OPC Unified Architecture

Specification

Part 3: Address Space Model

Version 1.00

July 28, 2006

CONTENTS

1 Scope

This specification describes the OPC UA *AddressSpace* and its *Objects*.

2 Reference documents

[UA Part 1] OPC UA Specification: Part 1 – Concepts, Version 1.0 or later
http://www.opcfoundation.org/UA/Part1/

[UA Part 2] OPC UA Specification: Part 2 – Security Model, Version 1.0 or later
http://www.opcfoundation.org/UA/Part2/

[UA Part 4] OPC UA Specification: Part 4 – Services, Version 1.0 or later
http://www.opcfoundation.org/UA/Part4/

[UA Part 5] OPC UA Specification: Part 5 – Information Model, Version 1.0 or later
http://www.opcfoundation.org/UA/Part5/

[UA Part 6] OPC UA Specification: Part 6 – Mapping, Version 1.0 or later
http://www.opcfoundation.org/UA/Part6/

[UA Part 8] OPC UA Specification: Part 8 – Data Access, Version 1.0 or later
http://www.opcfoundation.org/UA/Part8/

[XML Schema Part 1] http://www.w3.org/TR/xmlschema-1/

[XML Schema Part 2] http://www.w3.org/TR/xmlschema-2/

[EDDL] IEC 61804-3 Ed. 1.0 – Function blocks (FB) for process control - Part 3: Electronic Device Description Language (EDDL)

[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 [UA 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 [UA Part 2] terms used in this part.

3.3 OPC UA Address Space Model terms

3.3.1 DataVariable

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

3.3.2 EventType

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

3.3.3 Hierarchical Reference

Hierarchical References are *References* that are used to construct hierarchies in the *AddressSpace*. All hierarchical *ReferenceTypes* are derived from the *HierarchicalReferences*.

3.3.4 InstanceDeclaration

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

3.3.5 ModellingRule

The *ModellingRule* of a *Node* defines how the *Node* will be used for instantiation or which rule was used to create the *Node*. It also defines subtyping rules for InstanceDeclaration and can specify the ownership of a *Node*.

3.3.6 Property

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

3.3.7 SourceNode

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

3.3.8 TargetNode

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

3.3.9 TypeDefinitionNode

A *TypeDefinitionNode* is a *Node* that is used to define the type of another *Node*. *ObjectType*, *VariableType*, *ReferenceType*, and *data type Nodes* are *TypeDefinitionNodes*.

3.3.10 VariableType

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

3.4 Abbreviations and symbols

EDDL	Electronic Device Description Language
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.

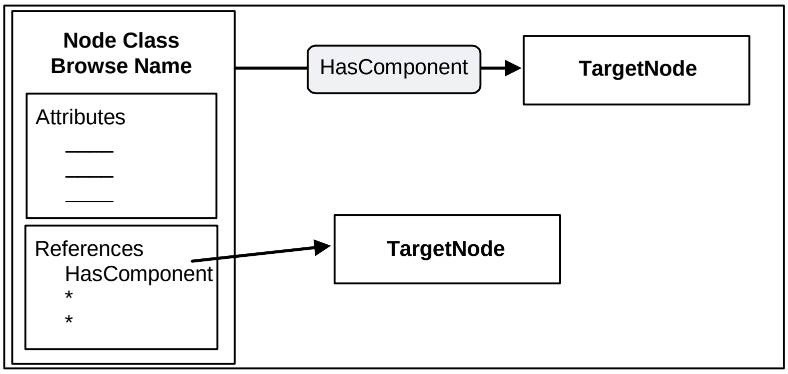

Figure 1 – AddressSpace Node diagrams

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 standard *AddressSpace NodeClasses*. Table 1 describes the format of the tables used to define *NodeClasses*.

Table 1 – NodeClass Table Conventions

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

The Name column contains the name of the *Attribute*, the name of the *ReferenceType* used to create a *Reference* or the name of a standard *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* must 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 OPC UA 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.

OPC UA also defines standard *ReferenceTypes*, but *ReferenceTypes* can also be specified by a server or by a client using the *NodeManagement Services* specified in [UA 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*.

OPC UA defines standard *Properties*, but standard *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.

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. Standard *ReferenceTypes*, *DataTypes* and *EventTypes* are defined in Clauses 6-8.

The informative Appendix A describes general considerations how to use the Address Space Model and the informative Appendix B provides a UML Model of the Address Space Model.

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.

