

## **OPC Unified Architecture**

### **Specification**

#### **Part 3: Address Space Model**

**Release 1.01**

**February 6, 2009**

Specification Type:	Industry Standard Specification	Comments:	Report or view errata: <a href="http://www.opcfoundation.org/errata">http://www.opcfoundation.org/errata</a>
Title:	OPC Unified Architecture Part 3 :Address Space Model	Date:	February 6, 2009
Version:	Release 1.01	Software:	MS-Word
		Source:	OPC UA Part 3 - Address Space Model 1.01 Specification.doc
Author:	OPC Foundation	Status:	Release

## CONTENTS

Page

FOREWORD .....	x
<u>AGREEMENT OF USE</u> .....	x
1 Scope .....	1
2 Reference documents .....	1
3 Terms, definitions, abbreviations, and conventions .....	1
3.1 OPC UA Part 1 terms .....	1
3.2 OPC UA Part 2 terms .....	2
3.3 OPC UA Address Space Model terms .....	2
3.3.1 DataType .....	2
3.3.2 DataVariable .....	2
3.3.3 EventType .....	2
3.3.4 Hierarchical Reference .....	2
3.3.5 InstanceDeclaration .....	2
3.3.6 ModellingRule .....	2
3.3.7 Property .....	2
3.3.8 SourceNode .....	3
3.3.9 TargetNode .....	3
3.3.10 TypeDefinitionNode .....	3
3.3.11 VariableType .....	3
3.4 Abbreviations and symbols .....	3
3.5 Conventions .....	3
3.5.1 Conventions for AddressSpace figures .....	3
3.5.2 Conventions for defining NodeClasses .....	4
4 AddressSpace concepts .....	5
4.1 Overview .....	5
4.2 Object Model .....	5
4.3 Node Model .....	5
4.3.1 General .....	5
4.3.2 NodeClasses .....	6
4.3.3 Attributes .....	6
4.3.4 References .....	6
4.4 Variables .....	7
4.4.1 General .....	7
4.4.2 Properties .....	7
4.4.3 DataVariables .....	7
4.5 TypeDefinitionNodes .....	8
4.5.1 Overview .....	8
4.5.2 Complex TypeDefinitionNodes and their InstanceDeclarations .....	8
4.5.3 Subtyping .....	9
4.5.4 Instantiation of complex TypeDefinitionNodes .....	10
4.6 Event Model .....	11
4.6.1 Overview .....	11
4.6.2 EventTypes .....	11

4.6.3	Event Categorization .....	11
4.7	Methods.....	12
5	Standard NodeClasses.....	12
5.1	Overview .....	12
5.2	Base NodeClass .....	12
5.2.1	General.....	12
5.2.2	NodeId .....	13
5.2.3	NodeClass .....	13
5.2.4	BrowseName.....	13
5.2.5	DisplayName.....	13
5.2.6	Description.....	14
5.2.7	WriteMask.....	14
5.2.8	UserWriteMask.....	14
5.3	ReferenceType NodeClass.....	15
5.3.1	General.....	15
5.3.2	Attributes .....	15
5.3.3	References.....	17
5.4	View NodeClass.....	18
5.5	Objects.....	20
5.5.1	Object NodeClass.....	20
5.5.2	ObjectType NodeClass .....	22
5.5.3	Standard ObjectType FolderType.....	23
5.5.4	Client-side creation of Objects of an ObjectType .....	23
5.6	Variables .....	23
5.6.1	General.....	23
5.6.2	Variable NodeClass.....	23
5.6.3	Properties .....	27
5.6.4	DataVariable .....	27
5.6.5	VariableType NodeClass .....	29
5.6.6	Client-side creation of Variables of an VariableType.....	30
5.7	Method NodeClass.....	31
5.8	DataTypes .....	32
5.8.1	DataType Model .....	32
5.8.2	Encoding Rules for different kinds of DataTypes .....	34
5.8.3	DataType NodeClass.....	35
5.8.4	DataTypeDictionary, DataTypeDescription, DataTypeEncoding and DataTypeSystem .....	36
5.9	Summary of Attributes of the NodeClasses .....	39
6	Type Model for ObjectTypes and VariableTypes .....	39
6.1	Overview .....	39
6.2	Definitions .....	39
6.2.1	InstanceDeclaration.....	39
6.2.2	Instances without ModellingRules .....	39
6.2.3	InstanceDeclarationHierarchy .....	40
6.2.4	Similar Node of InstanceDeclaration .....	40
6.2.5	BrowsePath.....	40
6.2.6	Attribute Handling of InstanceDeclarations.....	40
6.2.7	Attribute Handling of Variable and VariableTypes.....	40
6.3	Subtyping of ObjectTypes and VariableTypes .....	41

6.3.1	Overview.....	41
6.3.2	Attributes .....	41
6.3.3	InstanceDeclarations .....	41
6.4	Instances of ObjectTypes and VariableTypes.....	44
6.4.1	Overview.....	44
6.4.2	Creating an Instance .....	44
6.4.3	Constraints on an Instance .....	45
6.4.4	ModellingRules.....	46
6.5	Changing Type Definitions that are already used .....	52
6.6	ModelParent .....	52
7	Standard ReferenceTypes.....	53
7.1	General .....	53
7.2	References ReferenceType .....	54
7.3	HierarchicalReferences ReferenceType.....	54
7.4	NonHierarchicalReferences ReferenceType.....	55
7.5	HasChild ReferenceType.....	55
7.6	Aggregates ReferenceType .....	55
7.7	HasComponent ReferenceType .....	55
7.8	HasProperty ReferenceType .....	56
7.9	HasOrderedComponent ReferenceType.....	56
7.10	HasSubtype ReferenceType .....	56
7.11	Organizes ReferenceType .....	56
7.12	HasModellingRule ReferenceType.....	56
7.13	HasModelParent ReferenceType .....	57
7.14	HasTypeDefinition ReferenceType.....	57
7.15	HasEncoding ReferenceType .....	57
7.16	HasDescription ReferenceType .....	57
7.17	GeneratesEvent .....	58
7.18	AlwaysGeneratesEvent .....	58
7.19	HasEventSource .....	58
7.20	HasNotifier.....	58
8	Standard DataTypes .....	61
8.1	General .....	61
8.2	NodeId.....	61
8.2.1	General.....	61
8.2.2	NamespaceIndex.....	61
8.2.3	IdentifierType .....	61
8.2.4	Identifier value .....	62
8.3	QualifiedName .....	62
8.4	LocaleId.....	63
8.5	LocalizedText .....	63
8.6	Argument.....	64
8.7	BaseDataType .....	64
8.8	Boolean .....	64
8.9	Byte.....	64
8.10	ByteString.....	64
8.11	DateTime.....	64
8.12	Double.....	64
8.13	Duration.....	65

8.14	Enumeration .....	65
8.15	Float .....	65
8.16	Guid .....	65
8.17	SByte .....	65
8.18	IdType .....	65
8.19	Image .....	65
8.20	ImageBMP .....	65
8.21	ImageGIF .....	65
8.22	ImageJPG .....	65
8.23	ImagePNG .....	65
8.24	Integer .....	65
8.25	Int16 .....	66
8.26	Int32 .....	66
8.27	Int64 .....	66
8.28	TimeZoneDataType .....	66
8.29	NamingRuleType .....	66
8.30	NodeClass .....	66
8.31	Number .....	66
8.32	String .....	67
8.33	Structure .....	67
8.34	UInteger .....	67
8.35	UInt16 .....	67
8.36	UInt32 .....	67
8.37	UInt64 .....	67
8.38	UtcTime .....	67
8.39	XmlElement .....	67
9	Standard EventTypes .....	68
9.1	General .....	68
9.2	BaseEventType .....	68
9.3	SystemEventType .....	68
9.4	AuditEventType .....	69
9.5	AuditSecurityEventType .....	70
9.6	AuditChannelEventType .....	70
9.7	AuditOpenSecureChannelEventType .....	70
9.8	AuditSessionEventType .....	70
9.9	AuditCreateSessionEventType .....	71
9.10	AuditUrlMismatchEventType .....	71
9.11	AuditActivateSessionEventType .....	71
9.12	AuditCancelEventType .....	71
9.13	AuditCertificateEventType .....	71
9.14	AuditCertificateDataMismatchEventType .....	71
9.15	AuditCertificateExpiredEventType .....	71
9.16	AuditCertificateInvalidEventType .....	71
9.17	AuditCertificateUntrustedEventType .....	71
9.18	AuditCertificateRevokedEventType .....	71
9.19	AuditCertificateMismatchEventType .....	72
9.20	AuditNodeManagementEventType .....	72
9.21	AuditAddNodesEventType .....	72
9.22	AuditDeleteNodesEventType .....	72

9.23	AuditAddReferencesEventType .....	72
9.24	AuditDeleteReferencesEventType .....	72
9.25	AuditUpdateEventType.....	72
9.26	AuditWriteUpdateEventType.....	72
9.27	AuditHistoryUpdateEventType .....	72
9.28	AuditUpdateMethodEventType .....	72
9.29	DeviceFailureEventType .....	72
9.30	ModelChangeEvents .....	73
9.30.1	General.....	73
9.30.2	NodeVersion Property .....	73
9.30.3	Views .....	73
9.30.4	Event Compression .....	73
9.30.5	BaseModelChangeEvent Type .....	73
9.30.6	GeneralModelChangeEvent Type.....	73
9.30.7	Guidelines for ModelChangeEvents .....	74
9.31	SemanticChangeEvent Type .....	74
9.31.1	General.....	74
9.31.2	ViewVersion and NodeVersion Properties .....	74
9.31.3	Views .....	74
9.31.4	Event Compression .....	74
Annex A	(informative): How to use the Address Space Model .....	75
A.1	Overview .....	75
A.2	Type definitions .....	75
A.3	ObjectTypes .....	75
A.4	VariableTypes.....	75
A.4.1	General.....	75
A.4.2	Properties or DataVariables.....	76
A.4.3	Many Variables and / or complex DataTypes .....	76
A.5	Views .....	77
A.6	Methods.....	77
A.7	Defining ReferenceTypes .....	77
A.8	Defining ModellingRules.....	77
Annex B	(informative): OPC UA Meta Model in UML.....	78
B.1	Background .....	78
B.2	Notation.....	78
B.3	Meta Model.....	79
B.3.1	Base .....	80
B.3.2	ReferenceType.....	80
B.3.3	Predefined ReferenceTypes.....	81
B.3.4	Attributes .....	82
B.3.5	Object and ObjectType .....	83
B.3.6	EventNotifier .....	83
B.3.7	Variable and VariableType.....	84
B.3.8	Method.....	85
B.3.9	DataType .....	86
B.3.10	View.....	86
Annex C	(normative): OPC Binary Type Description System.....	87
C.1	Concepts .....	87
C.2	Schema Description .....	88

C.2.1	TypeDictionary .....	88
C.2.2	TypeDescription .....	89
C.2.3	OpaqueType .....	89
C.2.4	EnumeratedType .....	90
C.2.5	StructuredType .....	90
C.2.6	FieldType .....	90
C.2.7	EnumeratedValue .....	92
C.2.8	ByteOrder .....	92
C.2.9	ImportDirective .....	92
C.3	Standard Type Descriptions .....	92
C.4	Type Description Examples .....	93
C.5	OPC Binary XML Schema .....	95
C.6	OPC Binary Standard TypeDictionary .....	96
Annex D (normative):	Graphical Notation .....	99
D.1	Scope .....	99
D.2	Notation .....	99
D.2.1	Overview .....	99
D.2.2	Simple Notation .....	99
D.2.3	Extended Notation .....	100



## FIGURES

Figure 1 – AddressSpace Node diagrams .....	3
Figure 2 – OPC UA Object Model .....	5
Figure 3 – AddressSpace Node Model .....	6
Figure 4 – Reference Model .....	7
Figure 5 – Example of a Variable Defined By a VariableType .....	8
Figure 6 – Example of a Complex TypeDefinition .....	9
Figure 7 – Object and its Components defined by an ObjectType .....	10
Figure 8 – Symmetric and Non-Symmetric References .....	16
Figure 9 – Variables, VariableTypes and their DataTypes.....	32
Figure 10 – DataType Model.....	33
Figure 11 – Example of DataType Modelling .....	38
Figure 12 – Subtyping TypeDefinitionNodes.....	41
Figure 13 – The Fully-Inherited InstanceDeclarationHierarchy for BetaType .....	43
Figure 14 – An Instance and its TypeDefinitionNode .....	45
Figure 15 – Example for several References between InstanceDeclarations .....	46
Figure 16 – Example on changing instances based on InstanceDeclarations .....	48
Figure 17 – Example on changing InstanceDeclarations based on an InstanceDeclaration ...	49
Figure 18 – Use of the Standard ModellingRule New .....	50
Figure 19 – Example using the Standard ModellingRules Optional and Mandatory .....	51
Figure 20 – Example on using ExposesItsArray.....	52
Figure 21 – Complex example on using ExposesItsArray.....	52
Figure 22 – Example on ModelParents.....	53
Figure 23 – Standard ReferenceType Hierarchy .....	54
Figure 24 – Event Reference Example .....	59
Figure 25 – Complex Event Reference Example.....	60
Figure 26 – Standard EventType Hierarchy .....	68
Figure 27 – Audit Behaviour of a Server .....	69
Figure 28 – Audit Behaviour of an Aggregating Server .....	70
Figure 29 – Background of OPC UA Meta Model .....	78
Figure 30 – Notation (I) .....	78
Figure 31 – Notation (II) .....	79
Figure 32 – Base.....	80
Figure 33 – Reference and ReferenceType .....	80
Figure 34 – Predefined ReferenceTypes .....	81
Figure 35 – Attributes .....	82
Figure 36 – Object and ObjectType.....	83
Figure 37 – EventNotifier.....	83
Figure 38 – Variable and VariableType .....	84
Figure 39 – Method .....	85
Figure 40 – DataType.....	86

Figure 41 – View .....	86
Figure 42 – OPC Binary Dictionary Structure .....	87
Figure 43 –Example of a Reference connecting two Nodes .....	100
Figure 44 – Example of using a TypeDefinition inside a Node.....	101
Figure 45 – Example of exposing Attributes .....	101
Figure 46 – Example of exposing Properties inline .....	102

**TABLES**

Table 1 – NodeClass Table Conventions.....	4
Table 2 - Base NodeClass .....	13
Table 3 – Bit mask for WriteMask and UserWriteMask.....	14
Table 4 – ReferenceType NodeClass.....	15
Table 5 – View NodeClass.....	18
Table 6 – Object NodeClass .....	20
Table 7 – ObjectType NodeClass.....	22
Table 8 – Variable NodeClass .....	24
Table 9 – VariableType NodeClass .....	29
Table 10 – Method NodeClass.....	31
Table 11 – DataType NodeClass .....	35
Table 12 – Overview about Attributes .....	39
Table 13 – The InstanceDeclarationHierarchy for BetaType .....	42
Table 14 – The Fully-Inherited InstanceDeclarationHierarchy for BetaType.....	43
Table 15 – Rule for ModellingRules Properties when Subtyping.....	47
Table 16 – Properties of ModellingRules.....	49
Table 17 – NodeId Definition .....	61
Table 18 – IdType Values.....	62
Table 19 – NodeId Null Values .....	62
Table 20 – QualifiedName Definition.....	62
Table 21 –LocaleId Examples.....	63
Table 22 – LocalizedText Definition .....	63
Table 23 – Argument Definition.....	64
Table 24 – TimeZoneDataType Definition .....	66
Table 25 – NamingRuleType Values .....	66
Table 26 – NodeClass Values.....	66
Table 27 – TypeDictionary Components.....	88
Table 28 – TypeDescription Components.....	89
Table 29 – OpaqueType Components .....	89
Table 30 – EnumeratedType Components.....	90
Table 31 – StructuredType Components .....	90
Table 32 – FieldType Components .....	91
Table 33 – EnumeratedValue Components .....	92
Table 34 – ImportDirective Components .....	92
Table 35 – Standard Type Descriptions .....	93
Table 36 – Notation of Nodes depending on the NodeClass .....	99
Table 37 – Simple Notation of Nodes depending on the NodeClass .....	100

## OPC FOUNDATION

---

### UNIFIED ARCHITECTURE –

#### FOREWORD

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

Copyright © 2006-2009, OPC Foundation, Inc.

#### AGREEMENT OF USE

##### COPYRIGHT RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

OPC Foundation members and non-members are prohibited from copying and redistributing this specification. All copies must be obtained on an individual basis, directly from the OPC Foundation Web site <http://www.opcfoundation.org>.

##### PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OPC specifications may require use of an invention covered by patent rights. OPC shall not be responsible for identifying patents for which a license may be required by any OPC specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OPC specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

##### WARRANTY AND LIABILITY DISCLAIMERS

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OPC FOUNDATION MAKES NO WARRANTY OF ANY KIND, EXPRESSED OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE. IN NO EVENT SHALL THE OPC FOUNDATION BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you.

##### RESTRICTED RIGHTS LEGEND

This Specification is provided with Restricted Rights. Use, duplication or disclosure by the U.S. government is subject to restrictions as set forth in (a) this Agreement pursuant to DFARs 227.7202-3(a); (b) subparagraph (c)(1)(i) of the Rights in Technical Data and Computer Software clause at DFARs 252.227-7013; or (c) the Commercial Computer Software Restricted Rights clause at FAR 52.227-19 subdivision (c)(1) and (2), as applicable. Contractor / manufacturer are the OPC Foundation, 16101 N. 82nd Street, Suite 3B, Scottsdale, AZ, 85260-1830

## COMPLIANCE

The OPC Foundation shall at all times be the sole entity that may authorize developers, suppliers and sellers of hardware and software to use certification marks, trademarks or other special designations to indicate compliance with these materials. Products developed using this specification may claim compliance or conformance with this specification if and only if the software satisfactorily meets the certification requirements set by the OPC Foundation. Products that do not meet these requirements may claim only that the product was based on this specification and must not claim compliance or conformance with this specification.

## TRADEMARKS

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

## GENERAL PROVISIONS

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

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

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

## ISSUE REPORTING

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



# OPC Unified Architecture Specification

## Part 3: Address Space Model

### 1 Scope

This specification 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 Reference documents

Part 1: OPC UA Specification: Part 1 – Concepts, Version 1.01 or later

<http://www.opcfoundation.org/UA/Part1/>

Part 2: OPC UA Specification: Part 2 – Security Model, Version 1.01 or later

<http://www.opcfoundation.org/UA/Part2/>

Part 4: OPC UA Specification: Part 4 – Services, Version 1.01 or later

<http://www.opcfoundation.org/UA/Part4/>

Part 5: OPC UA Specification: Part 5 – Information Model, Version 1.01 or later

<http://www.opcfoundation.org/UA/Part5/>

Part 6: OPC UA Specification: Part 6 – Mappings, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part6/>

Part 8: OPC UA Specification: Part 8 – Data Access, Version 1.01 or later

<http://www.opcfoundation.org/UA/Part8/>

Part 9: OPC UA Specification: Part 9 – Alarms and Conditions, Version 1.0 or later

<http://www.opcfoundation.org/UA/Part9/>

Part 11: OPC UA Specification: Part 11 – Historical Access, Version 1.01 or later

<http://www.opcfoundation.org/UA/Part11/>

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

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

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

### 3 Terms, definitions, abbreviations, and conventions

#### 3.1 OPC UA Part 1 terms

The following terms defined in Part 1 apply.

- 1) AddressSpace
- 2) Attribute
- 3) Complex Data
- 4) Event
- 5) Information Model
- 6) Message

- 7) Method
- 8) Node
- 9) NodeClass
- 10) Notification
- 11) Object
- 12) ObjectType
- 13) Reference
- 14) ReferenceType
- 15) Service
- 16) Subscription
- 17) Variable
- 18) View

### 3.2 OPC UA Part 2 terms

There are no Part 2 terms used in this part.

