
7.17 GeneratesEvent	68
7.18 AlwaysGeneratesEvent	68
7.19 HasEventSource	68
7.20 HasNotifier	69
8 Standard DataTypes	70
8.1 General	70
8.2 NodeId	70
8.2.1 General	70
8.2.2 NamespaceIndex	70
8.2.3 IdentifierType	71
8.2.4 Identifier value	71
8.3 QualifiedName	72
8.4 LocaleId	72
8.5 LocalizedText	72
8.6 Argument	73
8.7 BaseDataType	73
8.8 Boolean	73
8.9 Byte	73
8.10 ByteString	73
8.11 DateTime	74
8.12 Double	74
8.13 Duration	74
8.14 Enumeration	74
8.15 Float	74
8.16 Guid	74
8.17 SByte	74
8.18 IdType	74
8.19 Image	74
8.20 ImageBMP	74
8.21 ImageGIF	74

	Page
8.22 ImageJPG	74
8.23 ImagePNG	75
8.24 Integer	75
8.25 Int16.....	75
8.26 Int32.....	75
8.27 Int64.....	75
8.28 TimeZoneDataType	75
8.29 NamingRuleType.....	75
8.30 NodeClass	75
8.31 Number	76
8.32 String	76
8.33 Structure	76
8.34 UInteger	76
8.35 UInt16	76
8.36 UInt32	76
8.37 UInt64	76
8.38 UtcTime	76
8.39 XmlElement.....	76
9 Standard EventTypes	77
9.1 General	77
9.2 BaseEventType	77
9.3 SystemEventType	78
9.4 AuditEventType	78
9.5 AuditSecurityEventType	79
9.6 AuditChannelEventType	79
9.7 AuditOpenSecureChannelEventType.....	79
9.8 AuditSessionEventType.....	80
9.9 AuditCreateSessionEventType	80
9.10 AuditUrlMismatchEventType.....	80
9.11 AuditActivateSessionEventType	80
9.12 AuditCancelEventType	80
9.13 AuditCertificateEventType	80
9.14 AuditCertificateDataMismatchEventType	80
9.15 AuditCertificateExpiredEventType	80
9.16 AuditCertificateInvalidEventType	80
9.17 AuditCertificateUntrustedEventType	80
9.18 AuditCertificateRevokedEventType.....	81
9.19 AuditCertificateMismatchEventType	81
9.20 AuditNodeManagementEventType.....	81
9.21 AuditAddNodesEventType	81
9.22 AuditDeleteNodesEventType	81
9.23 AuditAddReferencesEventType	81
9.24 AuditDeleteReferencesEventType	81
9.25 AuditUpdateEventType	81
9.26 AuditWriteUpdateEventType.....	81
9.27 AuditHistoryUpdateEventType	81
9.28 AuditUpdateMethodEventType	81

	Page
9.29 DeviceFailureEventType	82
9.30 ModelChangeEvents	82
9.30.1 General	82
9.30.2 NodeVersion Property	82
9.30.3 Views	82
9.30.4 Event Compression	82
9.30.5 BaseModelChangeEvent	82
9.30.6 GeneralModelChangeEvent	83
9.30.7 Guidelines for ModelChangeEvents	83
9.31 SemanticChangeEvent	83
9.31.1 General	83
9.31.2 ViewVersion and NodeVersion Properties	83
9.31.3 Views	83
9.31.4 Event Compression	83
Annex A (informative) How to use the Address Space Model	84
Annex B (informative) OPC UA Meta Model in UML	87
Annex C (normative) OPC Binary Type Description System	96
Annex D (normative) Graphical Notation	108
Bibliography	113
Annex ZA (normative) Normative references to international publications with their corresponding European publications	114
Figure 1 – AddressSpace Node diagrams	13
Figure 2 – OPC UA Object Model	15
Figure 3 – AddressSpace Node Model	16
Figure 4 – Reference Model	17
Figure 5 – Example of a Variable Defined by a VariableType	18
Figure 6 – Example of a Complex TypeDefinition	19
Figure 7 – Object and its Components defined by an ObjectType	20
Figure 8 – Symmetric and Non-Symmetric References	26
Figure 9 – Variables, VariableTypes and their DataTypes	42
Figure 10 – DataType Model	43
Figure 11 – Example of DataType Modelling	48
Figure 12 – Subtyping TypeDefinitionNodes	51
Figure 13 – The Fully-Inherited InstanceDeclarationHierarchy for BetaType	53
Figure 14 – An Instance and its TypeDefinitionNode	54
Figure 15 – Example for several References between InstanceDeclarations	56
Figure 16 – Example on changing instances based on InstanceDeclarations	58
Figure 17 – Example on changing InstanceDeclarations based on an InstanceDeclaration ..	59
Figure 18 – Use of the Standard ModellingRule New	60
Figure 19 – Example using the Standard ModellingRules Optional and Mandatory	61
Figure 20 – Example on using ExposesItsArray	62
Figure 21 – Complex example on using ExposesItsArray	62
Figure 22 – Example on ModelParents	63