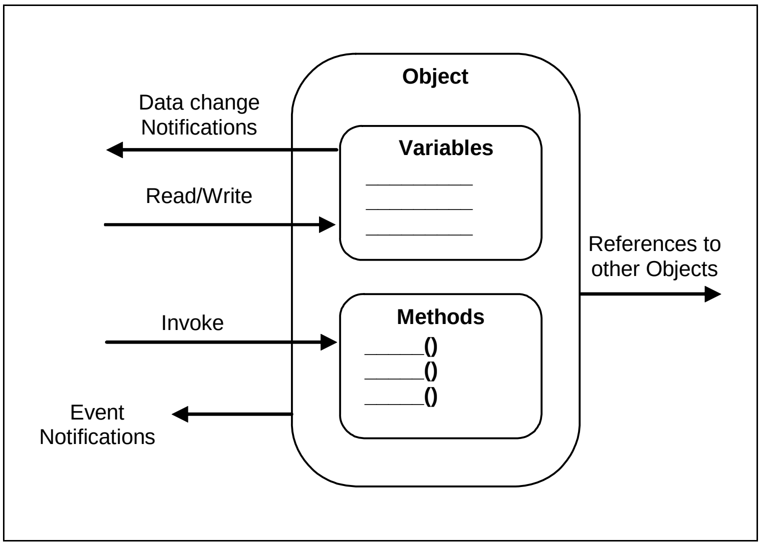

Figure 2 – OPC UA Object Model

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 Clause 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 Clause discusses the details of the Node Model.

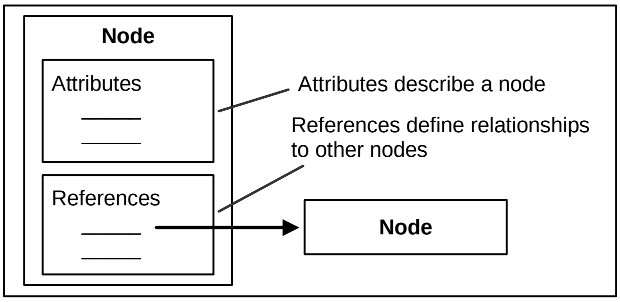

Figure 3 – AddressSpace Node Model

4.3.2 NodeClasses

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

Clause 5 defines the standard *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* may 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 [UA 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 integer id[1], a name, a description, a data type and a mandatory/optional indicator. The set of *Attributes* defined for each *NodeClass* may 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 [UA 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 Clause 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. Figure 4 illustrates this model of a *Reference*.

[1] The integer ids of *Attributes* are defined in [UA Part 6].

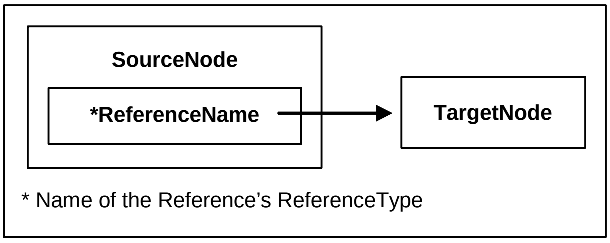

Figure 4 – Reference Model

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.

Server names are URIs that identify OPC servers on the network. Each server name is unique within the scope of the network in which it is installed. If the network has Internet scope, then it is globally unique.

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, instead of defining additional metadata that is used to instantiate all *Nodes* from a *NodeClass*.

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”.

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

A *Node* and its *Properties* must 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*.

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* must always be elements of their complex definitions. Following the example introduced by the description of *Properties* in Clause 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 requires servers to provide type definitions for *Objects* and *Variables*. The *HasTypeDefinition Reference* is used to link an instance with its type definition represented by a *TypeDefinitionNode*. This *Reference* is mandatory; OPC UA requires type definitions. However, [UA 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.

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 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* must 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*.

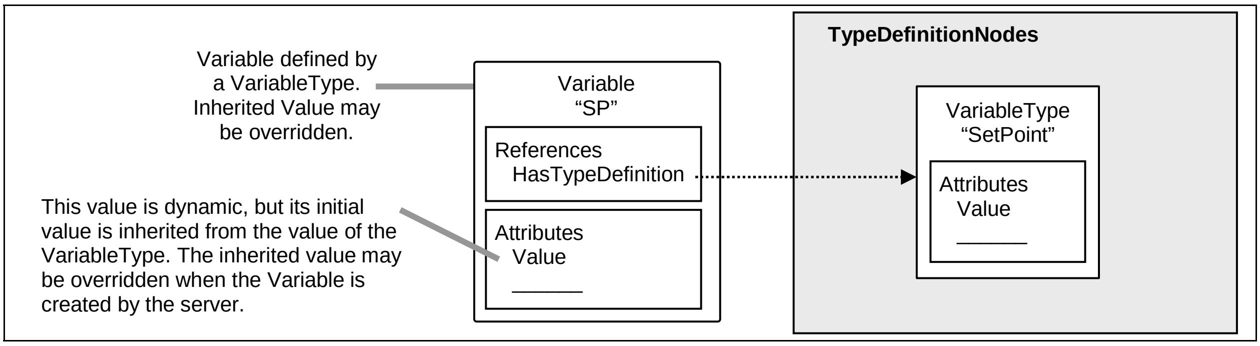

Figure 5 – Example of a Variable Defined By a VariableType

Objects and *Variables* inherit the *Attributes* specified by their *TypeDefinitionNode*.