### 3.3 OPC UA Address Space Model terms

#### 3.3.1 DataType

A *DataType* is represented by a *DataType Node*. The *DataType* is used together with the *ValueRank Attribute* to define the data type of a *Variable*.

#### 3.3.2 DataVariable

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

#### 3.3.3 EventType

An *ObjectType Node* that represents the type definition of an *Event*.

#### 3.3.4 Hierarchical Reference

A *Reference* that is used to construct hierarchies in the *AddressSpace*.

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

#### 3.3.5 InstanceDeclaration

A *Node* that is used by a complex *TypeDefinitionNode* to expose its complex structure. It is an instance used by a type definition.

#### 3.3.6 ModellingRule

Metadata of an *InstanceDeclaration* that defines how the *InstanceDeclaration* will be used for instantiation. It also defines subtyping rules for an *InstanceDeclaration*.

#### 3.3.7 Property

*Variables* that are the *TargetNode* for a *HasProperty Reference*. *Properties* describe the characteristics of a *Node*.



### 3.3.8 SourceNode

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

### 3.3.9 TargetNode

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

### 3.3.10 TypeDefinitionNode

A *Node* that is used to define the type of another *Node*. *ObjectType* and *VariableType* *Nodes* are *TypeDefinitionNodes*.

### 3.3.11 VariableType

A *Node* that represents the type definition for a *Variable*.

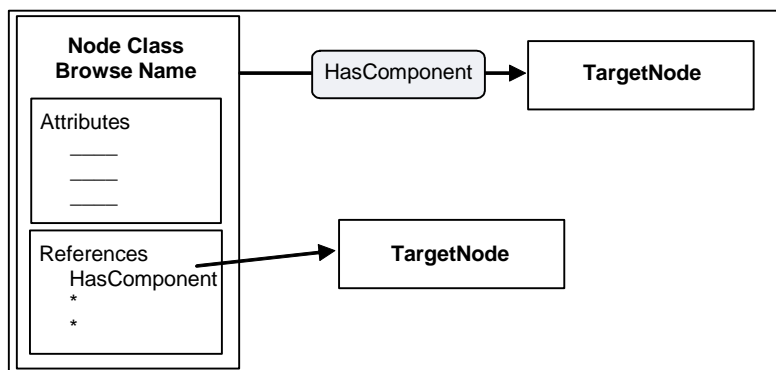
## 3.4 Abbreviations and symbols

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

## 3.5 Conventions

### 3.5.1 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*.

### 3.5.2 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
<b>Attributes</b>			
"Attribute name"	"M" or "O"	Data type of the <i>Attribute</i>	Defines the <i>Attribute</i> .
<b>References</b>			
"Reference name"	"1", "0..1" or "0..*"	Not used	Describes the use of the <i>Reference</i> by the <i>NodeClass</i> .
<b>Standard Properties</b>			
"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* must 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 specification may define *Attributes*. Thus, all *Attributes* of the *NodeClass* are specified in the table and can only be extended by other parts of this multi-part specification.

This specification also defines *ReferenceTypes*, but *ReferenceTypes* can also be specified by a server or by a client using the *NodeManagement Services* specified in Part 4. Thus, the *NodeClass* tables contained in this specification 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 specification 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 document are defined by their name, which is mapped to the *BrowseName* having the *NamespaceIndex* 0, which represents the *Namespace* for OPC UA.

The "use" column (optional or mandatory) does not imply a specific *ModellingRule* for *Properties*. Different server 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 7-9.

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.

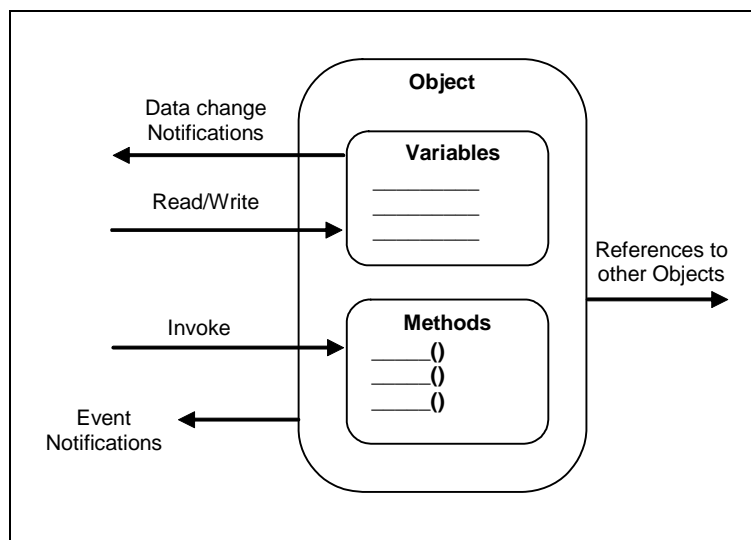


Figure 2 – OPC UA Object Model

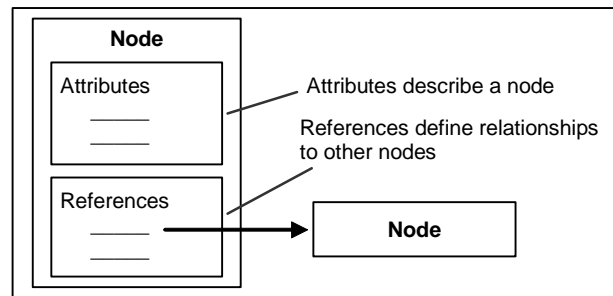
The elements of this model are represented in the *AddressSpace* as *Nodes*. Each *Node* is assigned to a *NodeClass* and each *NodeClass* represents a different element of the Object Model. Clause 5 defines the *NodeClasses* used to represent this model.

### 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 Part 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<sup>1</sup>, 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.

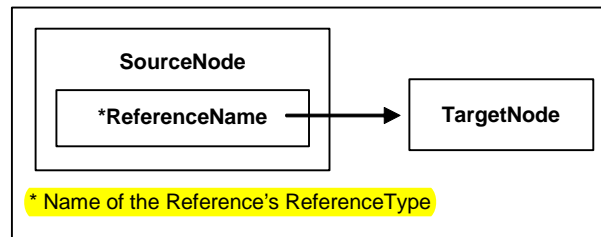
### 4.3.4 References

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

<sup>1</sup> The attribute ids of *Attributes* are defined in Part 6.



**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 Overview

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, Part 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*.

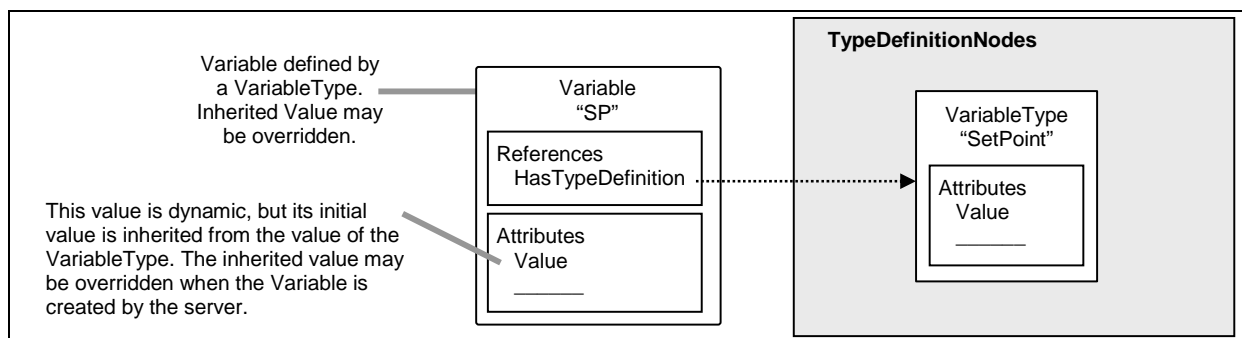


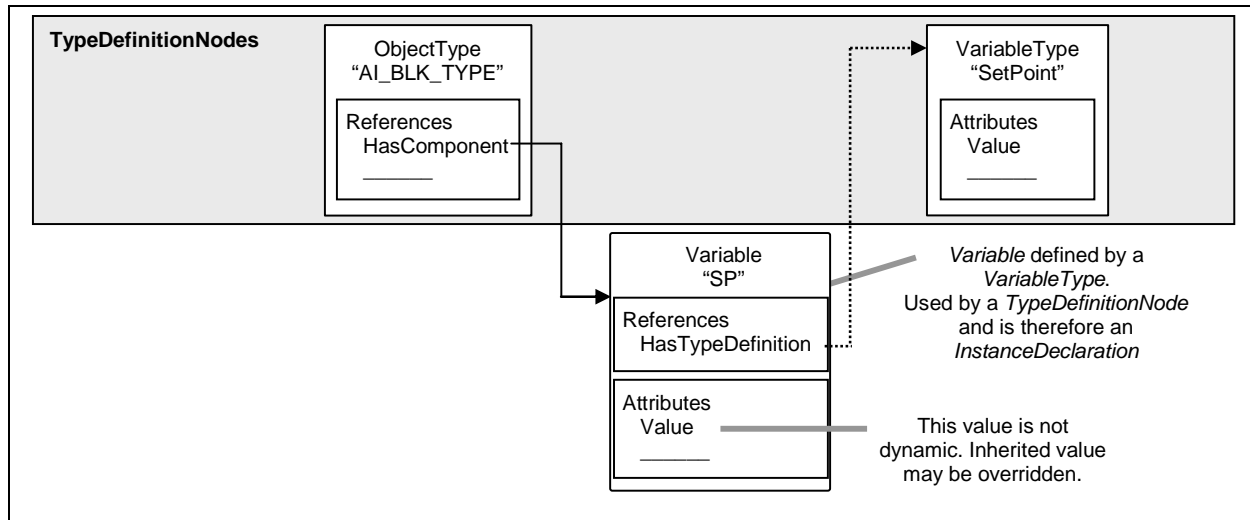
Figure 5 – Example of a Variable Defined By a VariableType

### 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 specification 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 is 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*.

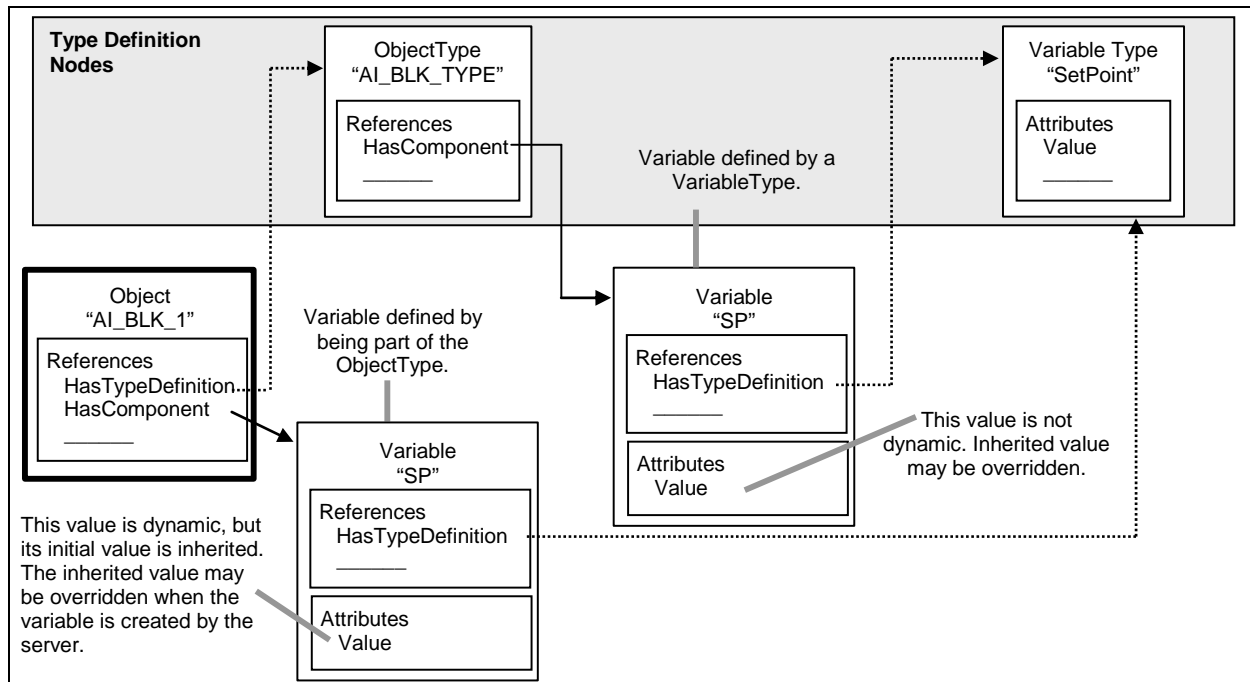


Figure 7 – Object and its Components defined by an ObjectType

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 Part 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 Overview

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 Part 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 Part 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. Part 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 multi-part specification 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 Part 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 7.19 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<sup>2</sup> *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.

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 Part 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

---

<sup>2</sup> The owning *Object* or *ObjectType* is specified in the service call when invoking the *Method*.

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
<b>Attributes</b>			
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
<b>References</b>			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 Part 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*.

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 Part 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, i.e. 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 Part 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 <i>AccessLevel Attribute</i> is writable.
ArrayDimensions	1	Indicates if the <i>ArrayDimensions Attribute</i> is writable.
BrowseName	2	Indicates if the <i>BrowseName Attribute</i> is writable.
ContainsNoLoops	3	Indicates if the <i>ContainsNoLoops Attribute</i> is writable.
DataType	4	Indicates if the <i>DataType Attribute</i> is writable.
Description	5	Indicates if the <i>Description Attribute</i> is writable.
DisplayName	6	Indicates if the <i>DisplayName Attribute</i> is writable.
EventNotifier	7	Indicates if the <i>EventNotifier Attribute</i> is writable.
Executable	8	Indicates if the <i>Executable Attribute</i> is writable.
Historizing	9	Indicates if the <i>Historizing Attribute</i> is writable.
InverseName	10	Indicates if the <i>InverseName Attribute</i> is writable.
IsAbstract	11	Indicates if the <i>IsAbstract Attribute</i> is writable.
MinimumSamplingInterval	12	Indicates if the <i>MinimumSamplingInterval Attribute</i> is writable.
NodeClass	13	Indicates if the <i>NodeClass Attribute</i> is writable.
NodeId	14	Indicates if the <i>NodeId Attribute</i> is writable.
Symmetric	15	Indicates if the <i>Symmetric Attribute</i> is writable.
UserAccessLevel	16	Indicates if the <i>UserAccessLevel Attribute</i> is writable.
UserExecutable	17	Indicates if the <i>UserExecutable Attribute</i> is writable.
UserWriteMask	18	Indicates if the <i>UserWriteMask Attribute</i> is writable.
ValueRank	19	Indicates if the <i>ValueRank Attribute</i> is writable.
WriteMask	20	Indicates if the <i>WriteMask Attribute</i> 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 specification 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 Part 5. Servers may also define *ReferenceTypes*. In addition, Part 4 defines *NodeManagement Services* that allow clients to add *ReferenceTypes* to the *AddressSpace*.

**Table 4 – ReferenceType NodeClass**

Name	Use	Data Type	Description
<b>Attributes</b>			
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> , i.e. the meaning of the <i>ReferenceType</i> as seen from the <i>TargetNode</i> .
<b>References</b>			
HasProperty	0..*		Used to identify the Properties (See 5.3.3.2)
HasSubtype	0..*		Used to identify subtypes (See 5.3.3.3)
<b>Standard Properties</b>			
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* can not be instantiated and are used only for organizational reasons, e.g. to specify some general semantics or constraints that are inherited to its subtypes.

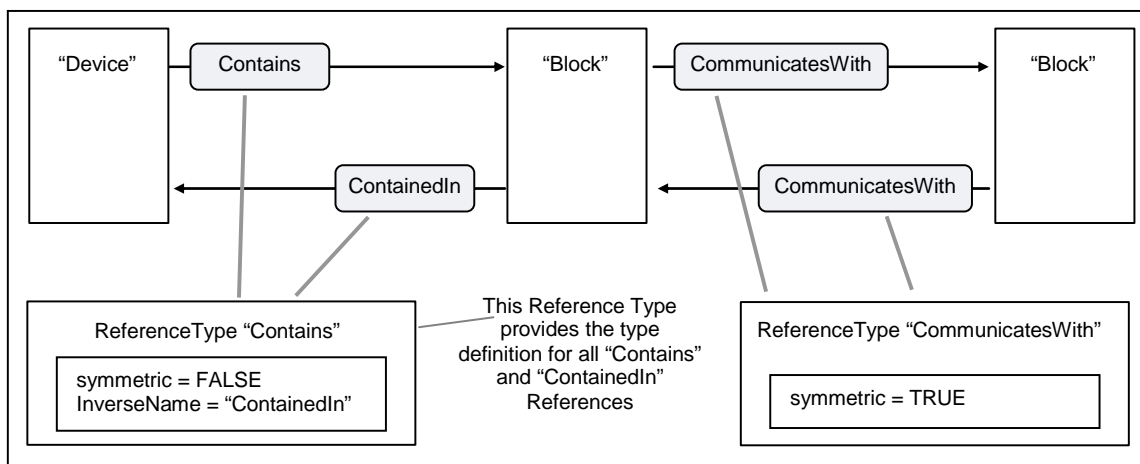
The *Symmetric Attribute* is used to indicate whether or not the meaning of the *ReferenceType* is the same for both the *SourceNode* and *TargetNode*.