	Page
Figure 23 – Standard ReferenceType Hierarchy.....	64
Figure 24 – Event Reference Example	69
Figure 25 – Complex Event Reference Example	70
Figure 26 – Standard EventType Hierarchy.....	77
Figure 27 – Audit Behaviour of a Server	78
Figure 28 – Audit Behaviour of an Aggregating Server	79
Figure B.1 – Background of OPC UA Meta Model	87
Figure B.2 – Notation (I)	88
Figure B.3 – Notation (II)	88
Figure B.4 – BaseNode.....	89
Figure B.5 – Reference and ReferenceType	89
Figure B.6 – Predefined ReferenceTypes.....	90
Figure B.7 – Attributes	91
Figure B.8 – Object and ObjectType	92
Figure B.9 – EventNotifier.....	92
Figure B.10 – Variable and VariableType	93
Figure B.11 – Method	94
Figure B.12 – DataType	95
Figure B.13 – View	95
Figure C.1 – OPC Binary Dictionary Structure.....	96
Figure D.1 – Example of a Reference connecting two Nodes	109
Figure D.2 – Example of using a TypeDefinition inside a Node	111
Figure D.3 – Example of exposing Attributes.....	111
Figure D.4 – Example of exposing Properties inline	112
Table 1 – NodeClass Table Conventions	14
Table 2 – Base NodeClass.....	23
Table 3 – Bit mask for WriteMask and UserWriteMask	24
Table 4 – ReferenceType NodeClass.....	25
Table 5 – View NodeClass	28
Table 6 – Object NodeClass	30
Table 7 – ObjectType NodeClass.....	32
Table 8 – Variable NodeClass.....	34
Table 9 – VariableType NodeClass	39
Table 10 – Method NodeClass	41
Table 11 – DataType NodeClass.....	45
Table 12 – Overview about Attributes	49
Table 13 – The InstanceDeclarationHierarchy for BetaType	52
Table 14 – The Fully-Inherited InstanceDeclarationHierarchy for BetaType.....	52
Table 15 – Rule for ModellingRules Properties when Subtyping.....	57
Table 16 – Properties of ModellingRules	59
Table 17 – NodeId Definition.....	70

	Page
Table 18 – IdentifierType Values.....	71
Table 19 – NodeId Null Values.....	71
Table 20 – QualifiedName Definition.....	72
Table 21 –LocaleId Examples	72
Table 22 – LocalizedText Definition	73
Table 23 – Argument Definition.....	73
Table 24 – TimeZoneDataType Definition	75
Table 25 – NamingRuleType Values	75
Table 26 – NodeClass Values	76
Table C.1 – TypeDictionary Components	97
Table C.2 – TypeDescription Components	98
Table C.3 – OpaqueType Components.....	98
Table C.4 – EnumeratedType Components	99
Table C.5 – StructuredType Components.....	99
Table C.6 – FieldType Components	100
Table C.7 – EnumeratedValue Components.....	101
Table C.8 – ImportDirective Components.....	101
Table C.9 – Standard Type Descriptions	102
Table D.1 – Notation of Nodes depending on the NodeClass	109
Table D.2 – Simple Notation of Nodes depending on the NodeClass.....	110

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 62541-4¹, *OPC Unified Architecture – Part 4: Services*

IEC 62541-5¹, *OPC Unified Architecture – Part 5: Information Model*

IEC 62541-6¹, *OPC Unified Architecture – Part 6: Mappings*

IEC 62541-8¹, *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.