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 Clause 5.11 specify how those *Nodes* are handled when creating an instance of the type definition.

A *TypeDefinitionNode* typically 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.

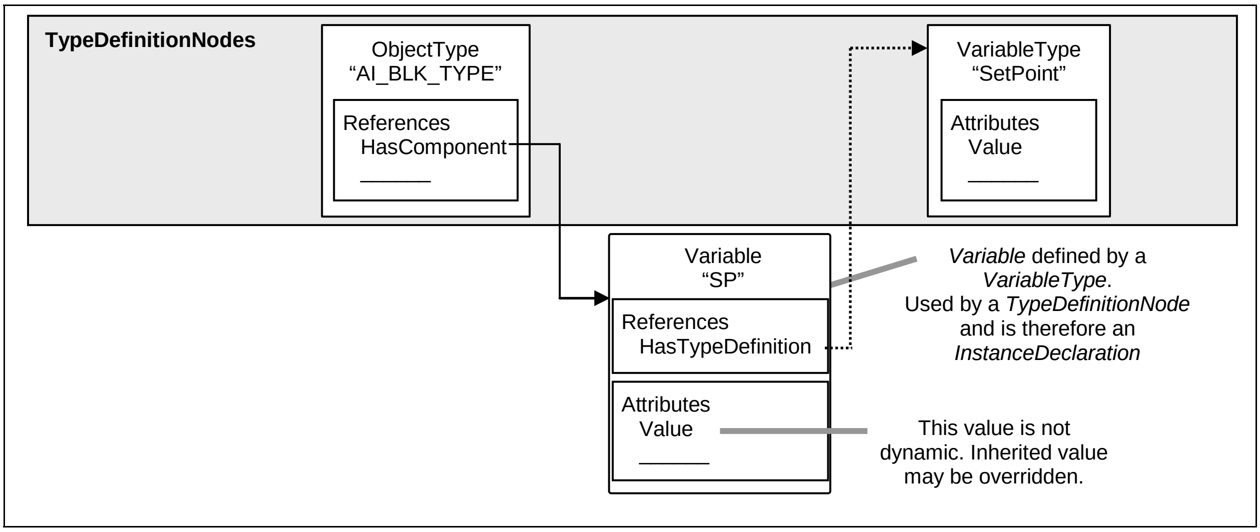

Figure 6 – Example of a Complex TypeDefinition

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

OPC UA allows subtyping of type definitions. The subtyping rules are defined in Clause 5.10. 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;
- the creation of 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 Clause 5.11. 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 New*, defined in Clause 5.11.3.3, 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*.

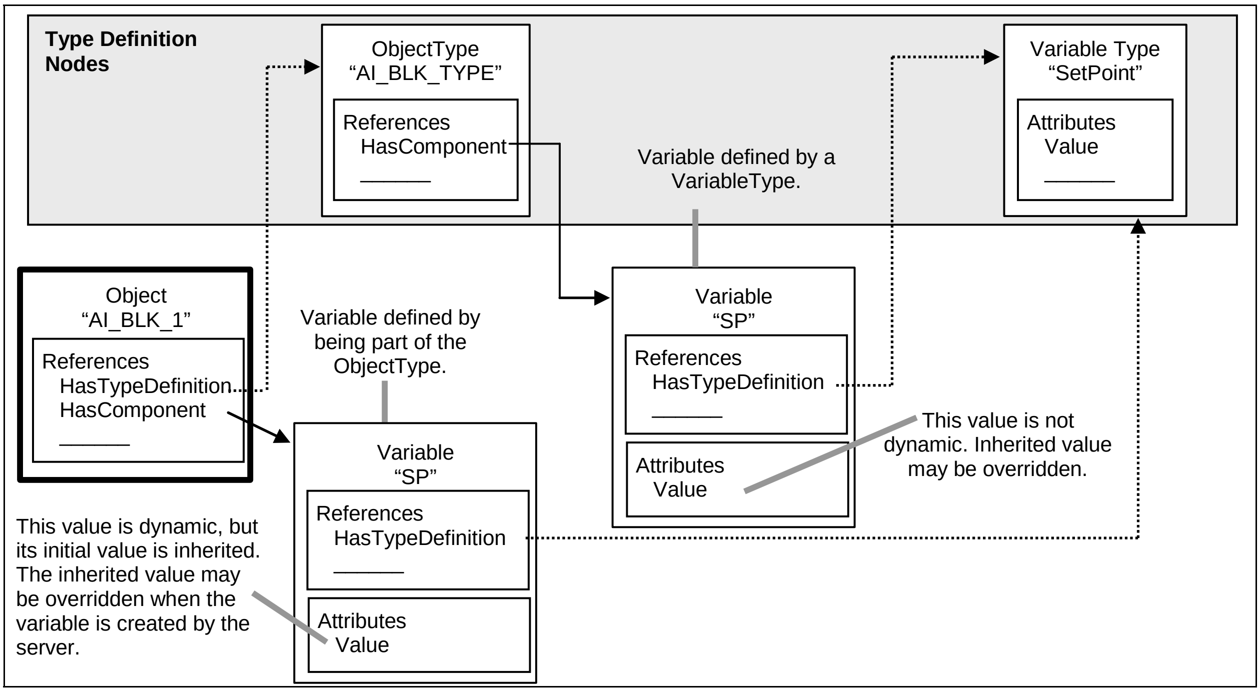

Figure 7 – Object and its Components defined by an ObjectType

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”.