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 this figure. 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 specification does not specify how this semantic should be exposed. However, the *Description Attribute* can be used for this purpose. This specification does provide 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 specification 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 specification 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 document. Additional parts of this multi-part specification 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:

1. 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.
2. 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”).
3. 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.
4. 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 Part 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 Part 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 multi-part specification 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																		
<b>Attributes</b>																					
Base NodeClass Attributes	M	--	Inherited from the <i>Base NodeClass</i> . See 5.2																		
EventNotifier	M	Byte	<p>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:</p> <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> <p>The second two bits also indicate if the history of the <i>Events</i> is available via the OPC UA server.</p>	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.																			
<b>References</b>																					
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 7.19 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> .																		
<b>Standard Properties</b>																					
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 multi-part specification 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 7.19 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
<b>Attributes</b>			
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.
<b>References</b>			
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> .
<b>Standard Properties</b>			
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 multi-part specification 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 Part 5. Its purpose is to provide *Objects* that have no other semantic than organizing of the *AddressSpace*. A special *ReferenceType* is used 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 Part 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 <i>Base NodeClass</i> . See 5.2																					
Value	M	Defined by the <i>Data Type Attribute</i>	The most recent value of the <i>Variable</i> that the server has. Its data type is defined by the <i>Data Type Attribute</i> . It is the only <i>Attribute</i> that does not have a data type associated with it. This allows all <i>Variables</i> to have a value defined by the same <i>Value Attribute</i> .																					
DataType	M	NodeId	<i>NodeId</i> of the <i>Data Type</i> definition for the <i>Value Attribute</i> . Standard <i>Data Types</i> are defined in Clause 8.																					
ValueRank	M	Int32	This <i>Attribute</i> indicates whether the <i>Value Attribute</i> of the <i>Variable</i> 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 <i>Attribute</i> specifies the length of each dimension for an array value. The <i>Attribute</i> is intended to describe the capability of the <i>Variable</i> , not the current size. The number of elements shall be equal to the value of the <i>ValueRank Attribute</i> . Shall be null if <i>ValueRank</i> <= 0. A value of 0 for an individual dimension indicates that the dimension has a variable length. For example, if a <i>Variable</i> is defined by the following C array: Int32 myArray[346]; then this <i>Variable's Data Type</i> would point to an Int32, the <i>Variable's ValueRank</i> has the value 1 and the <i>ArrayDimensions</i> is an array with one entry having the value 346.																					
AccessLevel	M	Byte	<p>The <i>AccessLevel Attribute</i> is used to indicate how the <i>Value</i> of a <i>Variable</i> can be accessed (read/write) and if it contains current and/or historic data. The <i>AccessLevel</i> does not take any user access rights into account, i.e. although the <i>Variable</i> is writeable this may be restricted to a certain user / user group.</p> <p>The <i>AccessLevel</i> is an 8-bit unsigned integer with the structure defined in the following table:</p> <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 <i>Variable</i> used as <i>Property</i> generates <i>SemanticChangeEvents</i> (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> <p>The first two bits also indicate if a current value of this <i>Variable</i> is available</p>	Field	Bit	Description	CurrentRead	0	Indicates if the current value is readable (0 means not readable, 1 means readable).	CurrentWrite	1	Indicates if the current value is writable (0 means not writable, 1 means writable).	HistoryRead	2	Indicates if the history of the value is readable (0 means not readable, 1 means readable).	HistoryWrite	3	Indicates if the history of the value is writable (0 means not writable, 1 means writable).	SemanticChange	4	Indicates if the <i>Variable</i> used as <i>Property</i> generates <i>SemanticChangeEvents</i> (see 9.31).	Reserved	5:7	Reserved for future use. Shall always be zero.
Field	Bit	Description																						
CurrentRead	0	Indicates if the current value is readable (0 means not readable, 1 means readable).																						
CurrentWrite	1	Indicates if the current value is writable (0 means not writable, 1 means writable).																						
HistoryRead	2	Indicates if the history of the value is readable (0 means not readable, 1 means readable).																						
HistoryWrite	3	Indicates if the history of the value is writable (0 means not writable, 1 means writable).																						
SemanticChange	4	Indicates if the <i>Variable</i> used as <i>Property</i> generates <i>SemanticChangeEvents</i> (see 9.31).																						
Reserved	5:7	Reserved for future use. Shall always be zero.																						

			and the second two bits indicates if the history of the <i>Variable</i> is available via the OPC UA server.																		
UserAccessLevel	M	Byte	<p>The <i>UserAccessLevel Attribute</i> is used to indicate how the <i>Value</i> of a <i>Variable</i> can be accessed (read/write) and if it contains current or historic data taking user access rights into account.</p> <p>The <i>UserAccessLevel</i> is an 8-bit unsigned integer with the structure defined in the following table:</p> <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> <p>The first two bits also indicate if a current value of this <i>Variable</i> is available and the second two bits indicate if the history of the <i>Variable</i> is available via the OPC UA server.</p>	Field	Bit	Description	CurrentRead	0	Indicates if the current value is readable (0 means not readable, 1 means readable).	CurrentWrite	1	Indicates if the current value is writable (0 means not writable, 1 means writable).	HistoryRead	2	Indicates if the history of the value is readable (0 means not readable, 1 means readable).	HistoryWrite	3	Indicates if the history of the value is writable (0 means not writable, 1 means writable).	Reserved	4:7	Reserved for future use. Shall always be zero.
Field	Bit	Description																			
CurrentRead	0	Indicates if the current value is readable (0 means not readable, 1 means readable).																			
CurrentWrite	1	Indicates if the current value is writable (0 means not writable, 1 means writable).																			
HistoryRead	2	Indicates if the history of the value is readable (0 means not readable, 1 means readable).																			
HistoryWrite	3	Indicates if the history of the value is writable (0 means not writable, 1 means writable).																			
Reserved	4:7	Reserved for future use. Shall always be zero.																			
MinimumSamplingInterval	O	Duration	<p>The <i>MinimumSamplingInterval Attribute</i> indicates how “current” the <i>Value</i> of the <i>Variable</i> will be kept. It specifies (in milliseconds) how fast the server can reasonably sample the value for changes (see Part 4 for a detailed description of sampling interval).</p> <p>A <i>MinimumSamplingInterval</i> of 0 indicates that the server is to monitor the item continuously. A <i>MinimumSamplingInterval</i> of -1 means indeterminate.</p>																		
Historizing	M	Boolean	<p>The <i>Historizing Attribute</i> indicates whether the <i>Server</i> is actively collecting data for the history of the <i>Variable</i>. This differs from the <i>AccessLevel Attribute</i> which identifies if the <i>Variable</i> has any historical data. A value of TRUE indicates that the <i>Server</i> is actively collecting data. A value of FALSE indicates the <i>Server</i> is not actively collecting data. Default value is FALSE.</p>																		
References																					
HasModellingRule	0..1		<i>Variables</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> ).																		
HasProperty	0..*		<i>HasProperty References</i> are used to identify the <i>Properties</i> of a <i>DataVariable</i> . <i>Properties</i> are not allowed to be the <i>SourceNode</i> of <i>HasProperty References</i> .																		
HasComponent	0..*		<i>HasComponent References</i> are used by complex <i>DataVariables</i> to identify their composed <i>DataVariables</i> . <i>Properties</i> are not allowed to use this <i>Reference</i> .																		
HasTypeDefinition	1		The <i>HasTypeDefinition Reference</i> points to the type definition of the <i>Variable</i> . Each <i>Variable</i> shall have exactly one type definition and therefore be the <i>SourceNode</i> of exactly one <i>HasTypeDefinition Reference</i> pointing to a <i>VariableType</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>Variable</i> (see 6.6 for details on <i>ModelParents</i> ).																		
<other References>	0..*		<i>Data Variables</i> may be the <i>SourceNode</i> of any other <i>References</i> . <i>Properties</i> may only be the <i>SourceNode</i> of any <i>non-hierarchical Reference</i> .																		
Standard Properties																					
NodeVersion	O	String	<p>The <i>NodeVersion Property</i> is used to indicate the version of a <i>DataVariable</i>. It does not apply to <i>Properties</i>.</p> <p>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 except for the <i>DataType Attribute</i> 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.</p> <p>Although the relationship of a <i>Variable</i> to its <i>DataType</i> is not modelled using <i>References</i>, changes to the <i>DataType Attribute</i> of a <i>Variable</i> lead to an update of the <i>NodeVersion Property</i>.</p>																		
LocalTime	O	TimeZone-DataType	<p>The <i>LocalTime Property</i> is only used for <i>DataVariables</i>. It does not apply to <i>Properties</i>.</p> <p>This <i>Property</i> 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 Part 4.</p>																		

			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 <i>Variables</i> of the <i>VariableType</i> <i>DataTypeDictionaryType</i> and <i>DataTypeDescriptionType</i> as described in 5.8.
DictionaryFragment	O	ByteString	Only used for <i>Variables</i> of the <i>VariableType</i> <i>DataTypeDescriptionType</i> as described in 5.8.
AllowNulls	O	Boolean	The <i>AllowNulls Property</i> is only used for <i>DataVariables</i> . It does not apply to <i>Properties</i> . This <i>Property</i> specifies if a NULL value is allowed for the <i>Value Attribute</i> of the <i>DataVariable</i> . 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 <i>Property</i> 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 Part 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 Part 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 Part 11 for details on historizing *Variables*.

Clients may read or write *Variable* values, or monitor them for value changes, as specified in Part 4. Part 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 Part 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*. Part 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
<b>Attributes</b>			
Base NodeClass Attributes	M	--	Inherited from the <i>Base NodeClass</i> . See 5.2
Value	O	Defined by the <i>DataType</i> attribute	The default <i>Value</i> for instances of this type.
DataType	M	NodeId	<i>NodeId</i> of the data type definition for instances of this type.
ValueRank	M	Int32	This <i>Attribute</i> indicates whether the <i>Value Attribute</i> of the <i>VariableType</i> 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 <i>Attribute</i> specifies the length of each dimension for an array value. The <i>Attribute</i> is intended to describe the capability of the <i>VariableType</i> , not the current size. The number of elements shall be equal to the value of the <i>ValueRank Attribute</i> . Shall be null if <i>ValueRank</i> <= 0. A value of 0 for an individual dimension indicates that the dimension has a variable length. For example, if a <i>VariableType</i> is defined by the following C array: Int32 myArray[346]; then this <i>VariableType</i> 's <i>DataType</i> would point to an Int32, the <i>VariableType</i> 's <i>ValueRank</i> has the value 1 and the <i>ArrayDimensions</i> is an array with one entry having the value 346.
IsAbstract	M	Boolean	A boolean <i>Attribute</i> with the following values: TRUE it is an abstract <i>VariableType</i> , i.e. no <i>Variable</i> of this type shall exist, only of its subtypes. FALSE it is not an abstract <i>VariableType</i> , i.e. <i>Variables</i> of this type can exist.
<b>References</b>			
HasProperty	0..*		<i>HasProperty References</i> are used to identify the <i>Properties</i> of the <i>VariableType</i> . The referenced <i>Nodes</i> may be instantiated by the instances of this type, depending on the <i>ModellingRules</i> defined in 6.4.4.
HasComponent	0..*		<i>HasComponent References</i> are used for complex <i>VariableTypes</i> to identify their containing <i>DataVariables</i> . Complex <i>VariableTypes</i> can only be used for <i>DataVariables</i> . The referenced <i>Nodes</i> may be instantiated by the instances of this type, depending on the <i>ModellingRules</i> defined in 6.4.4.
HasSubtype	0..*		<i>HasSubtype References</i> identify <i>VariableTypes</i> that are subtypes of this type. The inverse <i>subtype of 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>VariableTypes</i> may contain other <i>References</i> that can be instantiated by <i>Variables</i> defined by this <i>VariableType</i> . <i>ModellingRules</i> are defined in 6.4.4.
<b>Standard Properties</b>			
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 except for the <i>DataType Attribute</i> 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. Although the relationship of a <i>VariableType</i> to its <i>DataType</i> is not modelled using <i>References</i> , changes to the <i>DataType Attribute</i> of a <i>VariableType</i>

			lead to an update of the <i>NodeVersion</i> 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 multi-part specification may define additional *Properties* for *VariableTypes*. Part 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 Part 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 Part 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
<b>Attributes</b>			
Base NodeClass Attributes	M	--	Inherited from the <i>Base NodeClass</i> . See 5.2
Executable	M	Boolean	The <i>Executable Attribute</i> indicates if the <i>Method</i> is currently executable ("False" means not executable, "True" means executable). The <i>Executable Attribute</i> does not take any user access rights into account, i.e. although the <i>Method</i> is executable this may be restricted to a certain user / user group.
UserExecutable	M	Boolean	The <i>UserExecutable Attribute</i> indicates if the <i>Method</i> is currently executable taking user access rights into account ("False" means not executable, "True" means executable).
<b>References</b>			
HasProperty	0..*		<i>HasProperty References</i> identify the <i>Properties</i> for the <i>Method</i> .
HasModellingRule	0..1		<i>Methods</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> ).
HasModelParent	0..1		The <i>HasModelParent Reference</i> points to the <i>ModelParent</i> of the <i>Method</i> (see 6.6 for details on <i>ModelParents</i> ).
GeneratesEvent	0..*		<i>GeneratesEvent References</i> identify the type of <i>Events</i> that may be generated whenever the <i>Method</i> is called.
AlwaysGeneratesEvent	0..*		<i>AlwaysGeneratesEvent References</i> identify the type of <i>Events</i> that shall be generated whenever the <i>Method</i> is called.
<other References>	0..*		<i>Methods</i> may contain other <i>References</i> .
<b>Standard Properties</b>			
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.
InputArguments	O	Argument[]	The <i>InputArguments Property</i> is used to specify the arguments that shall be used by a client when calling the <i>Method</i> .
OutputArguments	O	Argument[]	The <i>OutputArguments Property</i> specifies the result returned from the <i>Method</i> 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 Part 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.

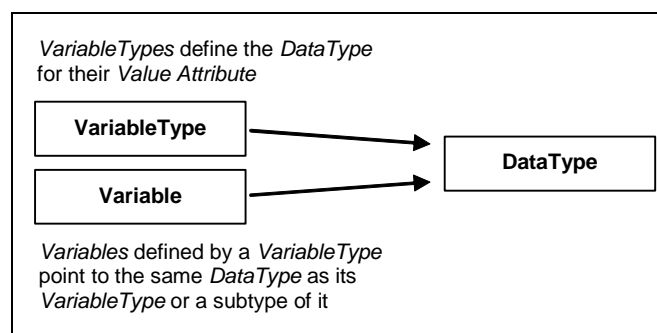
A *Method* shall always be the *TargetNode* of at least one *HasComponent Reference*. The *SourceNode* of these *HasComponent References* shall be an *Object* or an *ObjectType*. If a *Method* is called, the *NodeId* of one of those *Nodes* shall be put into the Call Service defined in Part 4 as parameter 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.

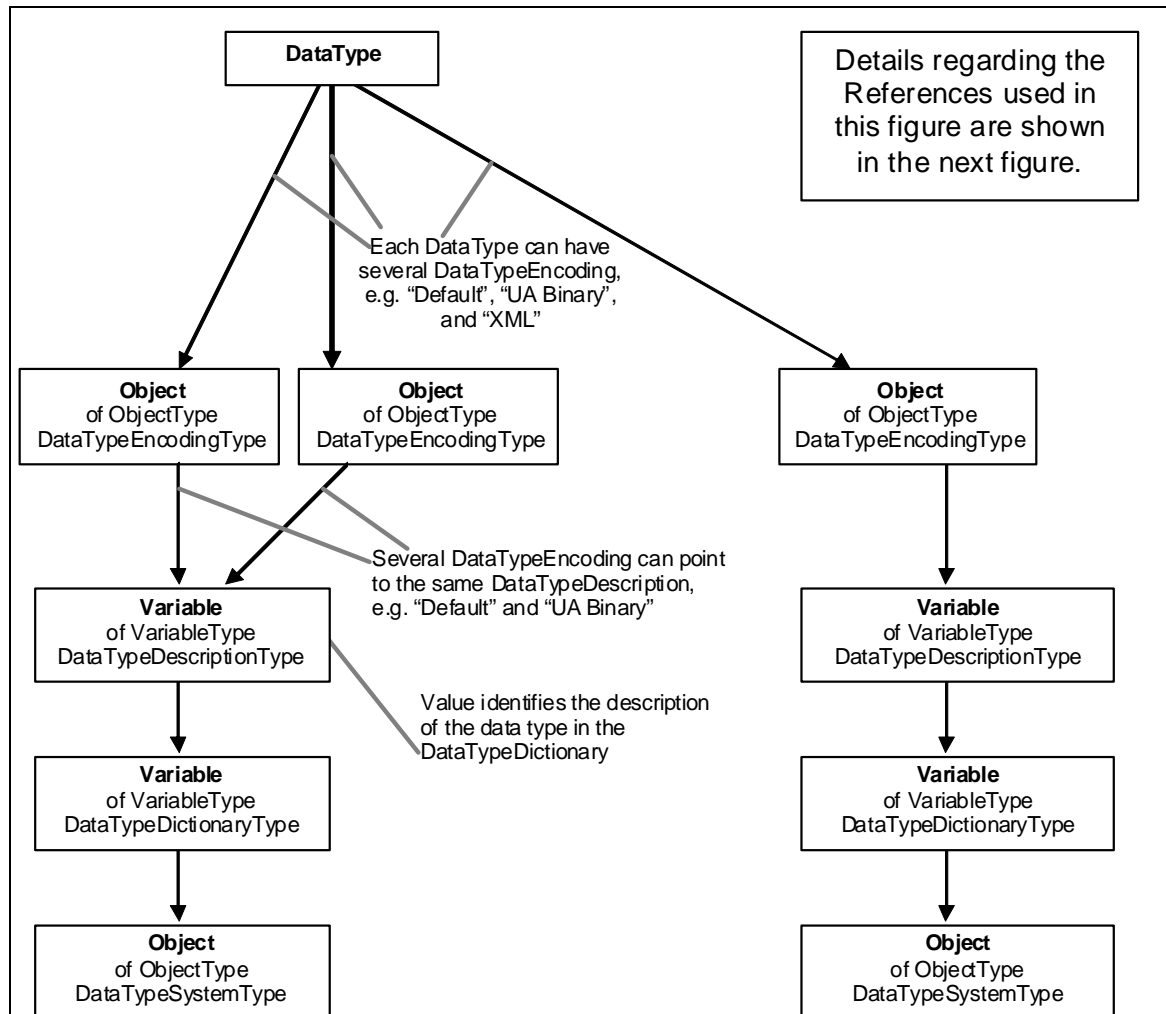


**Figure 9 – Variables, VariableTypes and their DataTypes**

In many cases, the *NodeId* of the *DataType Node* – the *DataTypeId* – will be well-known to clients and servers. Clause 8 defines *DataTypes* and Part 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 Part 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 Part 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 Part 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 Part 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 Part 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
<b>Attributes</b>			
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>DataType</i> . FALSE it is not an abstract <i>DataType</i> .
<b>References</b>			
HasProperty	0..*		<i>HasProperty References</i> identify the <i>Properties</i> for the <i>DataType</i> .
HasSubtype	0..*		<i>HasSubtype References</i> may be used to span a data type hierarchy.
HasEncoding	0..*		<i>HasEncoding References</i> identify the encodings of the <i>DataType</i> represented as <i>Objects</i> of type <i>DataTypeEncodingType</i> . Only concrete <i>Structured DataTypes</i> may use <i>HasEncoding References</i> . Abstract, <i>Built-in</i> , <i>Enumeration</i> , and <i>Simple DataTypes</i> are not allowed to be the <i>SourceNode</i> of a <i>HasEncoding Reference</i> . Each concrete <i>Structured DataType</i> shall point to at least one <i>DataTypeEncoding Object</i> with the <i>BrowseName</i> "Default Binary" or "Default XML" having the <i>NamespaceIndex</i> 0. The <i>BrowseName</i> of the <i>DataTypeEncoding Objects</i> shall be unique in the context of a <i>DataType</i> , i.e. a <i>DataType</i> shall not point to two <i>DataTypeEncodings</i> having the same <i>BrowseName</i> .
<b>Standard Properties</b>			
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.
EnumStrings	O	LocalizedText[]	The <i>EnumStrings Property</i> only applies for <i>Enumeration DataTypes</i> . It shall not be applied for other <i>DataTypes</i> . Each entry of the array of <i>LocalizedText</i> in this <i>Property</i> 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 document. Additional parts of this multi-part specification 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 Part 5.