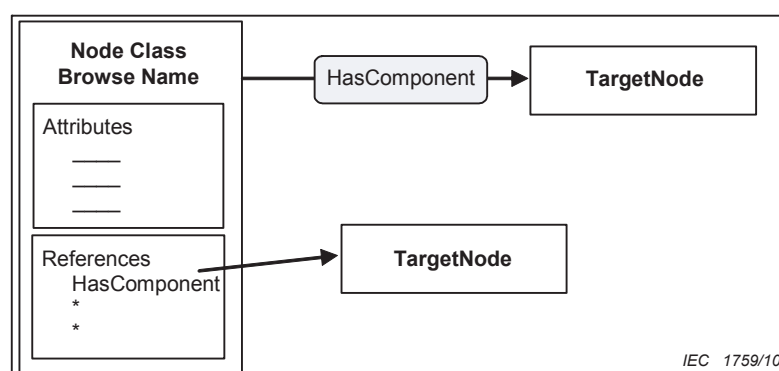


Figure 1 – AddressSpace Node diagrams

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 <i>Attribute</i>	Defines the <i>Attribute</i>
References			
"Reference name"	"1", "0..1" or "0..*"	Not used	Describes the use of the <i>Reference</i> by the <i>NodeClass</i> .
Standard Properties			
"Property name"	"M" or "O"	Data type of the <i>Property</i>	Defines the <i>Property</i> .

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.

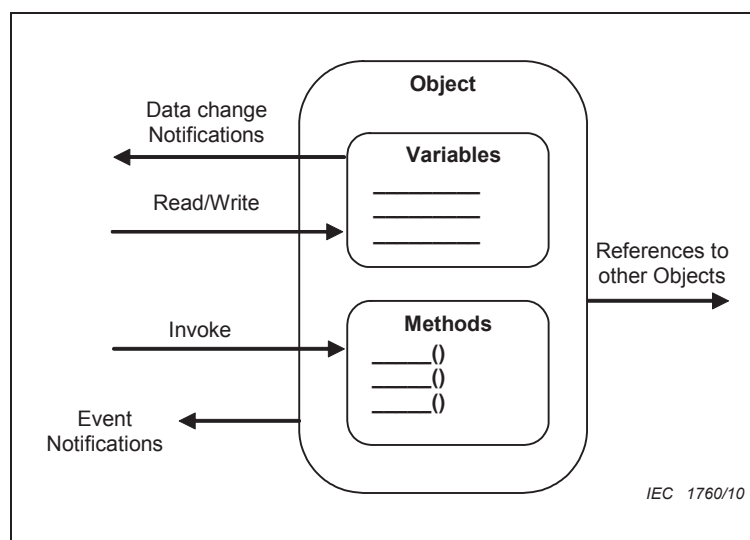


Figure 2 – OPC UA Object Model

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.

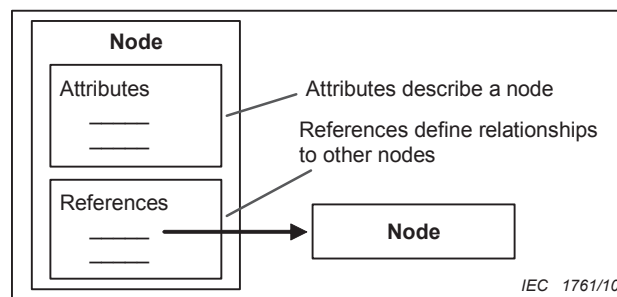


Figure 3 – AddressSpace Node Model

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*.

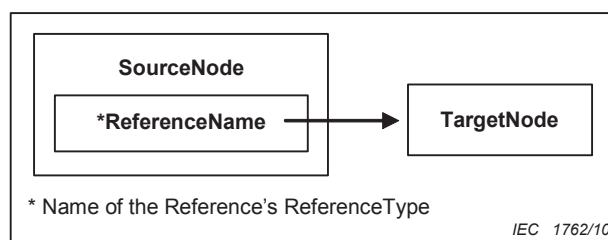


Figure 4 – Reference Model

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*.