To allow this simple addressing, a *TypeDefinitionNode* or an *InstanceDeclaration* must never reference two *Nodes* having the same *BrowseName* using *hierarchical References*. Instances based on *InstanceDeclarations* must always keep the same *BrowseName* as the *InstanceDeclaration* they are derived from. A special *Service* defined in [UA Part 4] called TranslateBrowsePathToNodeIds may be used to identify the instances based on the *InstanceDeclarations*. Using the simple Browse *Service* may 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 *InstanceDeclaration* must be of the same *TypeDefinitionNode* or a subtype of this *TypeDefinitionNode*.

A *TypeDefinitionNode* and its *InstanceDeclarations* must 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 standard Monitoring and Subscription *Services* defined in [UA Part 4] to subscribe to *Event Notifications* of a *Node*.

Any UA server that supports eventing must expose at least one *Node* as *EventNotifier*. The server *Object* defined in [UA Part 5] is used for this purpose. All *Events* generated by the server are available via this standard server *Object*.

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*. The only exception is the server *Object* that must be capable of providing all *Events* to a subscribing client.

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. [UA 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 8. Their representation in the *AddressSpace* is specified in [UA Part 5].

4.6.3 Event Categorization

Events can be categorised by using two mechanisms in combination or individually. New *EventTypes* can be defined and *Events* can be organised by using the *Event ReferenceTypes* described in Clause 6.16 and 6.17.

New *EventTypes* can be defined as subtypes of existing *EventTypes* that 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 Clause 6.16 and 6.17. 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[2] *Object*, similar to the methods of a class in object-oriented programming. *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 [UA 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. Clause 5.10 describes the rules for subtyping and Clause 5.11 the rules for instantiation of the type definitions.

[2] The owning *Object* is specified in the service call when invoking the *Method*.

5.2 Base NodeClass

5.2.1 General

The OPC UA Address Space Model defines a *Base NodeClass* from which all other *NodeClasses* are derived. The derived *NodeClasses* represent the various components of the OPC UA Object Model (see Clause 4.2). The *Attributes* of the *Base NodeClass* are specified in Table 2. There are no *References* specified for the *Base NodeClass*.

Table 2 - Base NodeClass

Name	**Use**	**Data Type**	**Description**
Attributes			
NodeId	M	NodeId	See Clause 5.2.2
NodeClass	M	NodeClass	See Clause 5.2.3
BrowseName	M	QualifiedName	See Clause 5.2.4
DisplayName	M	LocalizedText	See Clause 5.2.5
Description	O	LocalizedText	See Clause 5.2.6
References			No *References* specified for this *NodeClass*

5.2.2 NodeId

Nodes are unambiguously identified using a constructed identifier called the *NodeId*. Some servers may accept alternative *NodeIds* in addition to the canonical *NodeId* represented in this *Attribute*. The structure of the *NodeId* is defined in Clause 7.2.

5.2.3 NodeClass

The *NodeClass Attribute* identifies the *NodeClass* of a *Node*. Its data type is defined in Clause 7.21.

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 TranslateBrowsePathToNodeId *Service* defined in [UA 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*.

Clause 7.3 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 standard *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* must 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 [UA Part 4] for a description of session establishment and locales. Clause 7.5 defines the structure of the *DisplayName*.

5.2.6 Description

The optional *Description Attribute* must explain the meaning of the *Node* in a localised text using the same mechanisms as described for the *DisplayName* in Clause 5.2.5.

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

OPC UA defines a set of standard *ReferenceTypes* provided as an inherent part of the OPC UA Address Space Model. These *ReferenceTypes* are defined in Clause 6 and their representation in the *AddressSpace* is defined in [UA Part 5]. Servers may also define *ReferenceTypes*. In addition, [UA Part 4] defines *NodeManagement Services* that allow clients to add *ReferenceTypes* to the *AddressSpace*.

Table 3 – ReferenceType NodeClass

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

5.3.2 Attributes

The *ReferenceType NodeClass* inherits the base *Attributes* from the *Base NodeClass* defined in Clause 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* must 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 constrains 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* must be omitted. Examples of symmetric *ReferenceTypes* are “Connects To” and “Communicates With”. Both imply the same semantic coming from the *SourceNode* or the *TargetNode*.

If the *ReferenceType* is non-symmetric and not abstract, the *InverseName Attribute* must 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*.