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

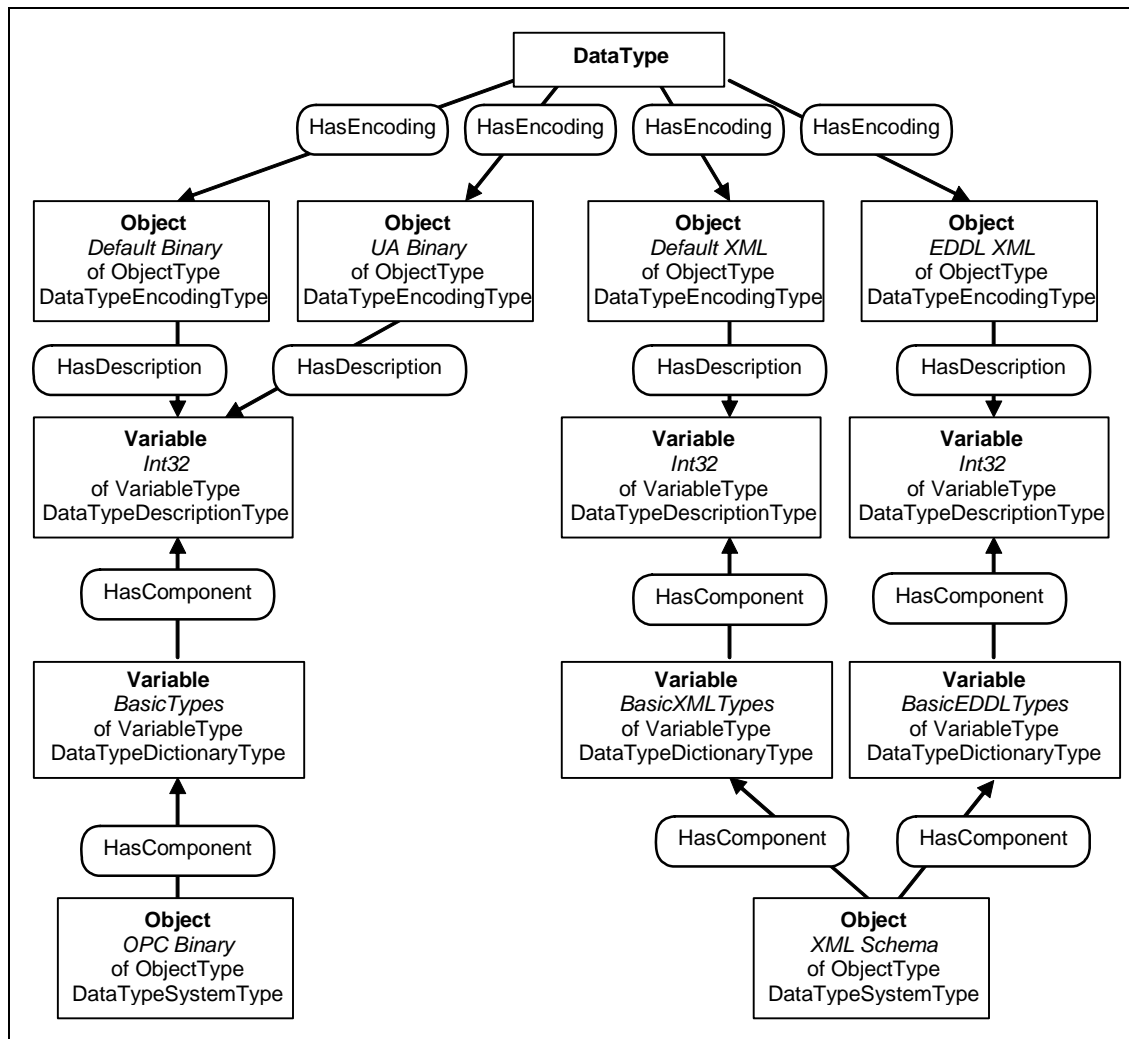


Figure 11 – Example of DataType Modelling

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 specification 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 clients can access without affecting the overall system performance.

However, servers should provide the whole *DataTypeDictionary* at once and 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	Data Type	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:

1. The *DataType Attribute* can only be changed to a new *DataType* if the new *DataType* is a subtype of the *DataType* originally used.
2. The *ValueRank Attribute* may only be further restricted
  - a. 'Any' may be set to any other value;
  - b. 'ScalarOrOneDimension' may be set to 'Scalar' or 'OneDimension';
  - c. 'OneOrMoreDimensions' may be set to a concrete number of dimensions (value > 0).
  - d. All other values of this *Attribute* shall not be changed.
3. 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 sub-types 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 sections 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 *InstanceDeclarationHierarchy* 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*.

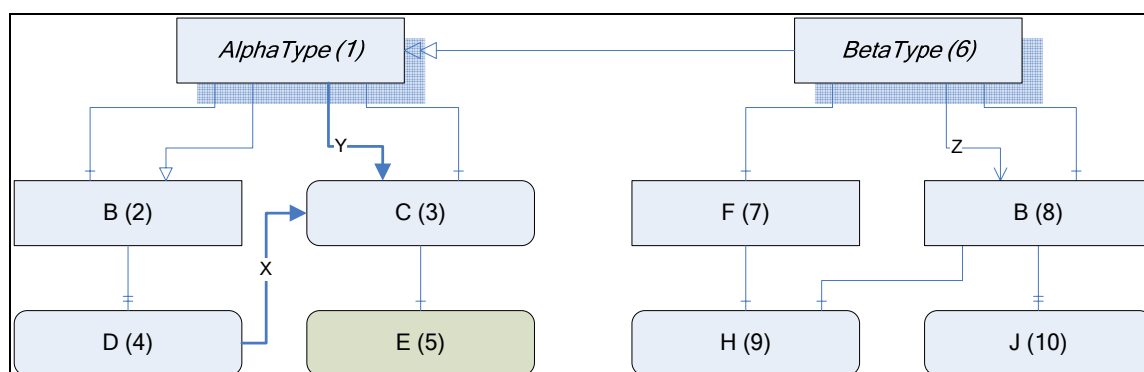


Figure 12 – Subtyping TypeDefinitionNodes

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**

		<b>BrowsePath</b>	<b>NodeId</b>	
		/	6	
		/F	7	
		/B	8	
		/F/H	9	
		/B/J	10	
		/B/H	9	
		<b>Source Path</b>	<b>ReferenceType</b>	<b>Target Path</b>
		/	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	-

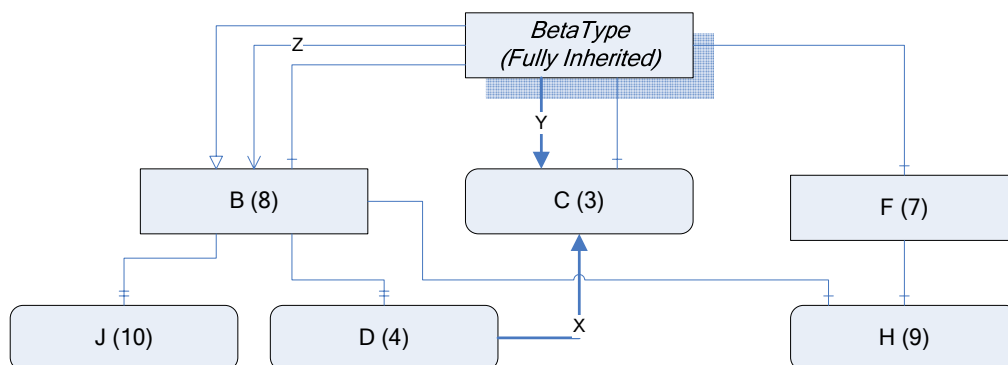
/	HasTypeDefinition	-	BetaType
/F	HasTypeDefinition	-	BaseObjectType
/B	HasTypeDefinition	-	BaseObjectType
/F/H	HasTypeDefinition	-	PropertyType
/B/J	HasTypeDefinition	-	PropertyType
/B	HasComponent	/B/H	-

/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 regards 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 Part 4.

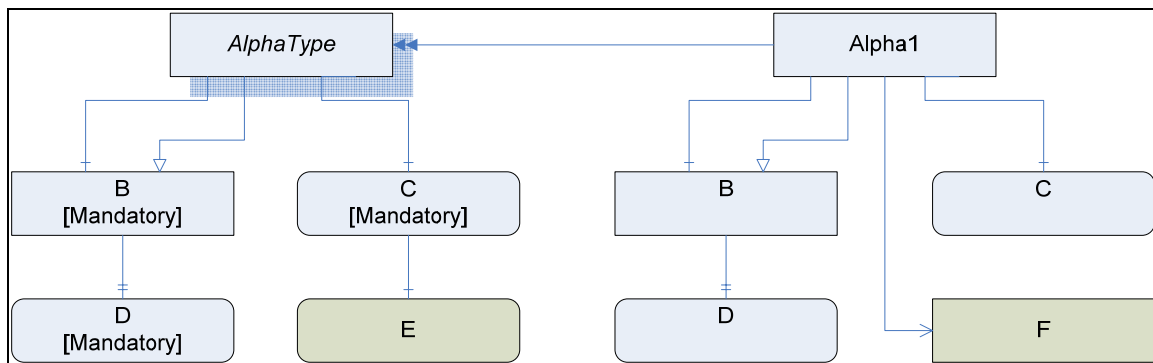
### 6.4.2 Creating an Instance

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 new

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

### 6.4.3 Constraints on an Instance

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

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

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

If two *InstanceDeclarations* of the fully-inherited *InstanceDeclarationHierarchy* have been connected directly with several *References*, all those *References* shall connect the same *Nodes*. An example is given in Figure 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.

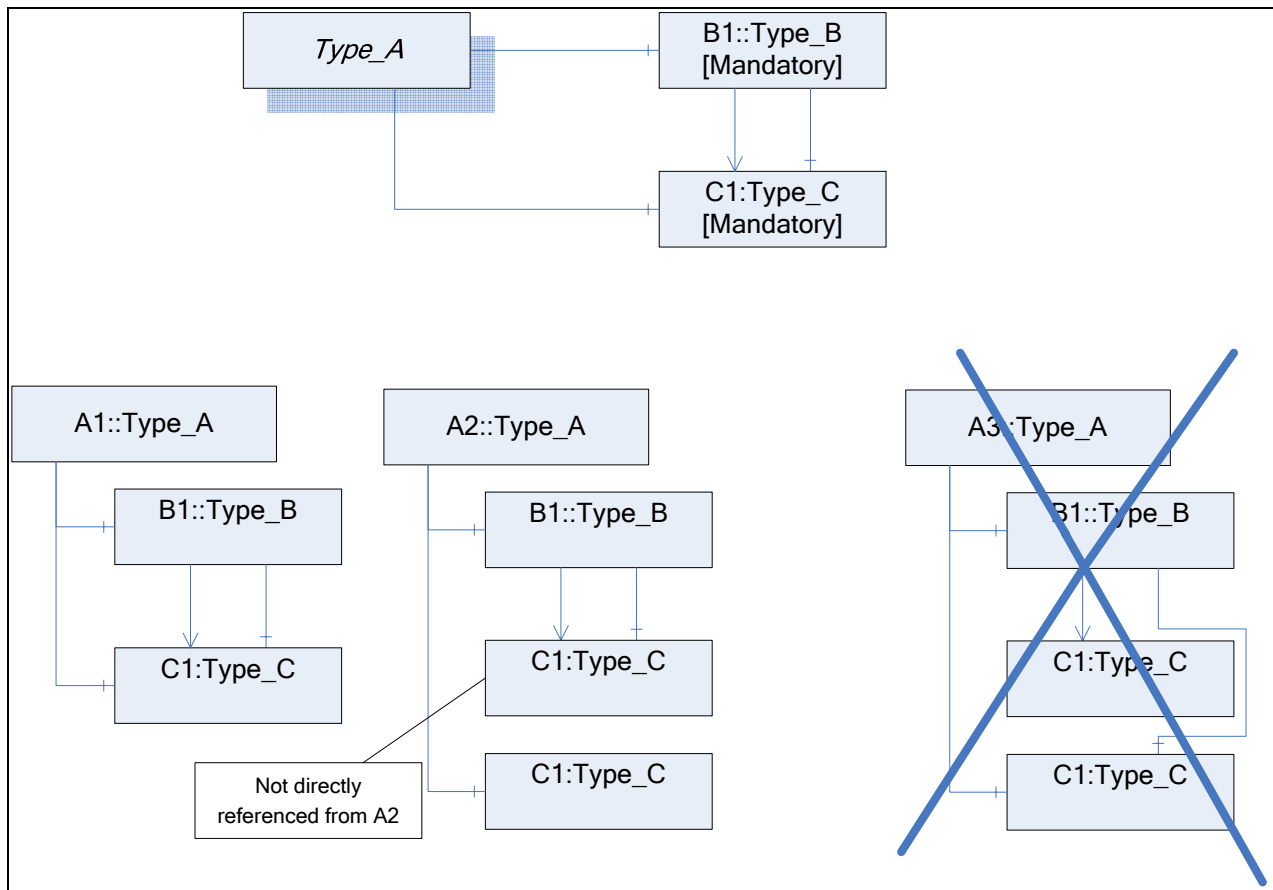


Figure 15 – Example for several References between InstanceDeclarations

## 6.4.4 ModellingRules

### 6.4.4.1 General

This specification defines *ModellingRules*. 6.4.4.5 defines *ModellingRules* in this document. Other parts of this multi-part specification 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 specification 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 the specification only specifies one *Property* for *ModellingRules*. Future versions of the specification may define additional *Properties* for *ModellingRules*. Part 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.

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 Part 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 Part 4) can typically not be used to access instances based on those *InstanceDeclarations*.

#### 6.4.4.3 Subtyping Rules for Properties of ModellingRules

It is allowed that subtypes override *ModellingRules* on their *InstanceDeclarations*. As a general rule for subtyping, constraints shall only be tightened, not loosened. Therefore, it is not allowed to specify on the supertype that an instance shall exist with the name (*NamingRule Mandatory*) and on the subtype make this optional (*NamingRule Optional*). Table 15 specifies the allowed changes on the *Properties* when exchanging the *ModellingRules* in the subtype.

**Table 15 – Rule for ModellingRules Properties when Subtyping**

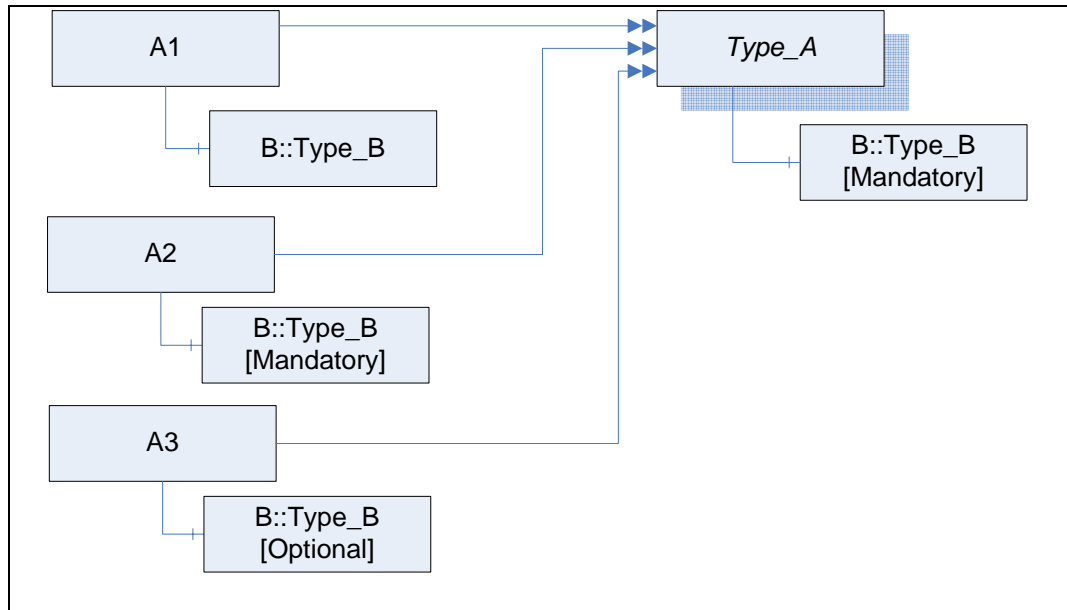
	Value on supertype	Value on subtype
NamingRule	Mandatory	Mandatory
NamingRule	Optional	Mandatory or Optional
NamingRule	Constraint	Constraint

#### 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*.

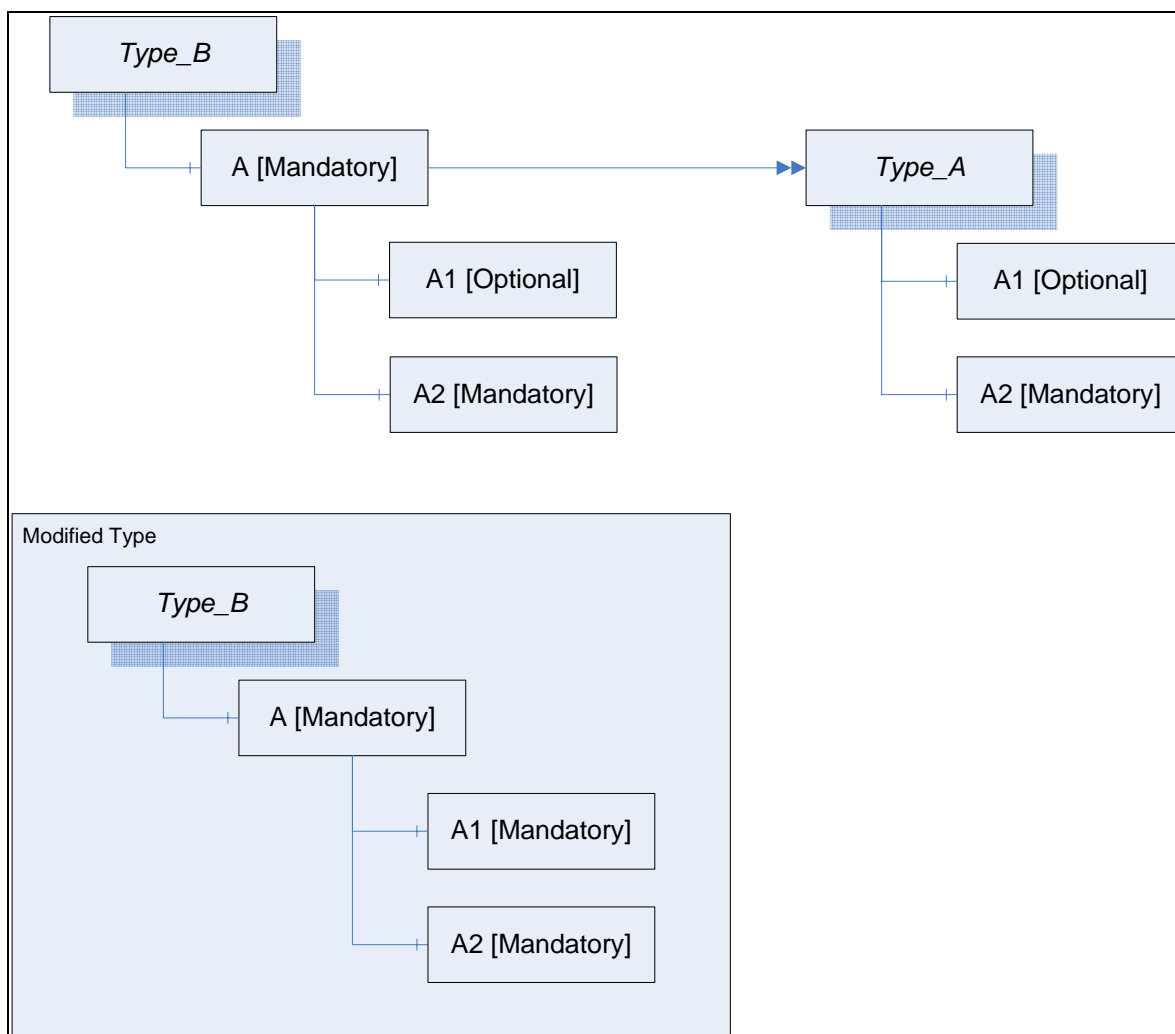
In Figure 16 an example is given. The instances A1, A2, and A3 are all valid instances of Type\_A, although B of A1 has no *ModellingRule* and B of A3 a different *ModellingRule* than B of Type\_A.



**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.

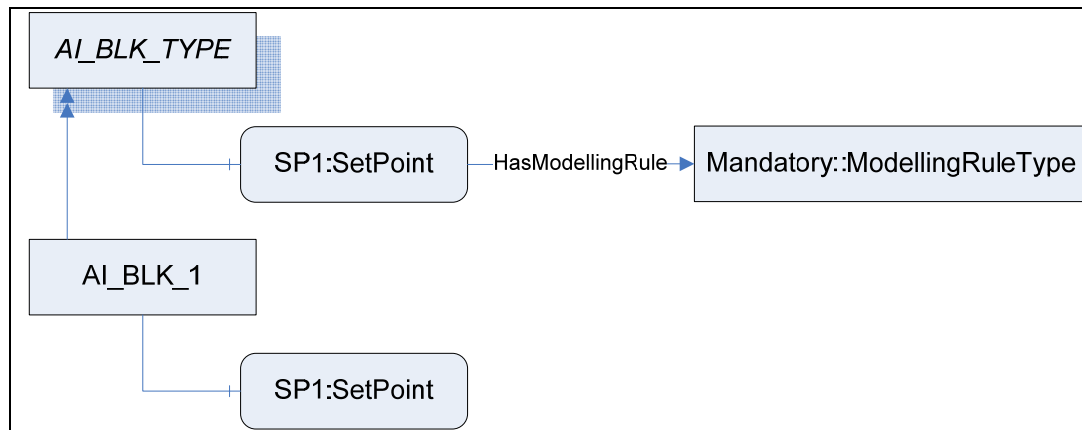
**Table 16 – Properties of ModellingRules**

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.