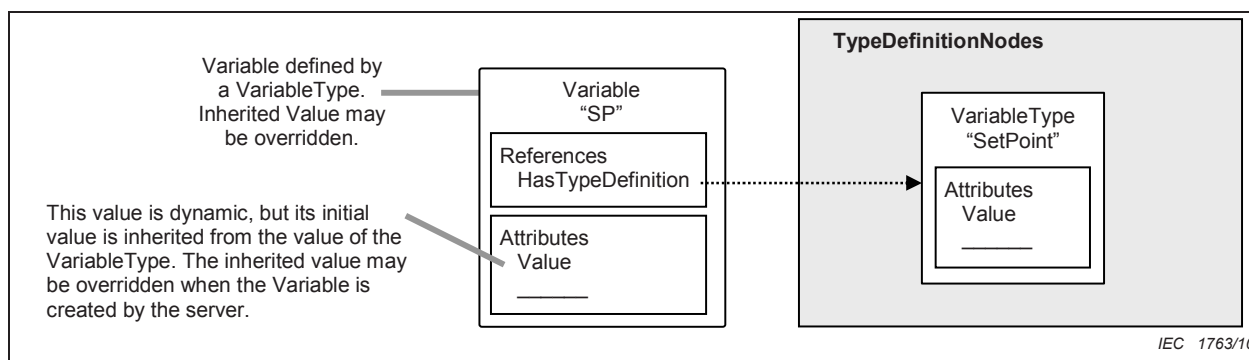


Figure 5 – Example of a Variable Defined by a VariableType

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.

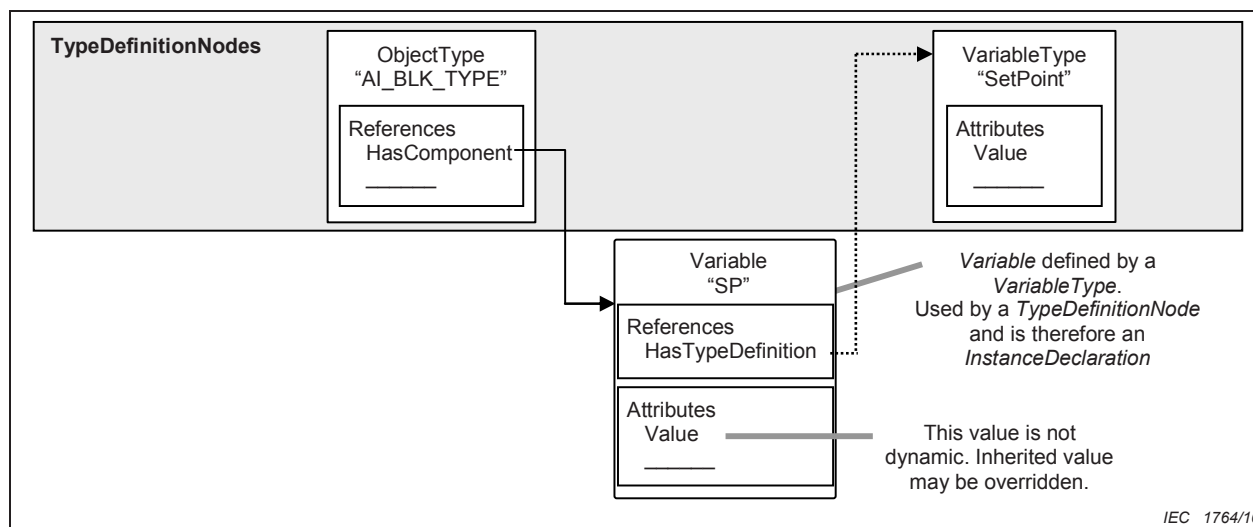


Figure 6 – Example of a Complex TypeDefinition

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*.