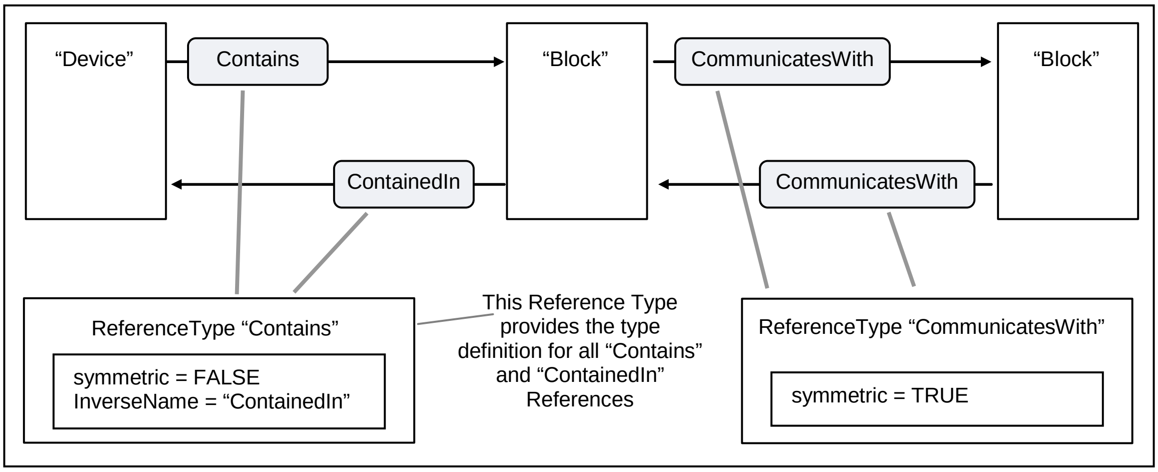

Figure 8 – Symmetric and Non-Symmetric References

It may 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*). OPC UA does not specify how this semantic should be exposed. However, the *Description Attribute* can be used for this purpose. OPC UA does provide a semantic for the standard *ReferenceTypes* specified in Clause 6.

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 must never be able to return to A, that is, a “No Loop” constraint.

OPC UA 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 6. OPC UA 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* must 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 must only refer to *Nodes* of the *Variable NodeClass*.

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

There are no additional standard *Properties* defined for *ReferenceTypes* in this document. Additional parts of this multi-part specification may define additional standard *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 “must 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* can have only zero or one super type. 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, OPC UA 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 4.

Table 4 – View NodeClass

<table>
<tr><th>Name</th><th>Use</th><th>Data Type</th><th>Description</th></tr>
<tr><td>Attributes</td><td></td><td></td><td></td></tr>
<tr><td>Base NodeClass Attributes</td><td>M</td><td>--</td><td>Inherited from the Base NodeClass. See Clause 5.2</td></tr>
<tr><td>ContainsNoLoops</td><td>M</td><td>Boolean</td><td>If set to “true” this Attribute indicates that following References in the context of the View contains no loops, i.e. starting from a Node “A” contained in the View and following the References in the context of the View Node “A” will not be reached again. It does not specify that there is only one path starting from the View Node to reach a Node contained in the View.
If set to “false” this Attribute indicates that following References in the context of the View may lead to loops.</td></tr>
<tr><td>EventNotifier</td><td>M</td><td>Byte</td><td>The EventNotifier Attribute is used to indicate if the Node can be used to subscribe to Events or the read / write historic Events.
The EventNotifier is an 8-bit unsigned integer with the structure defined in the following table:
<table>
<tr><th>Field</th><th>Bit</th><th>Description</th></tr>
<tr><td>SubscribeTo Events</td><td>0</td><td>Indicates if it can be used to subscribe to Events (0 means cannot be used to subscribe to Events, 1 means can be used to subscribe to Events).</td></tr>
<tr><td>Reserved</td><td>1</td><td>Reserved for future use. Must always be zero.</td></tr>
<tr><td>HistoryRead</td><td>2</td><td>Indicates if the history of the Events is readable (0 means not readable, 1 means readable).</td></tr>
<tr><td>HistoryWrite</td><td>3</td><td>Indicates if the history of the Events is writable (0 means not writable, 1 means writable).</td></tr>
<tr><td>Reserved</td><td>4:7</td><td>Reserved for future use. Must always be zero.</td></tr>
</table>
The second two bits also indicate if the history of the Events is available via the OPC UA server.</td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td>References</td><td></td><td></td><td></td></tr>
<tr><td>HierarchicalReferences</td><td>0..*</td><td></td><td>Top level Nodes in a View are referenced by hierarchical References (see Clause 6.3).</td></tr>
<tr><td>HasProperty</td><td>0..*</td><td></td><td>HasProperty References identify the Properties of the View.</td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td>Standard Properties</td><td></td><td></td><td></td></tr>
<tr><td>NodeVersion</td><td>O</td><td>String</td><td>The NodeVersion Property is used to indicate the version of a Node.
The NodeVersion Property is updated each time a Reference is added or deleted to the Node the Property belongs to. Attribute value changes do not cause the NodeVersion to change. Clients may read the NodeVersion Property or subscribe to it to determine when the structure of a Node has changed.</td></tr>
<tr><td>ViewVersion</td><td>O</td><td>UInt32</td><td>The version number for the View. When Nodes are added to or removed from a View, the value of the ViewVersion Property is updated. Clients may detect changes to the composition of a View using this Property. The value of the ViewVersion must always be greater than 0.</td></tr>
</table>

The *View NodeClass* inherits the base *Attributes* from the *Base NodeClass* defined in Clause 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*.