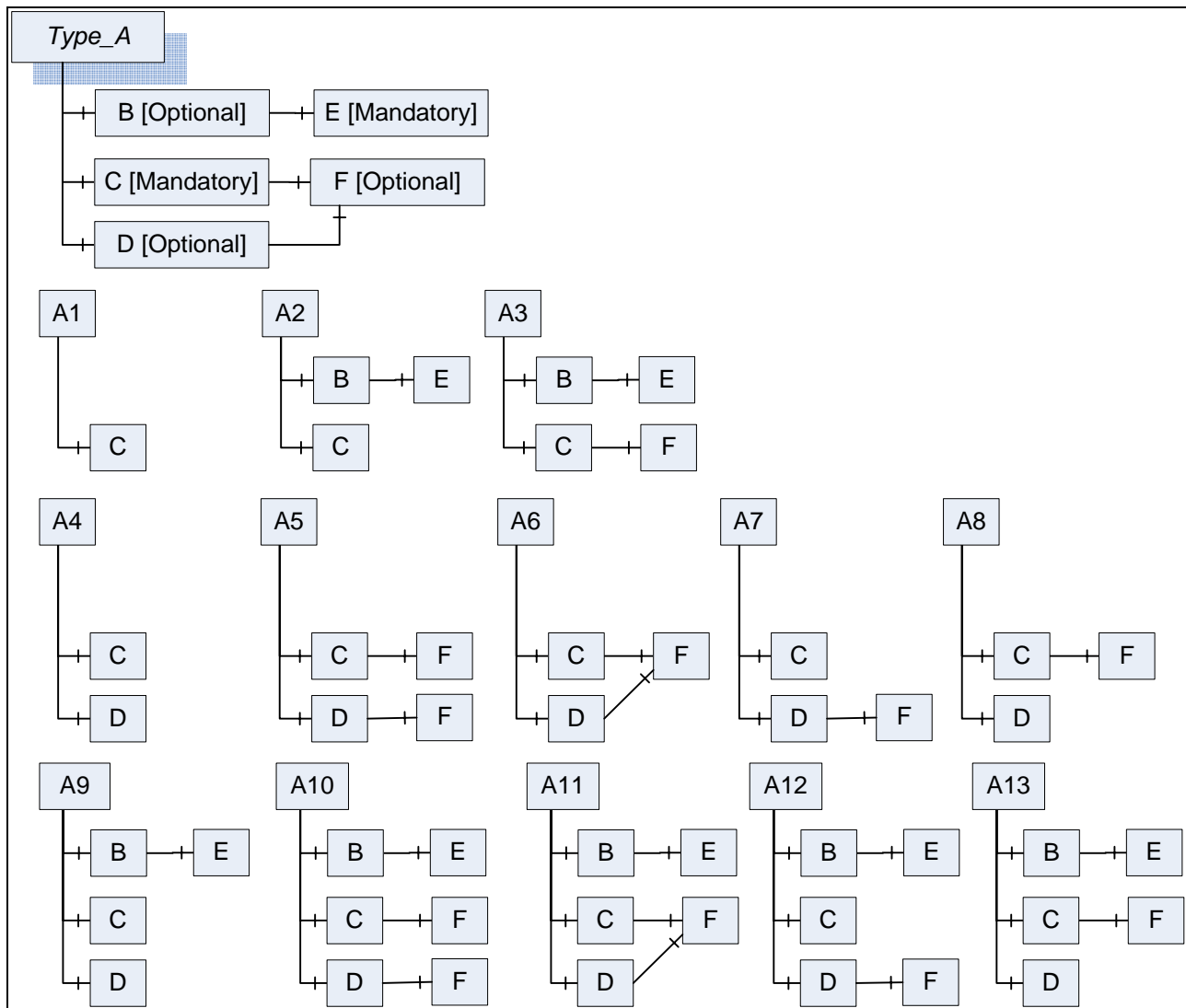


Figure 19 – Example using the Standard ModellingRules Optional and Mandatory

#### 6.4.4.5.4 ExposesItsArray

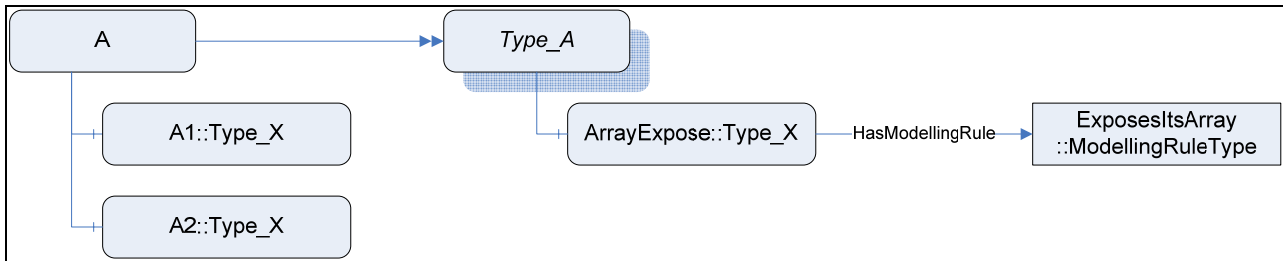
The *ExposesItsArray ModellingRule* exposes a special semantic on *VariableTypes* having a single- or multidimensional array as data type. It indicates that each value of the array will also be exposed as a *Variable* in the *AddressSpace*.

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 then 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 then 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

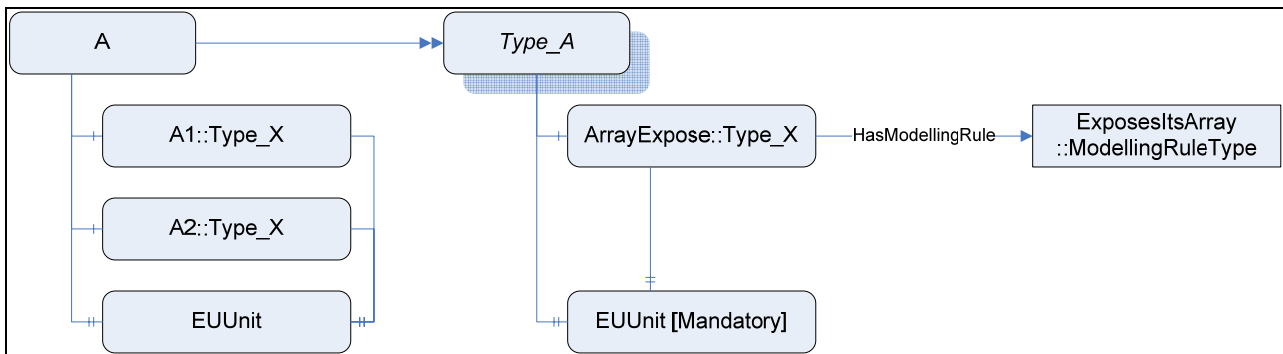
typically either get the array or access the *Variables* directly, so there is no need providing that information.



**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

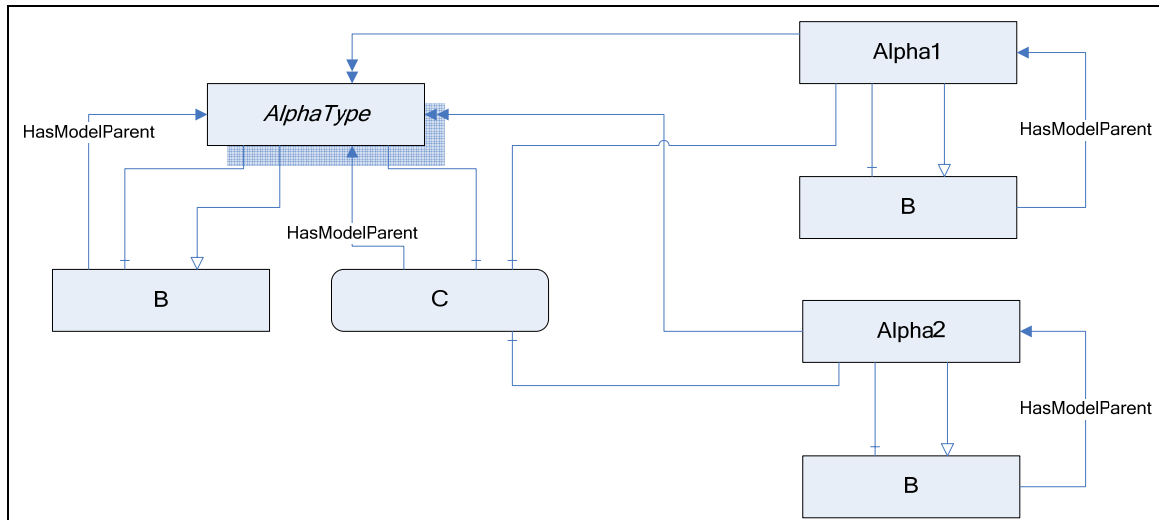
Each *Object*, *Variable* and *Method* may be referenced by several *hierarchical References*. To indicate the scope of such a *Node* 'A' it can expose its *ModelParent*. The *ModelParent* of 'A' is the scope where 'A' is defined. When a client intends to change 'A' it can use the *ModelParent* to determine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

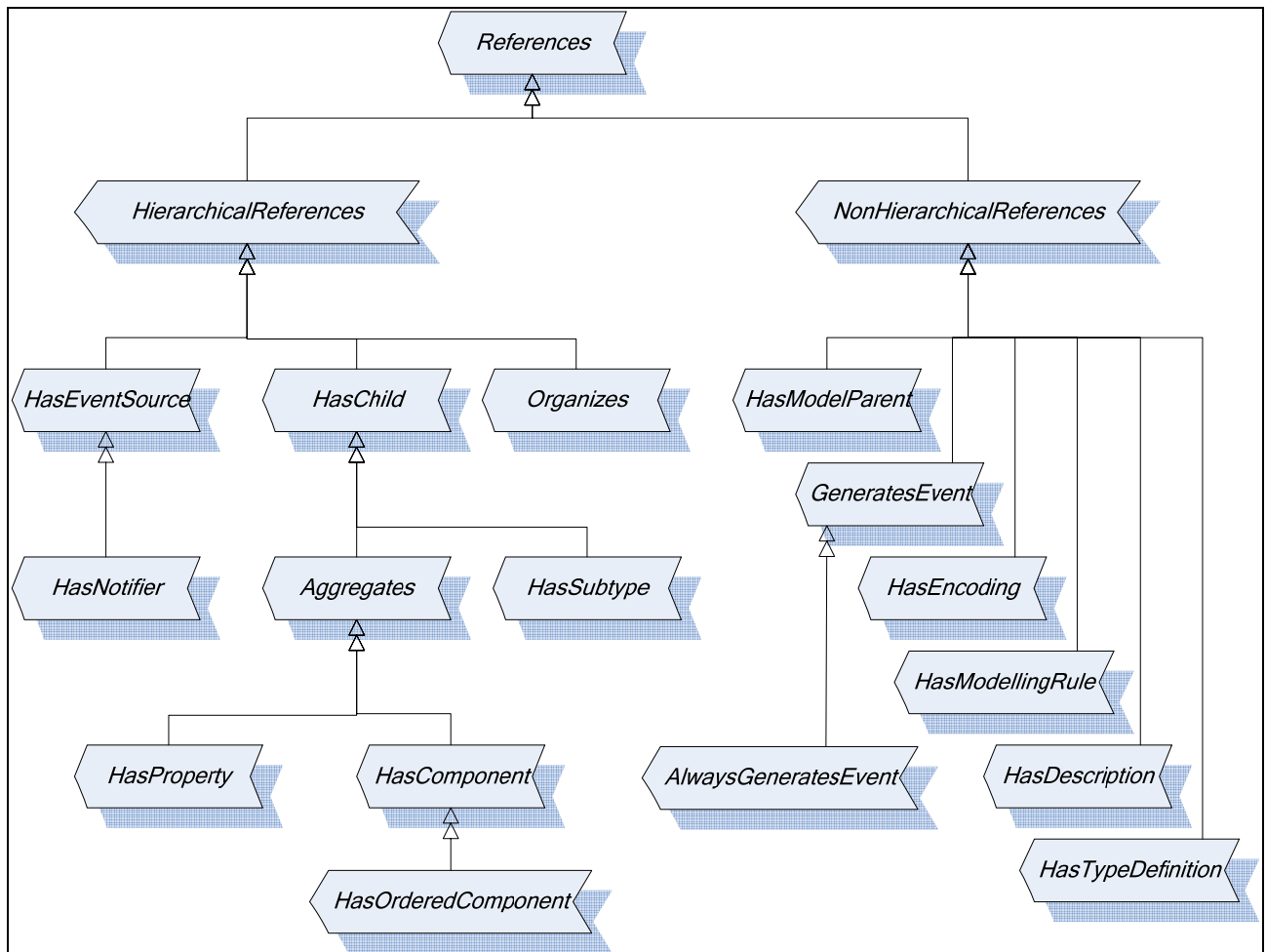


**Figure 22 – Example on ModelParents**

## 7 Standard ReferenceTypes

### 7.1 General

This specification 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 multi-part specification may specify additional *ReferenceTypes*. The following subclauses define the *ReferenceTypes*. Part 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 Part 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, i.e. 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 Part 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*, i.e. 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*, i.e. 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 Part 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*.

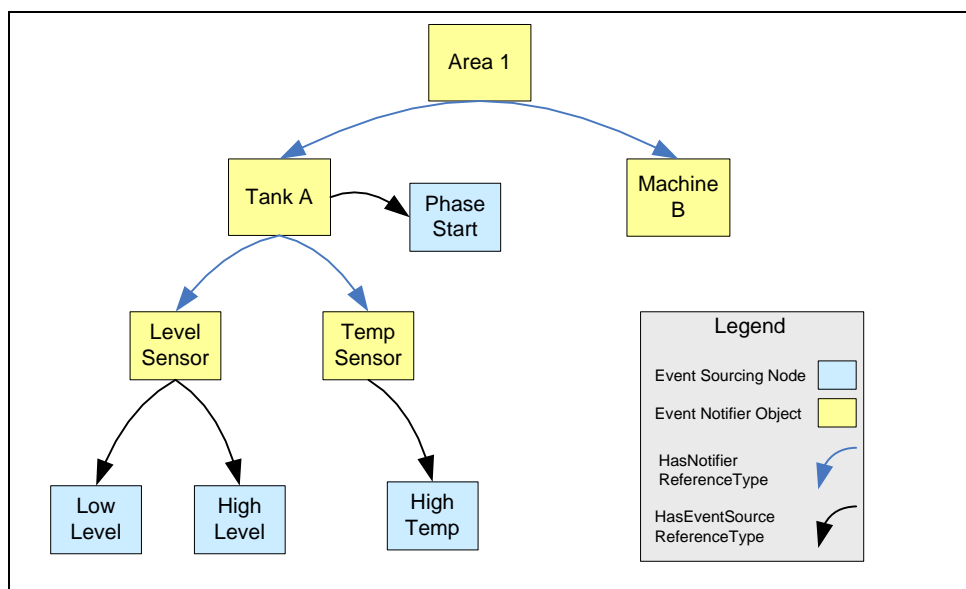
The semantic of this *ReferenceType* is to relate *Object Nodes* that are notifiers with other notifier *Object Nodes*. The *ReferenceType* is used to establish a hierarchical organization of event notifying *Objects*. It is a subtype of the *HasEventSource ReferenceType* defined in 7.19.



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”.

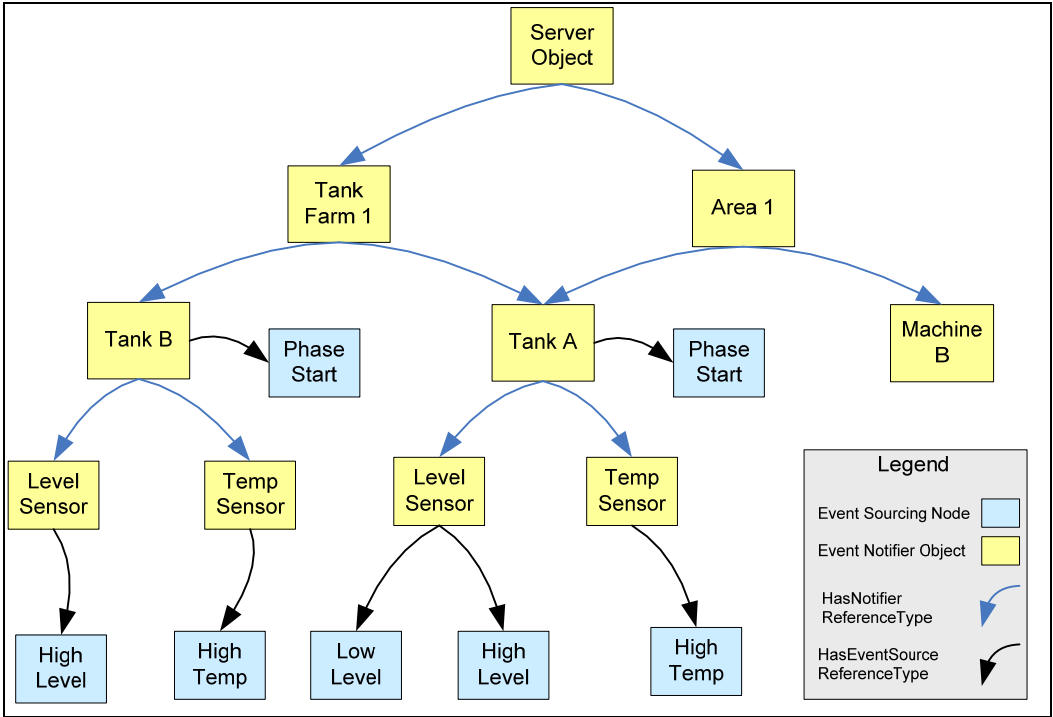


Figure 25 – Complex Event Reference Example

## 8 Standard DataTypes

### 8.1 General

The following subclauses define *DataTypes*. Their representation in the *AddressSpace* and the *DataType* hierarchy is specified in Part 5. Other parts of this multi-part specification 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 IdType	The format and data type of the identifier (see 8.2.3).
identifier	*	The identifier for a <i>Node</i> in the <i>AddressSpace</i> of an OPC UA server (see 8.2.4).

See Part 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 Part 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 – IdType 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 Part 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. 00000000-0000-0000-0000-00000000)
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 <i>NamespaceArray</i> . The client may read the <i>NamespaceArray Variable</i> to access the string value of the namespace.
name	String	The text portion of the <i>QualifiedName</i> .

## 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 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 RFC 3066 are restricted as follows:

- This specification permits only zero or one <country/region> component to follow the <language> component,
- This specification also permits the “-CHS” and “-CHT” three-letter <country/region> codes for “Simplified” and “Traditional” Chinese locales.
- 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 <i>NodeId</i> of the <i>DataType</i> of this argument
valueRank	Int32	Indicates whether the <i>dataType</i> is an array and how many dimensions the array has. It may have the following values: n>1: the <i>dataType</i> is an array with the specified number of dimensions. OneDimension (1): The <i>dataType</i> is an array with one dimension. OneOrMoreDimensions (0): The <i>dataType</i> is an array with one or more dimensions. Scalar (-1): The <i>dataType</i> is not an array. Any (-2): The <i>dataType</i> can be a scalar or an array with any number of dimensions. ScalarOrOneDimension (-3): The <i>dataType</i> can be a scalar or a one dimensional array.
arrayDimensions	UInt32[]	Specifies the length of each dimension for an array <i>dataType</i> . It is intended to describe the capability of the <i>dataType</i> , not the current size. The number of elements shall be equal to the value of the <i>valueRank</i> . Must be null if <i>valueRank</i> <= 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 Part 6.