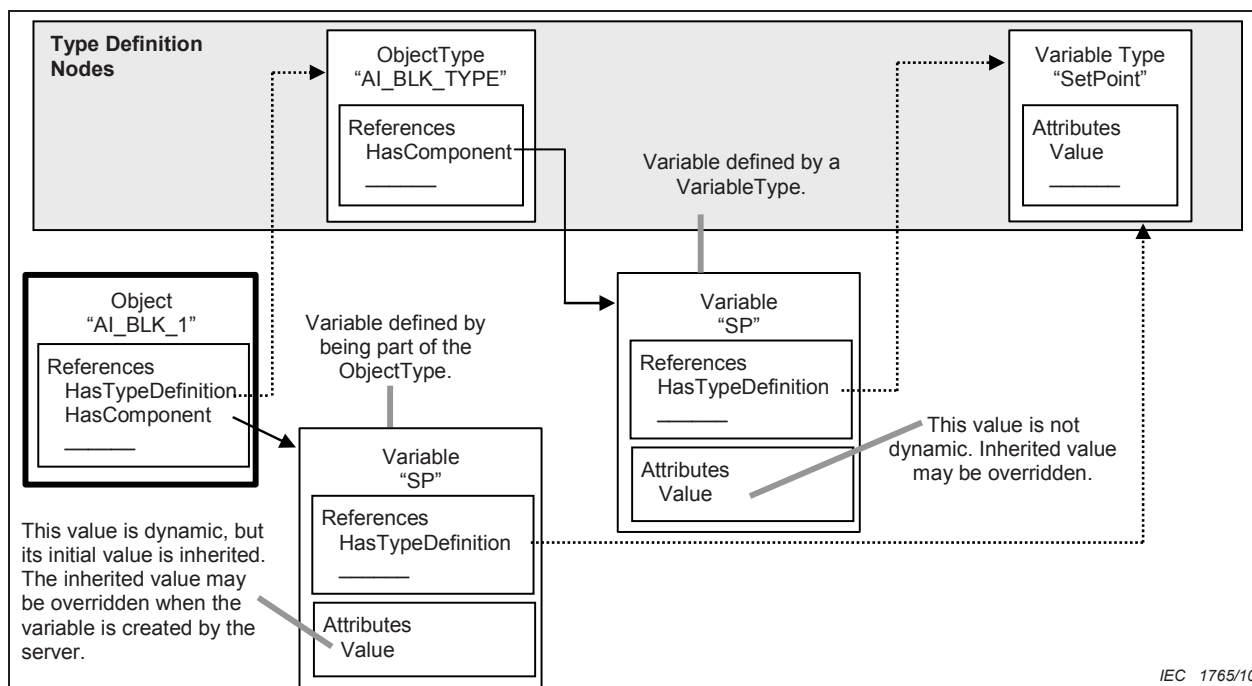


Figure 7 – Object and its Components defined by an ObjectType

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 <i>References</i> specified for this <i>NodeClass</i>

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*.

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

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

It is recommended that 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 <i>Value Attribute</i> is writable for a <i>VariableType</i> . It does not apply for <i>Variables</i> since this is handled by the <i>AccessLevel</i> and <i>UserAccessLevel Attributes</i> for the <i>Variable</i> . For <i>Variables</i> 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 <i>Base NodeClass</i> . See 5.2
IsAbstract	M	Boolean	A boolean <i>Attribute</i> with the following values: TRUE it is an abstract <i>ReferenceType</i> , i.e. no <i>References</i> of this type shall exist, only of its subtypes. FALSE it is not an abstract <i>ReferenceType</i> , i.e. <i>References</i> of this type can exist.
Symmetric	M	Boolean	A boolean <i>Attribute</i> with the following values: TRUE the meaning of the <i>ReferenceType</i> is the same as seen from both the <i>SourceNode</i> and the <i>TargetNode</i> . FALSE the meaning of the <i>ReferenceType</i> as seen from the <i>TargetNode</i> is the inverse of that as seen from the <i>SourceNode</i> .
InverseName	O	LocalizedText	The inverse name of the <i>Reference</i> , that is the meaning of the <i>ReferenceType</i> as seen from the <i>TargetNode</i> .
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 <i>NodeVersion Property</i> is used to indicate the version of a <i>Node</i> . The <i>NodeVersion Property</i> is updated each time a <i>Reference</i> is added or deleted to the <i>Node</i> the <i>Property</i> belongs to. <i>Attribute</i> value changes do not cause the <i>NodeVersion</i> to change. Clients may read the <i>NodeVersion Property</i> or subscribe to it to determine when the structure of a <i>Node</i> 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*.