Views are defined by the server. The browsing and querying *Services* defined in [UA 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 standard *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 may not be possible for servers to detect changes in the *View* contents. Servers may also generate a *ModelChangeEvent* described in Clause 8.20 if *Nodes* are added to or deleted from the *View*. There are no additional standard *Properties* defined for *Views* in this document. Additional parts of this multi-part specification may define additional standard *Properties* for *Views*.

Views can be the *SourceNode* of any *hierarchical Reference*. They must 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 5.

Table 5 – Object NodeClass

<table>
<tr><th>Name</th><th>Use</th><th>Data Type</th><th>Description</th></tr>
<tr><td>Attributes</td><td></td><td></td><td></td></tr>
<tr><td>Base NodeClass Attributes</td><td>M</td><td>--</td><td>Inherited from the Base NodeClass. See Clause 5.2</td></tr>
<tr><td>EventNotifier</td><td>M</td><td>Byte</td><td>The EventNotifier Attribute is used to indicate if the Node can be used to subscribe to Events or the read / write historic Events.
The EventNotifier is an 8-bit unsigned integer with the structure defined in the following table:
<table>
<tr><th>Field</th><th>Bit</th><th>Description</th></tr>
<tr><td>SubscribeTo Events</td><td>0</td><td>Indicates if it can be used to subscribe to Events (0 means cannot be used to subscribe to Events, 1 means can be used to subscribe to Events).</td></tr>
<tr><td>Reserved</td><td>1</td><td>Reserved for future use. Must always be zero.</td></tr>
<tr><td>HistoryRead</td><td>2</td><td>Indicates if the history of the Events is readable (0 means not readable, 1 means readable).</td></tr>
<tr><td>HistoryWrite</td><td>3</td><td>Indicates if the history of the Events is writable (0 means not writable, 1 means writable).</td></tr>
<tr><td>Reserved</td><td>4:7</td><td>Reserved for future use. Must always be zero.</td></tr>
</table>
The second two bits also indicate if the history of the Events is available via the OPC UA server.</td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td>References</td><td></td><td></td><td></td></tr>
<tr><td>HasComponent</td><td>0..*</td><td></td><td>HasComponent References identify the DataVariables, the Methods and Objects contained in the Object.</td></tr>
<tr><td>HasProperty</td><td>0..*</td><td></td><td>HasProperty References identify the Properties of the Object.</td></tr>
<tr><td>HasModellingRule</td><td>0..1</td><td></td><td>Objects can point to at most one ModellingRule Object using a HasModellingRule Reference (see Clause 5.11 for details on ModellingRules). If no ModellingRule is specified, the default ModellingRule None is used.</td></tr>
<tr><td>HasTypeDefinition</td><td>1</td><td></td><td>The HasTypeDefinition Reference points to the type definition of the Object. Each Object must have exactly one type definition and therefore be the SourceNode of exactly one HasTypeDefinition Reference pointing to an ObjectType. See Clause 4.5 for a description of type definitions.</td></tr>
<tr><td>Organizes</td><td>0..*</td><td></td><td>This Reference should be used only for Objects of the ObjectType FolderType (see Clause 5.5.3).</td></tr>
<tr><td>HasDescription</td><td>0..1</td><td></td><td>This Reference should be used only for Objects of the ObjectType DataTypeEncodingType (see Clause 5.8.3).</td></tr>
<tr><td>References</td><td>0..*</td><td></td><td>Objects may contain other References.</td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td>Standard Properties</td><td></td><td></td><td></td></tr>
<tr><td>NodeVersion</td><td>O</td><td>String</td><td>The NodeVersion Property is used to indicate the version of a Node.
The NodeVersion Property is updated each time a Reference is added or deleted to the Node the Property belongs to. Attribute value changes do not cause the NodeVersion to change. Clients may read the NodeVersion Property or subscribe to it to determine when the structure of a Node has changed.</td></tr>
</table>

The *Object NodeClass* inherits the base *Attributes* from the *Base NodeClass* defined in Clause 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 standard *Property NodeVersion* is used to indicate the version of the *Object*. There are no additional standard

Properties defined for *Objects* in this document. Additional parts of this multi-part specification may define additional standard *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 Clause 5.11.

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 6.

If the *Object* is used as *InstanceDeclaration* (see Clause 4.5) all *Nodes* referenced with *hierarchical References* must have unique *BrowseNames* in the context of this *Object*.