## 8.12 Double

This *Built-in DataType* defines a value that adheres to the IEEE 754 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 Part 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 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 Part 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 IS 10918-1.

### 8.23 ImagePNG

This *Simple DataType* defines a *ByteString* representing an image in PNG format. PNG is defined in ISO/IEC 15948:2003.

### 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 <i>offset</i> includes the DST correction. If FALSE then the <i>offset</i> 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 Part 6. All *Structured DataTypes* have to inherit from this *DataType* if they are not defined as primitives in this Part (like *NodeId* defined in 8.2, a *NodeId* is structured but treated in a special way as defined in Part 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. Part 6 defines details about this *DataType*.

### 8.39 XmlElement

This *Built-in DataType* is used to define XML elements. Part 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 Part 5. Other parts of this multi-part specification may specify additional *EventTypes*. Figure 26 informally describes the hierarchy of these *EventTypes*.

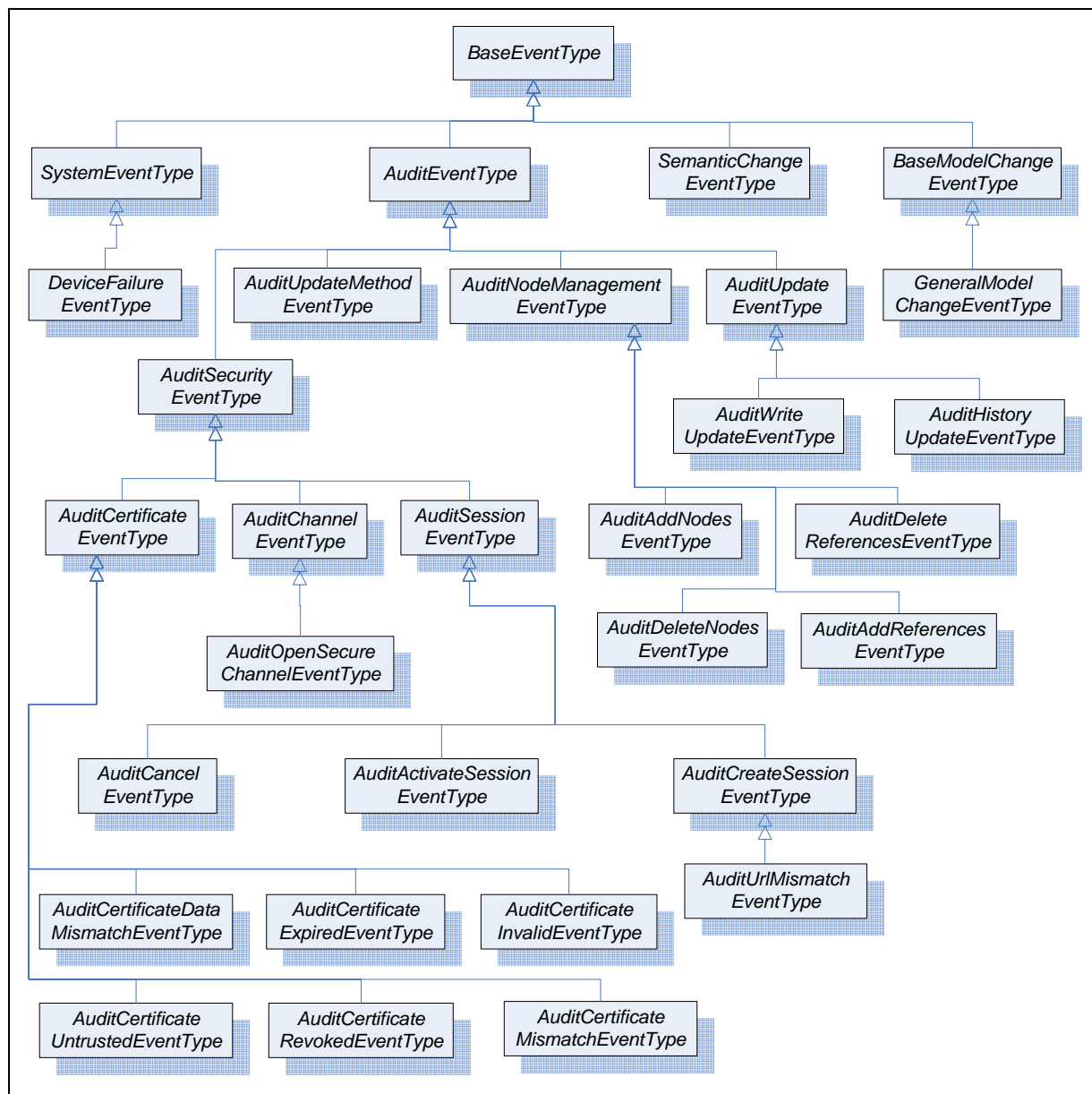


Figure 26 – Standard EventType Hierarchy

### 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.

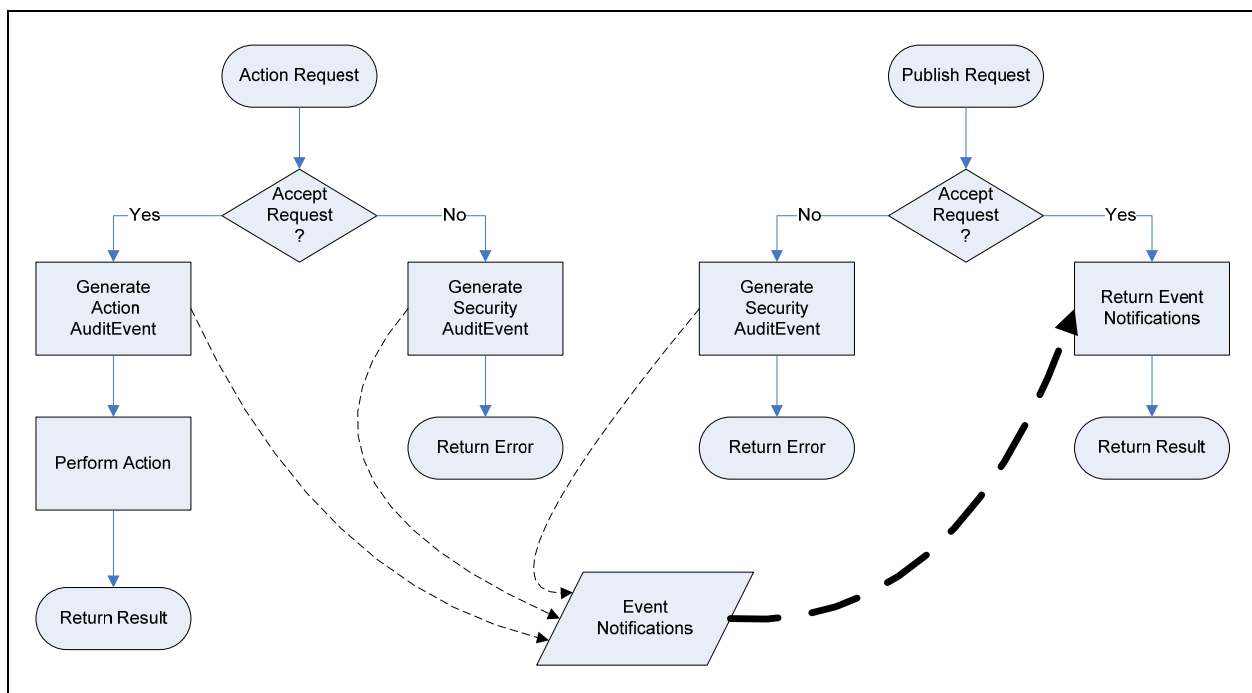
## 9.4 AuditEventType

*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.

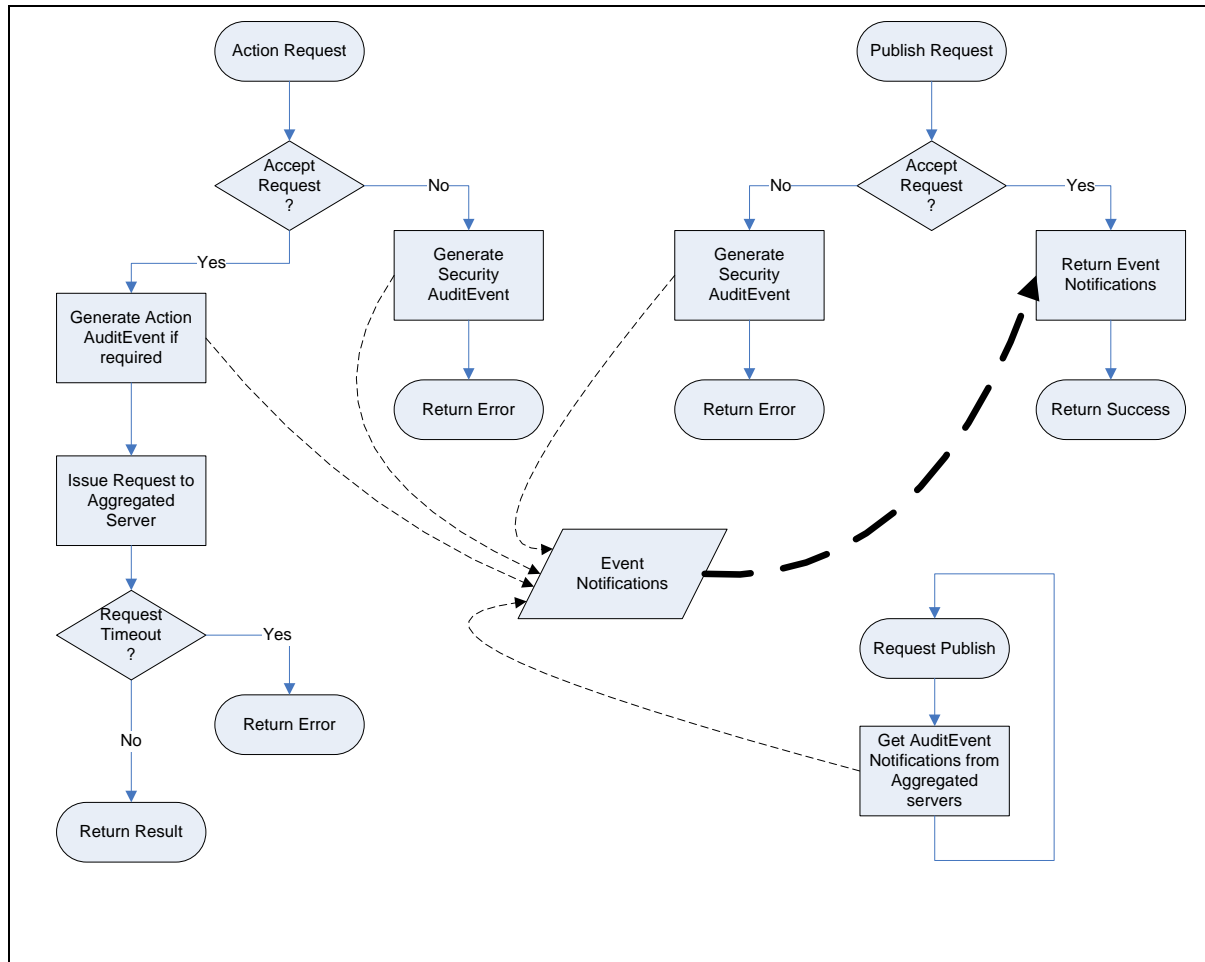


**Figure 27 – Audit Behaviour of a Server**

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 do to security on the aggregating server.



**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 Part 4.

### 9.7 AuditOpenSecureChannelEventType

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

### 9.9 AuditCreateSessionEventType

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

### 9.10 AuditUrlMismatchEventType

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

### 9.12 AuditCancelEventType

This is a subtype of AuditSessionEventType and is used for *Events* generated from calling the Cancel *Service* defined in Part 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 Part 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 i.e. 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 or 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 CA),

### 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 Part 4.

### 9.22 AuditDeleteNodesEventType

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

### 9.23 AuditAddReferencesEventType

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

### 9.24 AuditDeleteReferencesEventType

This is a subtype of AuditNodeManagementEventType and is used for *Events* generated from calling the DeleteReferences *Service* defined in Part 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 BaseModelChangeEvent Type

The *BaseModelChangeEvent Type* 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 GeneralModelChangeEvent Type

The *GeneralModelChangeEvent Type* is a subtype of the *BaseModelChangeEvent Type*. 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 ModelChangeEvent

Two types of *ModelChangeEvent*s 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 *ModelChangeEvent*s should respond to any *Event* of the *BaseModelChangeEvent* type including its subtypes like the *GeneralModelChangeEvent*.

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

## 9.31 SemanticChangeEvent

### 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 *ModelChangeEvent*s. This is defined in 9.30.3.

### 9.31.4 Event Compression

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



## Annex A (informative): How to use the Address Space Model

### A.1 Overview

This Appendix points out some general considerations how the Address Space Model can be used. The Appendix is informative, 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.

The Appendix of Part 5 gives an overview over the design decisions made when modelling the information about the server defined in Part 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

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 Part 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*<sup>3</sup> 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*”.

---

<sup>3</sup> *VariableTypes* other than the *PropertyType* which is used for all *Properties*

#### 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*, i.e. 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 Part 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:

- 1) 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.
- 2) Create a complex *DataType* and a simple *Variable* using this *DataType*.
- 3) 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.

The third approach combines both other approaches. Therefore the specific client can access the data in its native format in a transactional context, whereas the generic client can access the simple *DataTypes* of the components of the complex *Variable*. The disadvantage is that the server shall 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, e.g. 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*, i.e. *Methods* that cannot directly be assigned to a special *Object*, should be assigned to the *Server Object* defined in Part 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 29).

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 Part 5.

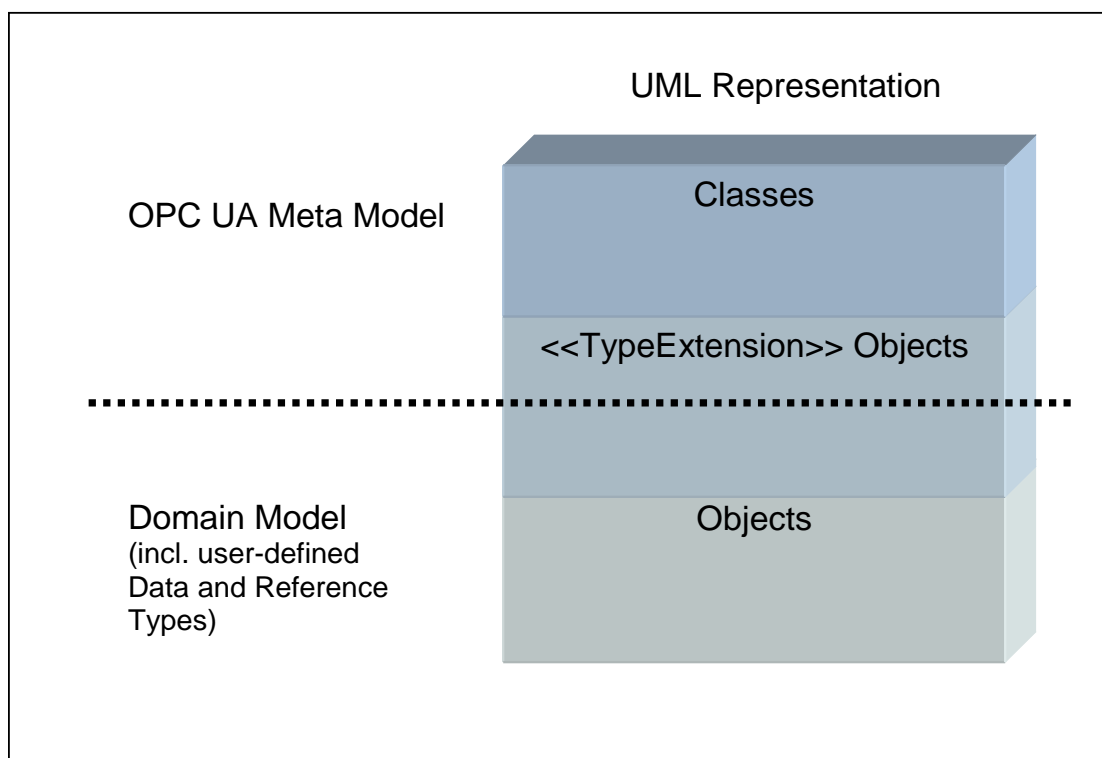


Figure 29 – Background of OPC UA Meta Model

### B.2 Notation

An example of a UML class representing the OPC UA concept *Base* is given in the UML class diagram in Figure 30. OPC Attributes inherit from the abstract class *Attribute* and have a value identifying their data type. They are composed to a *Node* either optional (0..1) or required (1), like *BrowseName* to *Base* in Figure 30.

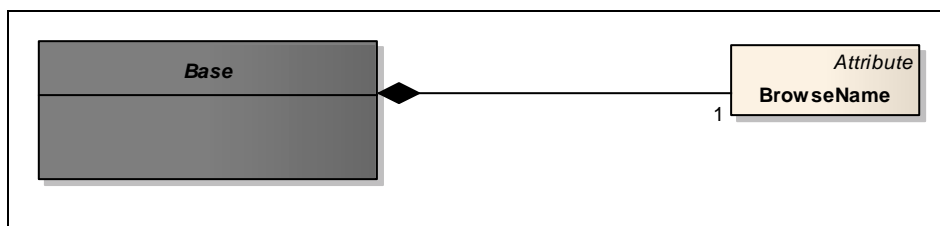
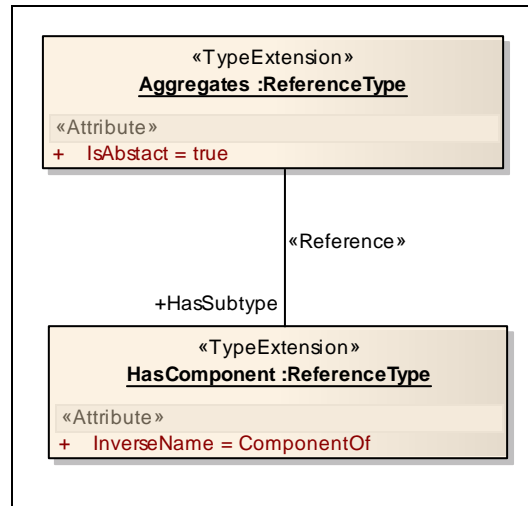


Figure 30 – Notation (I)

UML object diagrams are used to display <<TypeExtension>> objects (e.g. *HasComponent* in Figure 31). 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 31 – 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, e.g. the *BrowseName* of the UML object *HasComponent* in Figure 31 is “HasComponent”.

To highlight the classes explained in a class diagram, they are marked grey (e.g. *Base* in Figure 30). 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 multi-part specification can extend the OPC UA Meta Model by adding *Attributes* and defining new *ReferenceTypes*.