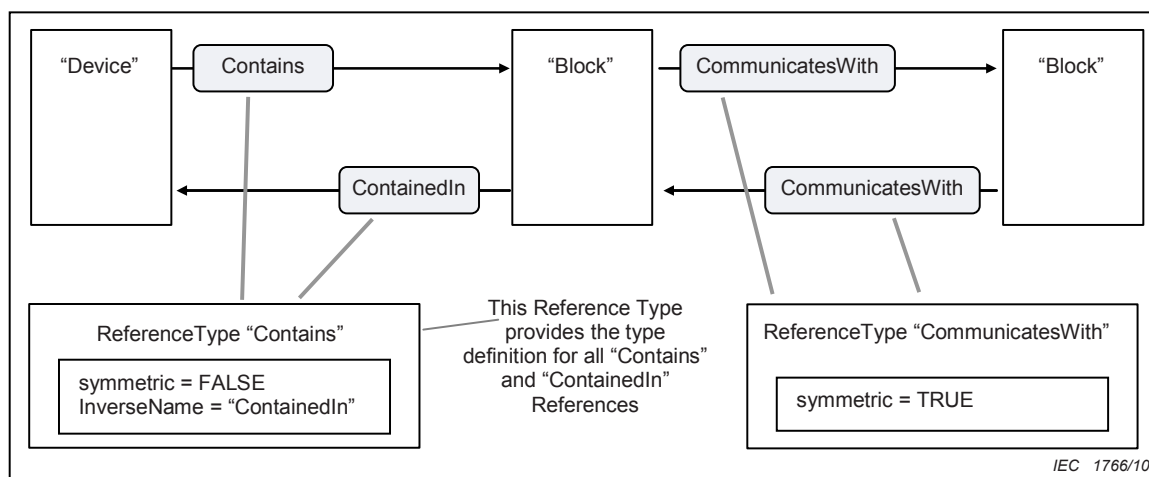


Figure 8 – Symmetric and Non-Symmetric References

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.

- 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.
- 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”).
- 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.
- 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

Name	Use	Data Type	Description																		
Attributes																					
Base NodeClass Attributes	M	--	Inherited from the <i>Base NodeClass</i> . See 5.2																		
ContainsNoLoops	M	Boolean	If set to “true” this <i>Attribute</i> indicates that following <i>References</i> in the context of the <i>View</i> contains no loops, i.e. starting from a <i>Node</i> “A” contained in the <i>View</i> and following the forward <i>References</i> in the context of the <i>View Node</i> “A” will not be reached again. It does not specify that there is only one path starting from the <i>View Node</i> to reach a <i>Node</i> contained in the <i>View</i> . If set to “false” this <i>Attribute</i> indicates that following <i>References</i> in the context of the <i>View</i> may lead to loops.																		
EventNotifier	M	Byte	The <i>EventNotifier Attribute</i> is used to indicate if the <i>Node</i> can be used to subscribe to <i>Events</i> or to read / write historic <i>Events</i>. The <i>EventNotifier</i> 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 <i>Events</i> (0 means cannot be used to subscribe to <i>Events</i>, 1 means can be used to subscribe to <i>Events</i>).</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 <i>Events</i> 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 <i>Events</i> 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 <i>Events</i> is available via the OPC UA server.	Field	Bit	Description	SubscribeTo Events	0	Indicates if it can be used to subscribe to <i>Events</i> (0 means cannot be used to subscribe to <i>Events</i> , 1 means can be used to subscribe to <i>Events</i>).	Reserved	1	Reserved for future use. Shall always be zero.	HistoryRead	2	Indicates if the history of the <i>Events</i> is readable (0 means not readable, 1 means readable).	HistoryWrite	3	Indicates if the history of the <i>Events</i> is writable (0 means not writable, 1 means writable).	Reserved	4:7	Reserved for future use. Shall always be zero.
Field	Bit	Description																			
SubscribeTo Events	0	Indicates if it can be used to subscribe to <i>Events</i> (0 means cannot be used to subscribe to <i>Events</i> , 1 means can be used to subscribe to <i>Events</i>).																			
Reserved	1	Reserved for future use. Shall always be zero.																			
HistoryRead	2	Indicates if the history of the <i>Events</i> is readable (0 means not readable, 1 means readable).																			
HistoryWrite	3	Indicates if the history of the <i>Events</i> is writable (0 means not writable, 1 means writable).																			
Reserved	4:7	Reserved for future use. Shall always be zero.																			
References																					
HierarchicalReferences	0..*		Top level <i>Nodes</i> in a <i>View</i> are referenced by <i>hierarchical References</i> (see 7.3).																		
HasProperty	0..*		<i>HasProperty References</i> identify the <i>Properties</i> of the <i>View</i> .																		
Standard Properties																					
NodeVersion	O	String	The <i>NodeVersion Property</i> is used to indicate the version of a <i>Node</i> . The <i>NodeVersion Property</i> is updated each time a <i>Reference</i> is added or deleted to the <i>Node</i> the <i>Property</i> belongs to. <i>Attribute</i> value changes do not cause the <i>NodeVersion</i> to change. Clients may read the <i>NodeVersion Property</i> or subscribe to it to determine when the structure of a <i>Node</i> has changed.																		
ViewVersion	O	UInt32	The version number for the <i>View</i> . When <i>Nodes</i> are added to or removed from a <i>View</i> , the value of the <i>ViewVersion Property</i> is updated. Clients may detect changes to the composition of a <i>View</i> using this <i>Property</i> . The value of the <i>ViewVersion</i> shall always be greater than 0.																		