If the *Object* is created based on an *InstanceDeclaration*, it must 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 6.

Table 6 – ObjectType NodeClass

Name	Use	Data Type	Description
Attributes			
Base NodeClass Attributes	M	--	Inherited from the *Base NodeClass*. See Clause 5.2
IsAbstract	M	Boolean	A boolean *Attribute* with the following values: TRUE it is an abstract *ObjectType*, i.e. no *Objects* of this type must exist, only of its subtypes. FALSE it is not an abstract *ObjectType*, i.e. *Objects* of this type can exist.
References			
HasComponent	0..*		*HasComponent References* identify the *DataVariables*, the *Methods*, and *Objects* contained in the *ObjectType*. If and how the referenced *Nodes* are instantiated when an *Object* of this type is instantiated, is specified in Clause 5.11.
HasProperty	0..*		*HasProperty References* identify the *Properties* of the *ObjectType*. If and how the *Properties* are instantiated when an *Object* of this type is instantiated, is specified in Clause 5.11.
HasSubtype	0..*		*HasSubtype References* identify *ObjectTypes* that are subtypes of this type. The inverse *SubtypeOf Reference* identifies the parent type of this type.
GeneratesEvent	0..*		*GeneratesEvent References* identify the type of *Events* instances of this type may generate.
References	0..*		*ObjectTypes* may contain other *References* that can be instantiated by *Objects* defined by this *ObjectType*.
Standard Properties			
NodeVersion	O	String	The *NodeVersion Property* is used to indicate the version of a *Node*. The *NodeVersion Property* is updated each time a *Reference* is added or deleted to the *Node* the *Property* belongs to. *Attribute* value changes do not cause the *NodeVersion* to change. Clients may read the *NodeVersion Property* or subscribe to it to determine when the structure of a *Node* has changed.

The *ObjectType NodeClass* inherits the base *Attributes* from the *Base NodeClass* defined in Clause 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 standard *Property NodeVersion* is used to indicate the version of the *ObjectType*. There are no additional standard *Properties* defined for *ObjectTypes* in this document. Additional parts of this multi-part specification may define additional standard *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 5.10. 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* may 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 6.

All *Nodes* referenced with *hierarchical References* must have unique *BrowseNames* in the context of an *ObjectType* (see Clause 4.5).

5.5.3 Standard ObjectType FolderType

The *ObjectType FolderType* is formally defined in [UA Part 5]. Its purpose is to provide *Objects* that have no other semantic than organizing of the *AddressSpace*. A special *ReferenceType* is introduced for those *Folder Objects*, the *Organizes ReferenceType*. The *SourceNode* of such a *Reference* should always be an *Object* of the *ObjectType FolderType*; the *TargetNode* can be of any *NodeClass*. *Organizes References* can be used in any combination with *Aggregates References* (*HasComponent*, *HasProperty*, etc.; see Clause 6.5) and do not prevent loops. Thus, they can be used to span multiple hierarchies.

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 Clause 4.4 and have different constraints described in the following subclauses, they use the same *NodeClass* described in Clause 5.6.2. The constraints of *Properties* based on this *NodeClass* are defined in Clause 5.6.3, the constraints of *DataVariables* in Clause 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 Clause 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 7.