### B.3.1 Base

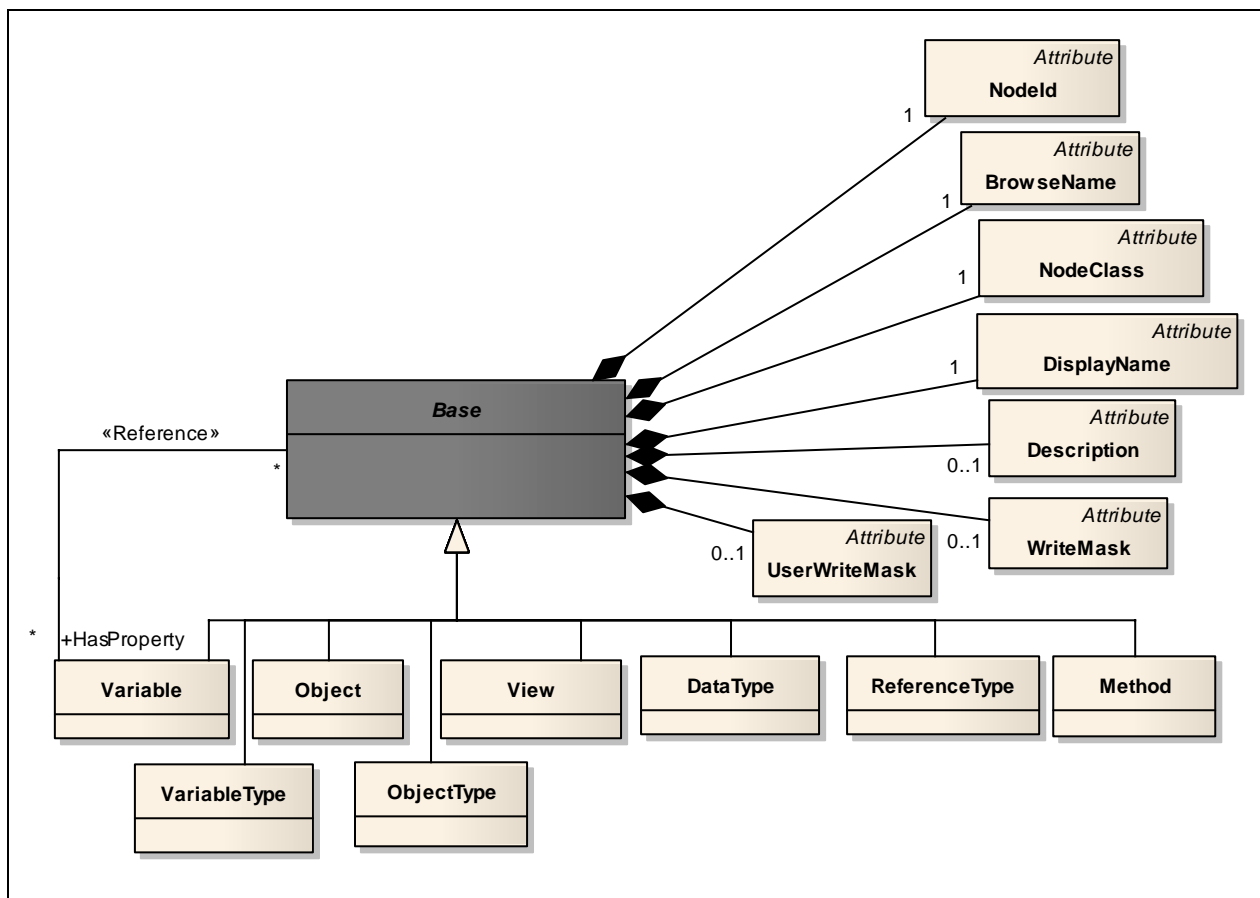


Figure 32 – Base

### B.3.2 ReferenceType

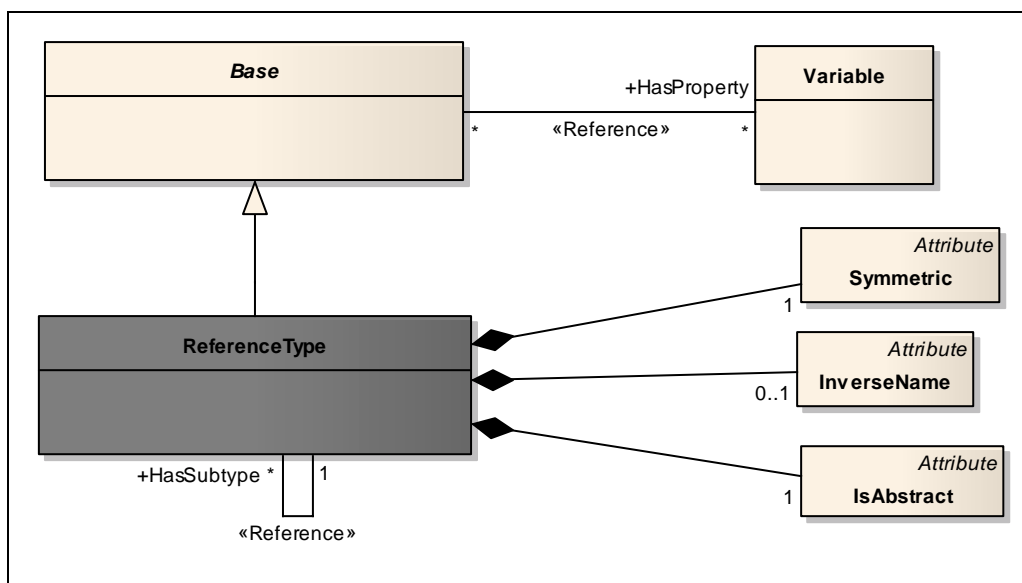


Figure 33 – Reference and ReferenceType

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

### B.3.3 Predefined ReferenceTypes

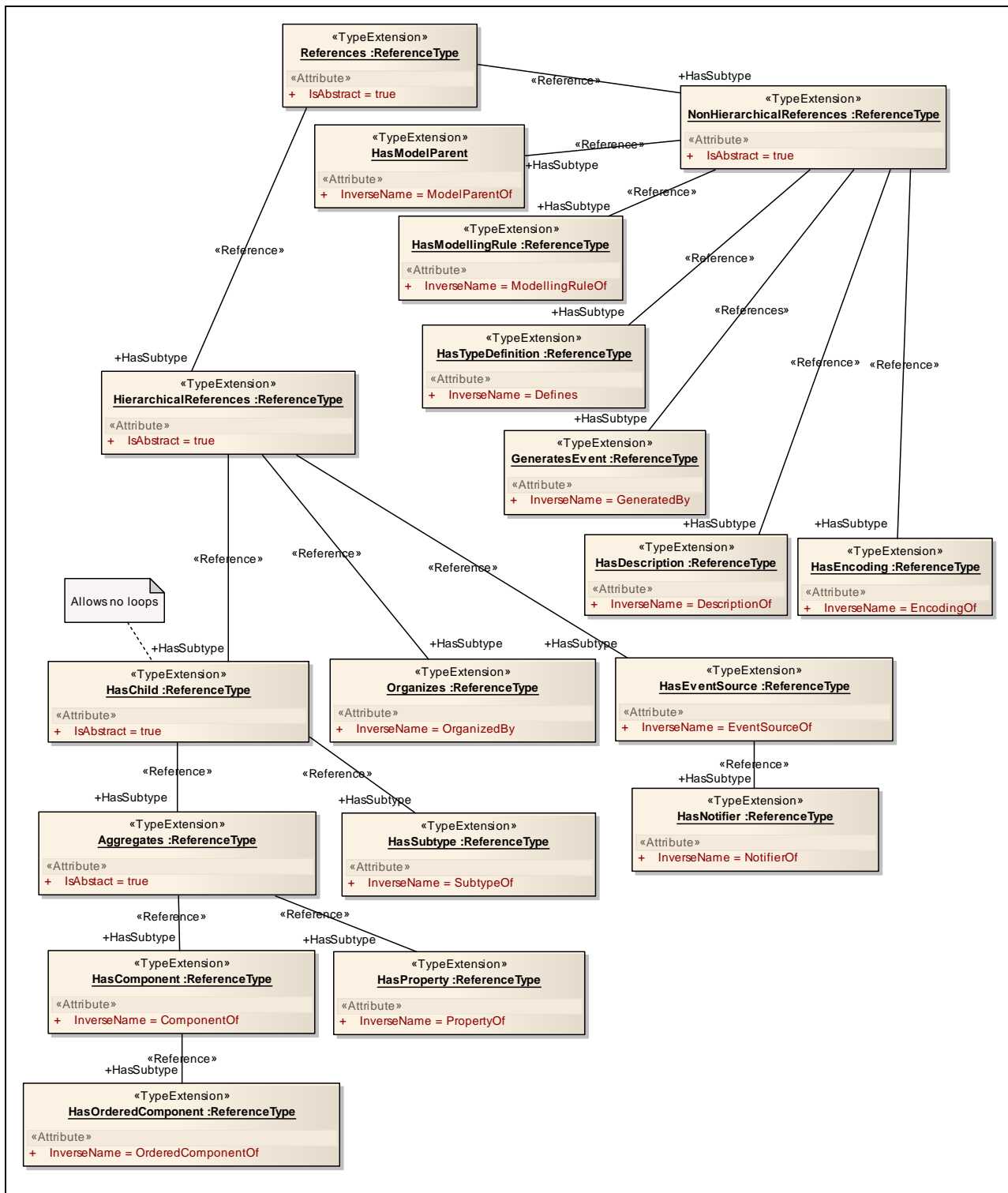


Figure 34 – Predefined ReferenceTypes

### B.3.4 Attributes

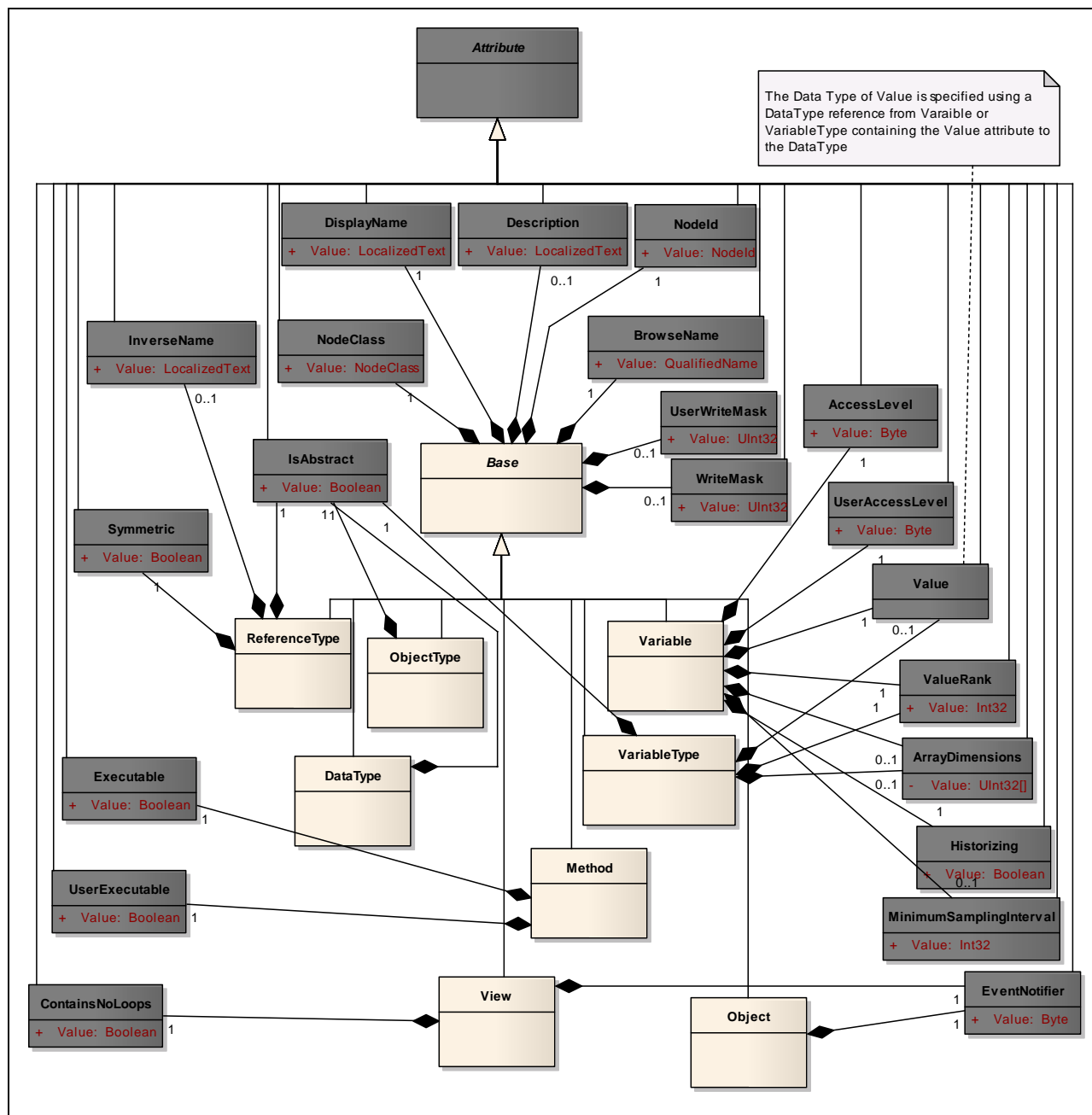


Figure 35 – Attributes

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

*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

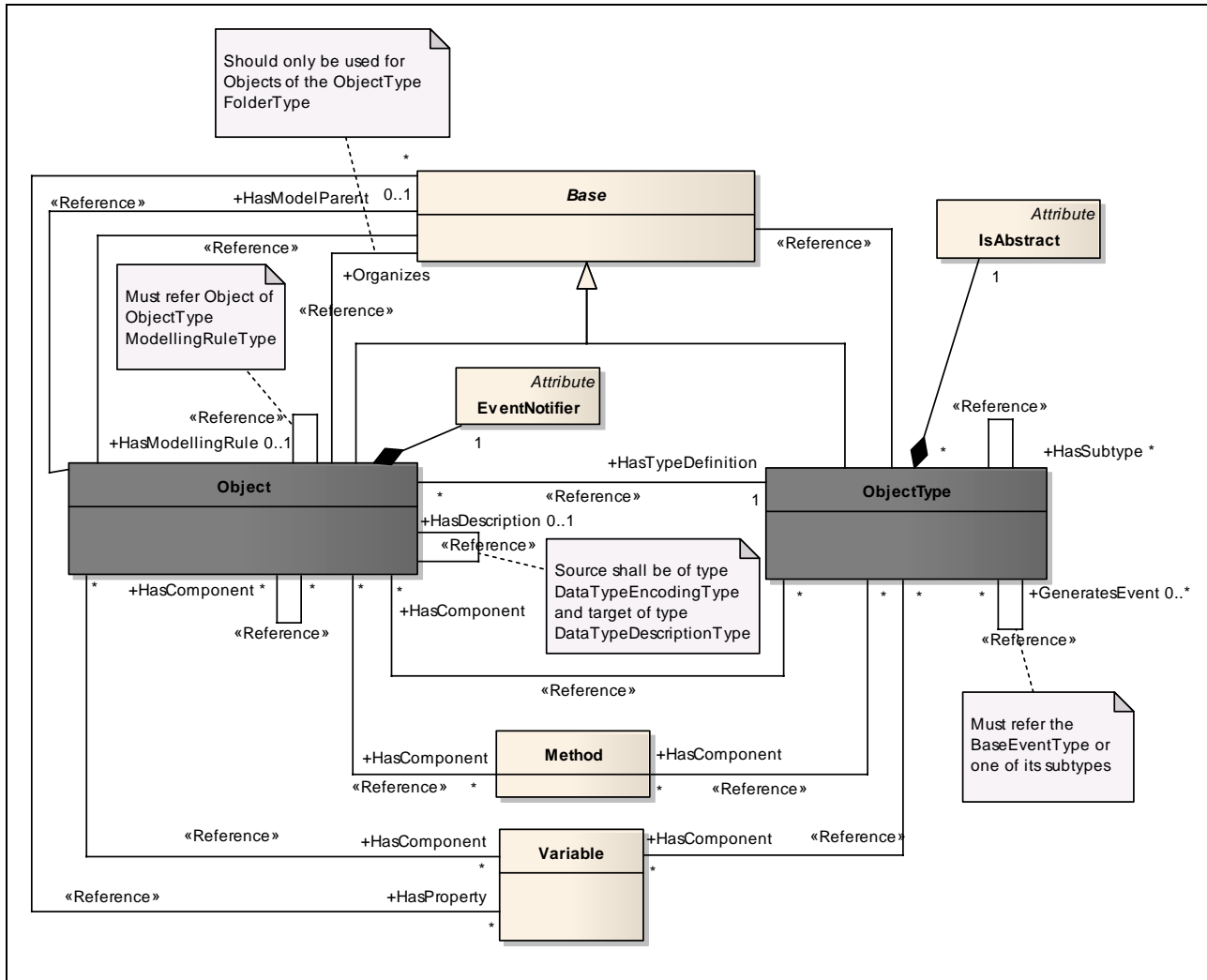


Figure 36 – Object and ObjectType

### B.3.6 EventNotifier

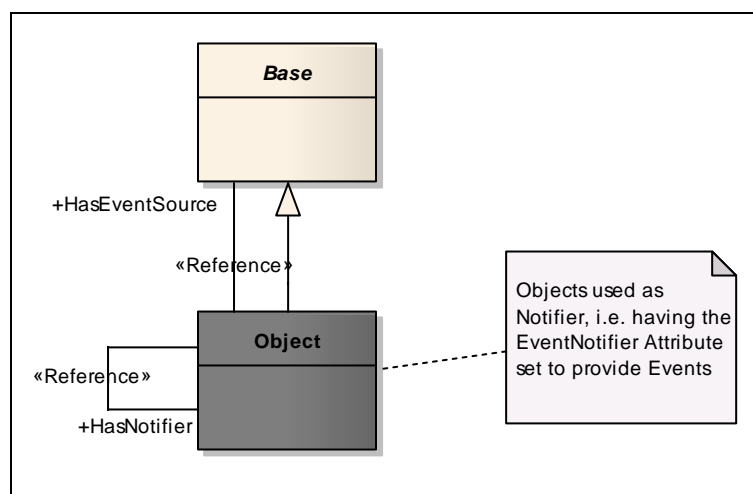


Figure 37 – EventNotifier



### B.3.8 Method

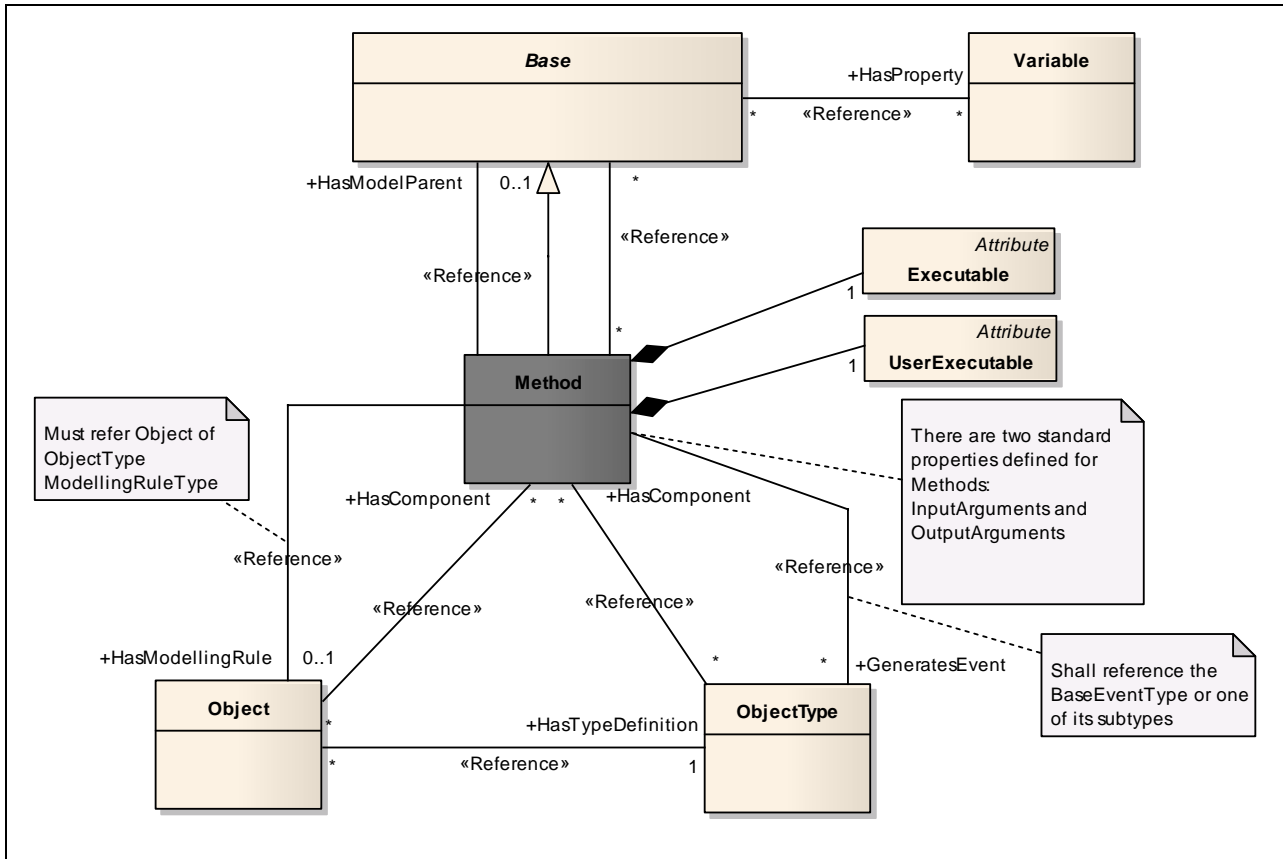


Figure 39 – Method



## Annex C (normative): OPC Binary Type Description System

### C.1 Concepts

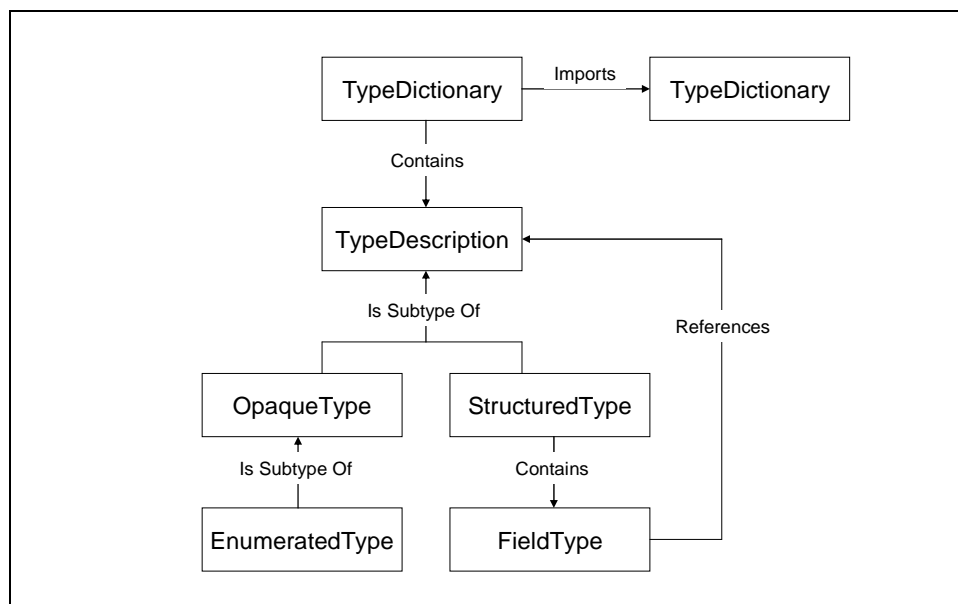
The OPC Binary XML Schema defines the format of OPC Binary *TypeDictionaries*. Each OPC Binary *TypeDictionary* is an XML document that contains one or more *TypeDescriptions* that describe the format of a binary-encoded value. Applications that have no advance knowledge of a particular binary encoding 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 means that the *TypeName* qualified with the *TargetNamespace* for the dictionary should be a globally-unique identifier for a *TypeDescription*.

Figure 42 below illustrates the structure of an OPC Binary *TypeDictionary*.



**Figure 42 – 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 27

**Table 27 – 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 <i>TypeDictionaries</i> that are referenced by <i>StructuredTypes</i> defined in the dictionary. Each import element specifies the <i>NamespaceURI</i> of the <i>TypeDictionary</i> being imported. The <i>TypeDictionary</i> element shall declare an XML namespace prefix for each imported namespace.
TargetNamespace	xs:string	Specifies the URI that qualifies all <i>TypeDescriptions</i> defined in the dictionary.
DefaultByteOrder	ByteOrder	Specifies the default <i>ByteOrder</i> for all <i>TypeDescriptions</i> that have the <i>ByteOrderSignificant</i> attribute set to "true". This value overrides the setting in any imported <i>TypeDictionary</i> . This value is overridden by the <i>DefaultByteOrder</i> specified on a <i>TypeDescription</i> .
TypeDescription	TypeDescription[]	One or more elements that describe the structure of a binary encoded value. A <i>TypeDescription</i> is an abstract type. A dictionary may only contain the <i>OpaqueType</i> , <i>EnumeratedType</i> and <i>StructuredType</i> 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 sub-types may appear in a *TypeDictionary*. The components of a *TypeDescription* are described in Table 28

**Table 28 – 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 understand what is contained in the value.
Name	xs: NCName	An attribute that specifies a name for the <i>TypeDescription</i> that is unique within the dictionary. The fields of structured types reference <i>TypeDescriptions</i> by using this name qualified with the dictionary namespace URI.
DefaultByteOrder	ByteOrder	An attribute that specifies the default <i>ByteOrder</i> for the type description. This value overrides the setting in any <i>TypeDictionary</i> or in any <i>StructuredType</i> 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 29.

**Table 29 – OpaqueType Components**

Name	Type	Description
TypeDescription	TypeDescription	An <i>OpaqueType</i> inherits all elements and attributes defined for a <i>TypeDescription</i> in Table 28.
LengthInBits	xs:string	An attribute which specifies the length of the <i>OpaqueType</i> in bits. This value should always be specified. If this value is not specified the <i>Documentation</i> 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 <i>DefaultByteOrder</i> attribute specified for containing <i>StructuredTypes</i> or the <i>TypeDictionary</i> . If <i>StructuredTypes</i> are nested the inner <i>StructuredTypes</i> override the byte order of the outer descriptions. If the <i>DefaultByteOrder</i> attribute is specified for the <i>OpaqueType</i> , then the <i>ByteOrder</i> is fixed and does not change according to context. If this attribute is “true”, then the <i>LengthInBits</i> 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 30.

**Table 30 – EnumeratedType Components**

Name	Type	Description
OpaqueType	OpaqueTypeDescription	An <i>EnumeratedType</i> inherits all elements and attributes defined for a <i>TypeDescription</i> in Table 28 and for an <i>OpaqueType</i> defined in Table 29. The <i>LengthInBits</i> 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 31.

**Table 31 – StructuredType Components**

Name	Type	Description
TypeDescription	TypeDescription	A <i>StructuredType</i> inherits all elements and attributes defined for a <i>TypeDescription</i> in Table 28.
Field	FieldType	One or more elements that describe the fields of the structure. Each field shall have a name that is unique within the <i>StructuredType</i> . Some fields may reference other fields in the <i>StructuredType</i> by using this name.
anyAttribute	*	Authors of a <i>TypeDictionary</i> may add their own attributes to any <i>StructuredType</i> 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 32.

**Table 32 – 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 understand what is contained in the field.
Name	xs:string	An attribute that specifies a name for the <i>Field</i> that is unique within the <i>StructuredType</i> . Other fields in the structured type reference a <i>Field</i> by using this name.
TypeName	xs:QName	An attribute that specifies the <i>TypeDescription</i> 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 <i>IsLengthInBytes</i> attributes specifies which of these definitions applies.
LengthField	xs:string	An attribute that indicates which other field in the <i>StructuredType</i> 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 <i>IsLengthInBytes</i> 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 <i>SwitchField</i> 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 <i>FieldType</i> referenced by this attribute shall precede the field with the <i>StructuredType</i> .
IsLengthInBytes	xs:boolean	An attribute that indicates whether the <i>Length</i> or <i>LengthField</i> 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 <i>Field</i> that controls whether this field is present in the encoded value. The field referenced by this attribute shall be an integer value (see the <i>LengthField</i> attribute).  The current value of the switch field is compared to the <i>SwitchValue</i> attribute using the <i>SwitchOperand</i> . If the condition evaluates to true then the field appears in the stream.  If the <i>SwitchValue</i> attribute is not specified, then this field is present if the value of the switch field is non-zero. The <i>SwitchOperand</i> field is ignored if it is present.  If the <i>SwitchOperand</i> attribute is missing, then the field is present if the value of the switch field is equal to the value of the <i>SwitchValue</i> attribute.  The <i>Field</i> referenced by this attribute shall precede the field with the <i>StructuredType</i> .