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 <i>Base NodeClass</i> . See 5.2																		
EventNotifier	M	Byte	The <i>EventNotifier Attribute</i> is used to indicate if the <i>Node</i> can be used to subscribe to <i>Events</i> or the read / write historic <i>Events</i>. The <i>EventNotifier</i> 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 <i>Events</i> (0 means cannot be used to subscribe to <i>Events</i>, 1 means can be used to subscribe to <i>Events</i>).</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 <i>Events</i> 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 <i>Events</i> 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 <i>Events</i> is available via the OPC UA server.	Field	Bit	Description	SubscribeTo Events	0	Indicates if it can be used to subscribe to <i>Events</i> (0 means cannot be used to subscribe to <i>Events</i> , 1 means can be used to subscribe to <i>Events</i>).	Reserved	1	Reserved for future use. shall always be zero.	HistoryRead	2	Indicates if the history of the <i>Events</i> is readable (0 means not readable, 1 means readable).	HistoryWrite	3	Indicates if the history of the <i>Events</i> is writable (0 means not writable, 1 means writable).	Reserved	4:7	Reserved for future use. Shall always be zero.
Field	Bit	Description																			
SubscribeTo Events	0	Indicates if it can be used to subscribe to <i>Events</i> (0 means cannot be used to subscribe to <i>Events</i> , 1 means can be used to subscribe to <i>Events</i>).																			
Reserved	1	Reserved for future use. shall always be zero.																			
HistoryRead	2	Indicates if the history of the <i>Events</i> is readable (0 means not readable, 1 means readable).																			
HistoryWrite	3	Indicates if the history of the <i>Events</i> is writable (0 means not writable, 1 means writable).																			
Reserved	4:7	Reserved for future use. Shall always be zero.																			
References																					
HasComponent	0..*		<i>HasComponent References</i> identify the <i>DataVariables</i> , the <i>Methods</i> and <i>Objects</i> contained in the <i>Object</i> .																		
HasProperty	0..*		<i>HasProperty References</i> identify the <i>Properties</i> of the <i>Object</i> .																		
HasModellingRule	0..1		<i>Objects</i> can point to at most one <i>ModellingRule Object</i> using a <i>HasModellingRule Reference</i> (see 6.4.4 for details on <i>ModellingRules</i>).																		
HasTypeDefinition	1		The <i>HasTypeDefinition Reference</i> points to the type definition of the <i>Object</i> . Each <i>Object</i> shall have exactly one type definition and therefore be the <i>SourceNode</i> of exactly one <i>HasTypeDefinition Reference</i> pointing to an <i>ObjectType</i> . See 4.5 for a description of type definitions.																		
HasModelParent	0..1		The <i>HasModelParent Reference</i> points to the <i>ModelParent</i> of the <i>Object</i> (see 6.6 for details on <i>ModelParents</i>).																		
HasEventSource	0..*		The <i>HasEventSource Reference</i> points to event sources of the <i>Object</i> . <i>References</i> of this type can only be used for <i>Objects</i> having their “SubscribeToEvents” bit set in the <i>EventNotifier Attribute</i> . See 7.18 for details.																		
HasNotifier	0..*		The <i>HasNotifier Reference</i> points to notifiers of the <i>Object</i> . <i>References</i> of this type can only be used for <i>Objects</i> having their “SubscribeToEvents” bit set in the <i>EventNotifier Attribute</i> . See 7.20 for details.																		