SwitchValue	xs:unsignedInt	This attribute specifies when the field appears in the encoded value. The value of the field referenced by the <i>SwitchName</i> attribute is compared using the <i>SwitchOperand</i> 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:  <div style="display: flex; justify-content: space-between;"> <div>Equal</div> <div><i>SwitchField</i> is equal to the <i>SwitchValue</i>.</div> </div> <div style="display: flex; justify-content: space-between;"> <div>GreaterThan</div> <div><i>SwitchField</i> is greater than the <i>SwitchValue</i>.</div> </div> <div style="display: flex; justify-content: space-between;"> <div>LessThan</div> <div><i>SwitchField</i> is less than the <i>SwitchValue</i>.</div> </div> <div style="display: flex; justify-content: space-between;"> <div>GreaterThanOrEqual</div> <div><i>SwitchField</i> is greater than or equal to the <i>SwitchValue</i>.</div> </div> <div style="display: flex; justify-content: space-between;"> <div>LessThanOrEqual</div> <div><i>SwitchField</i> is less than or equal to the <i>SwitchValue</i>.</div> </div> <div style="display: flex; justify-content: space-between;"> <div>NotEqual</div> <div><i>SwitchField</i> is not equal to the <i>SwitchValue</i>.</div> </div> In each case the field is present if the expression is true.
Terminator	xs:hexBinary	This attribute indicates that the field contains one or more instances of <i>TypeDescription</i> referenced by this field and that the last value has the binary

		<p>encoding specified by the value of this attribute.</p> <p>If this attribute is specified then the <i>TypeDescription</i> 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.</p> <p>Examples:</p> <table><tr><th><u>Field Data Type</u></th><th><u>Terminator</u></th><th><u>Byte Order</u></th><th><u>Hexadecimal String</u></th></tr><tr><td>Char</td><td>tab character</td><td>not applicable</td><td>09</td></tr><tr><td>WideChar:</td><td>tab character</td><td>BigEndian</td><td>0009</td></tr><tr><td>WideChar:</td><td>tab character</td><td>LittleEndian</td><td>0900</td></tr><tr><td>Int16</td><td>1</td><td>BigEndian</td><td>0001</td></tr><tr><td>Int16</td><td>1</td><td>LittleEndian</td><td>0100</td></tr></table>	<u>Field Data Type</u>	<u>Terminator</u>	<u>Byte Order</u>	<u>Hexadecimal String</u>	Char	tab character	not applicable	09	WideChar:	tab character	BigEndian	0009	WideChar:	tab character	LittleEndian	0900	Int16	1	BigEndian	0001	Int16	1	LittleEndian	0100
<u>Field Data Type</u>	<u>Terminator</u>	<u>Byte Order</u>	<u>Hexadecimal String</u>																							
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	*	<p>Authors of a <i>TypeDictionary</i> may add their own attributes to any <i>FieldType</i> 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.</p>																								

### C.2.7 EnumeratedValue

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

The components of an *EnumeratedValue* are described in Table 33.

**Table 33 – 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 34.

**Table 34 – ImportDirective Components**

Name	Type	Description
Namespace	xs:string	This attribute specifies the <i>TargetNamespace</i> for the <i>TypeDictionary</i> 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 <i>TypeDictionary</i> to import. This value could be a URL for a network resource, a Nodetd 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 35 – 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 single precision floating point value.
Double	An IEEE-754 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 necessary to explicitly 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  $\leq 0$ .

#### 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 *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:StructuredType 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:sequence>
    <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:complexType>
  <xs:attribute name="TargetNamespace" type="xs:string" use="required" />
  <xs:attribute name="DefaultByteOrder" type="ByteOrder" use="optional" />
</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 Scope

This Appendix defines a graphical notation for OPC UA data. This Appendix is normative, i.e. the notation is used in the specification 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





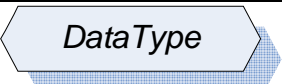
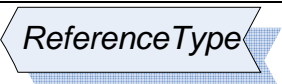


The notation is divided into two parts. The simple notation only provides a simplified view on the data hiding some details like *Attributes*. The extended notation allows exposing all structure information of OPC UA, including *Attribute* values. The simple and the extended notation can be combined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

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

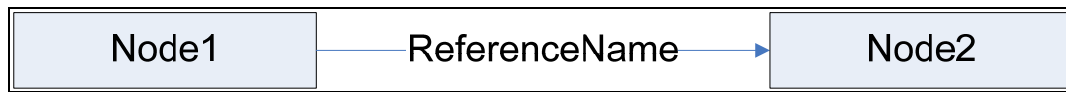
#### D.2.2 Simple Notation

Depending on their *NodeClass* *Nodes* are represented by different graphical forms as defined in Table 36.

**Table 36 – Notation of Nodes depending on the NodeClass**

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

*References* are represented as lines between *Nodes* as exemplified in Figure 43. 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 43 –Example of a Reference connecting two Nodes**

Table 37 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 37 – Simple Notation of Nodes depending on the NodeClass**

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

### D.2.3 Extended Notation

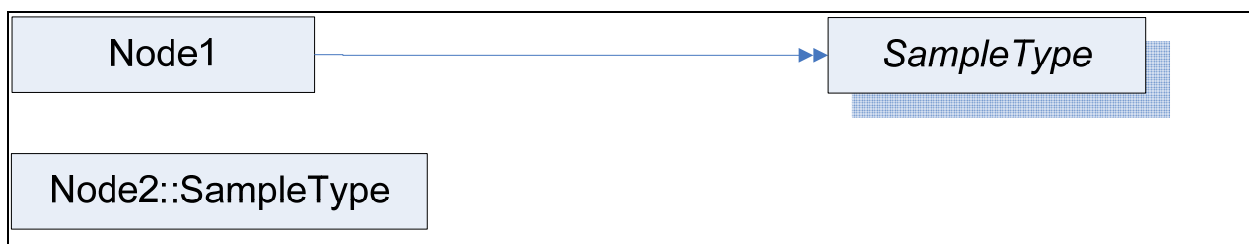
In the extended Notation some additional concepts are introduced. It is allowed only using some of those concepts on elements of a figure.

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*.

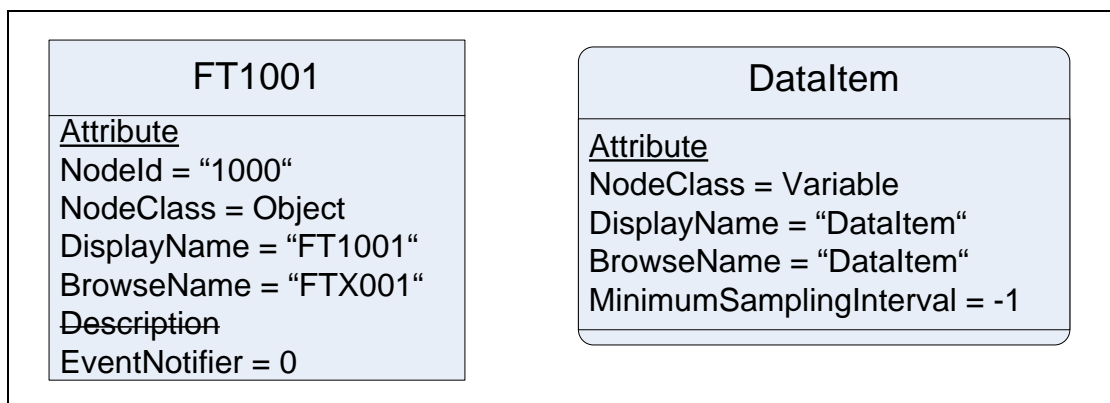
- The *BrowseName* contains the *NamespaceIndex* and a *String*. Such a structure can be exposed as [<NamespaceIndex>:]<String>; where the *NamespaceIndex* is optional. For example, a *BrowseName* can be “1:MyName”. Instead of that, also “MyName” can be used. This rule applies whenever a *BrowseName* 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 44 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*.



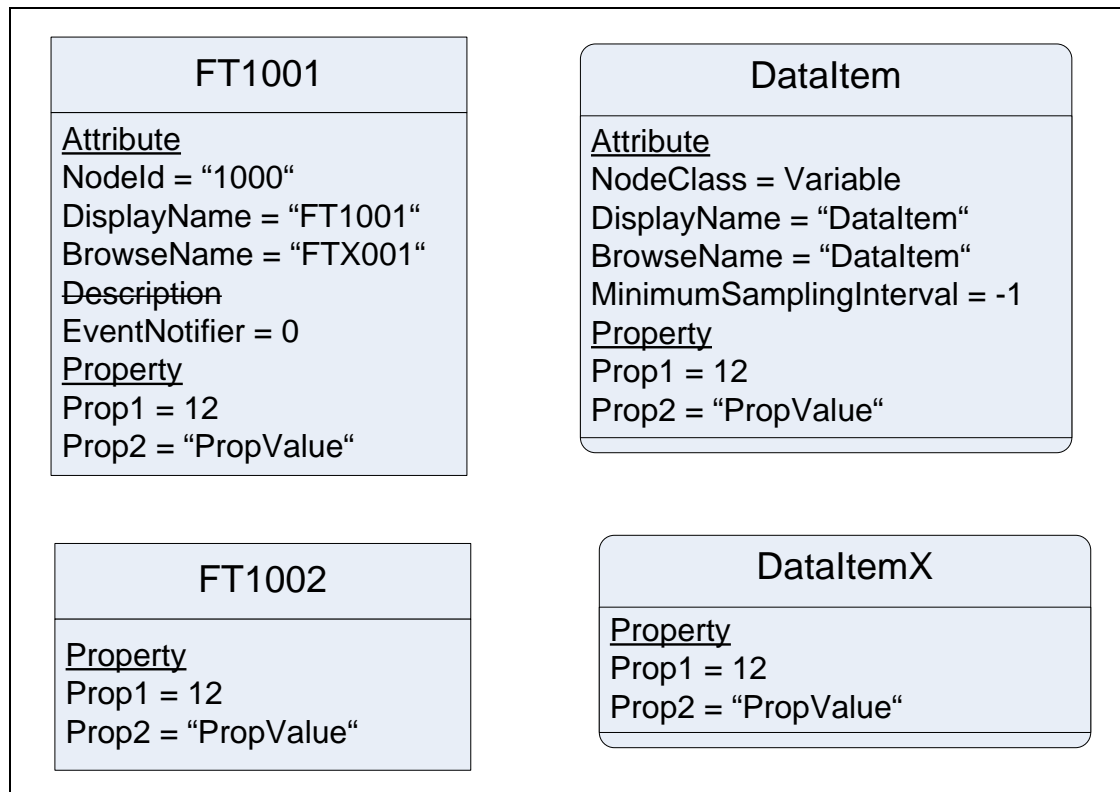
**Figure 44 – Example of using a TypeDefinition inside a Node**

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 only showing a subset of *Attributes*. If an optional *Attribute* is not provided, the *Attribute* can be marked in a line by strike through it, like “~~Description~~”. Examples of exposing *Attributes* are shown in Figure 45.



**Figure 45 – 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 46 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 46 – Example of exposing Properties inline**