Organizes	0..*		This <i>Reference</i> should be used only for <i>Objects</i> of the <i>ObjectType FolderType</i> (see 5.5.3).																		
HasDescription	0..1		This <i>Reference</i> shall be used only for <i>Objects</i> of the <i>ObjectType DataTypeEncodingType</i> (see 5.8.4).																		
<other References>	0..*		<i>Objects</i> may contain other <i>References</i> .																		
Standard Properties																					
NodeVersion	O	String	The <i>NodeVersion Property</i> is used to indicate the version of a <i>Node</i> . The <i>NodeVersion Property</i> is updated each time a <i>Reference</i> is added or deleted to the <i>Node</i> the <i>Property</i> belongs to. <i>Attribute</i> value changes do not cause the <i>NodeVersion</i> to change. Clients may read the <i>NodeVersion Property</i> or subscribe to it to determine when the structure of a <i>Node</i> has changed.																		
Icon	O	Image	The <i>Icon Property</i> provides an image that can be used by clients when displaying the <i>Node</i> . It is expected that the <i>Icon Property</i> contains a relatively small image.																		
NamingRule	O	NamingRuleType	The <i>NamingRule Property</i> defines the <i>NamingRule</i> of a <i>ModellingRule</i> (see 6.4.4.2.1 for details). This <i>Property</i> shall only be used for <i>Objects</i> of the type <i>ModellingRuleType</i> 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 <i>Base NodeClass</i> . See 5.2
IsAbstract	M	Boolean	A boolean <i>Attribute</i> with the following values: TRUE it is an abstract <i>ObjectType</i> , i.e. no <i>Objects</i> of this type shall exist, only of its subtypes. FALSE it is not an abstract <i>ObjectType</i> , i.e. <i>Objects</i> of this type can exist.
References			
HasComponent	0..*		<i>HasComponent References</i> identify the <i>DataVariables</i> , the <i>Methods</i> , and <i>Objects</i> contained in the <i>ObjectType</i> . If and how the referenced <i>Nodes</i> are instantiated when an <i>Object</i> of this type is instantiated, is specified in 6.4.
HasProperty	0..*		<i>HasProperty References</i> identify the <i>Properties</i> of the <i>ObjectType</i> . If and how the <i>Properties</i> are instantiated when an <i>Object</i> of this type is instantiated, is specified in 6.4.
HasSubtype	0..*		<i>HasSubtype References</i> identify <i>ObjectTypes</i> that are subtypes of this type. The inverse <i>SubtypeOf Reference</i> identifies the parent type of this type.
GeneratesEvent	0..*		<i>GeneratesEvent References</i> identify the type of <i>Events</i> instances of this type may generate.
<other References>	0..*		<i>ObjectTypes</i> may contain other <i>References</i> that can be instantiated by <i>Objects</i> defined by this <i>ObjectType</i> .
Standard Properties			
NodeVersion	O	String	The <i>NodeVersion Property</i> is used to indicate the version of a <i>Node</i> . The <i>NodeVersion Property</i> is updated each time a <i>Reference</i> is added or deleted to the <i>Node</i> the <i>Property</i> belongs to. <i>Attribute</i> value changes do not cause the <i>NodeVersion</i> to change. Clients may read the <i>NodeVersion Property</i> or subscribe to it to determine when the structure of a <i>Node</i> has changed.
Icon	O	Image	The <i>Icon Property</i> provides an image that can be used by clients when displaying the <i>Node</i> . It is expected that the <i>Icon Property</i> 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.