Table 7 – Variable NodeClass

<table>
<tr><th>Name</th><th>Use</th><th>Data Type</th><th>Description</th></tr>
<tr><td>Attributes</td><td></td><td></td><td></td></tr>
<tr><td>Base NodeClass Attributes</td><td>M</td><td>--</td><td>Inherited from the Base NodeClass. See Clause 5.2</td></tr>
<tr><td>Value</td><td>M</td><td>Defined by the DataType Attribute</td><td>The most recent value of the Variable that the server has. Its data type is defined by the DataType Attribute. It is the only Attribute that does not have a data type associated with it. This allows all Variables to have a value defined by the same Value Attribute.</td></tr>
<tr><td>DataType</td><td>M</td><td>NodeId</td><td>NodeId of the DataType definition for the Value Attribute. Standard DataTypes are defined in Clause 7.</td></tr>
<tr><td>ArraySize</td><td>M</td><td>Int32</td><td>This Attribute indicates whether the Value Attribute of the Variable is an array.
If it is not an array, the ArraySize is set to -1.
If it is an array, the ArraySize specifies the number of elements in the array.
The value 0 is used to indicate an array whose size has not been allocated or is not known.
For example, if a Variable is defined by the following C array:
Int32 myArray[346];
then this Variable's DataType would point to an Int32 and the Variable's ArraySize has the value 346.</td></tr>
<tr><td>AccessLevel</td><td>M</td><td>Byte</td><td>The AccessLevel Attribute is used to indicate how the Value of a Variable can be accessed (read/write) and if it contains current and/or historic data. The AccessLevel does not take any user access rights into account, i.e. although the Variable is writeable this may be restricted to a certain user / user group.
The AccessLevel is an 8-bit unsigned integer with the structure defined in the following table:
<table>
<tr><th>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. Must always be zero.</td></tr>
</table>
The first two bits also indicate if a current value of this Variable is available and the second two bits indicates if the history of the Variable is available via the OPC UA server.</td></tr>
<tr><td>UserAccessLevel</td><td>M</td><td>Byte</td><td>The UserAccessLevel Attribute is used to indicate how the Value of a Variable can be accessed (read/write) and if it contains current or historic data taking user access rights into account.
The UserAccessLevel is an 8-bit unsigned integer with the structure defined in the following table:
<table>
<tr><th>Field</th><th>Bit</th><th>Description</th></tr>
<tr><td>CurrentRead</td><td>0</td><td>Indicates if the current value is readable
(0 means not readable, 1 means readable).</td></tr>
<tr><td>CurrentWrite</td><td>1</td><td>Indicates if the current value is writable
(0 means not writable, 1 means writable).</td></tr>
<tr><td>HistoryRead</td><td>2</td><td>Indicates if the history of the value is readable
(0 means not readable, 1 means readable).</td></tr>
<tr><td>HistoryWrite</td><td>3</td><td>Indicates if the history of the value is writable
(0 means not writable, 1 means writable).</td></tr>
<tr><td>Reserved</td><td>4:7</td><td>Reserved for future use. Must always be zero.</td></tr>
</table>
The first two bits also indicate if a current value of this Variable is available and the second two bits indicate if the history of the Variable is available via the OPC UA server.</td></tr>
<tr><td>MinimumSamplingInterval</td><td>O</td><td>Int32</td><td>The MinimumSamplingInterval Attribute indicates how "current" the Value of the Variable will be kept. It specifies (in milliseconds) how fast the server can reasonably sample the value for changes (see [UA Part 4] for a detailed description of sampling interval).
A MinimumSamplingInterval of 0 indicates that the server is to monitor the item continuously. A MinimumSamplingInterval of -1 means indeterminate.</td></tr>
</table>

References			
HasModellingRule	0..1		*Variables* can point to at most one *ModellingRule Object* using a *HasModellingRule Reference* (see Clause 5.11 for details on *ModellingRules*). If no *ModellingRule* is specified, the default *ModellingRule None* is used.
HasProperty	0..*		*HasProperty References* are used to identify the *Properties* of a *DataVariable*. *Properties* are not allowed to be the *SourceNode* of *HasProperty References.*
HasComponent	0..*		*HasComponent References* are used by complex *DataVariables* to identify their composed *DataVariables*. *Properties* are not allowed to use this *Reference*.
HasTypeDefinition	1		The *HasTypeDefinition Reference* points to the type definition of the *Variable.* Each *Variable* must have exactly one type definition and therefore be the *SourceNode* of exactly one *HasTypeDefinition Reference* pointing to a *VariableType*. See Clause 4.5 for a description of type definitions.
References	0..*		*Data Variables* may be the *SourceNode* of any other *References*. *Properties* may only be the *SourceNode* of any *non-hierarchical Reference*.
Standard Properties			
NodeVersion	O	String	The *NodeVersion Property* is used to indicate the version of a *DataVariable*. It does not apply for *Properties*. The *NodeVersion Property* is updated each time a *Reference* is added or deleted to the *Node* the *Property* belongs to. *Attribute* value changes except for the *DataType Attribute* do not cause the *NodeVersion* to change. Clients may read the *NodeVersion Property* or subscribe to it to determine when the structure of a *Node* has changed. Although the relationship of a *Variable* to its *DataType* is not modelled using *References*, changes to the *DataType Attribute* of a *Variable* lead to an update of the *NodeVersion Property*.
VariableTimeZone	O	Int32	The *VariableTimeZone Property* is only used for *DataVariables*. It does not apply for *Properties*. This *Property* specifies the time difference (in minutes) between the SourceTimestamp (UTC) associated with the value and the standard time at the location in which the value was obtained. The SourceTimestamp is defined in [UA Part 4]. VariableTimeZone must not be dependent on Standard/Daylight savings time at the originating location, because this would add ambiguities.
DataTypeVersion	O	String	Only used for *Variables* of the *VariableType DataTypeDictionaryType* and *DataTypeDescriptionType* as described in Clause 5.8.
DictionaryFragment	O	String	Only used for *Variables* of the *VariableType DataTypeDescriptionType* as described in Clause 5.8.

The *Variable NodeClass* inherits the base *Attributes* from the *Base NodeClass* defined in Clause 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* and *ArraySize 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 [UA Part 4].

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.

Clients may read or write *Variable* values, or monitor them for value changes, as specified in [UA Part 4]. [UA 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 Clause 5.11.

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

The other *References* are described separately for *Properties* and *DataVariables* in the following subclauses.

5.6.3 Property

Properties are used to define the characteristics of *Nodes*. *Properties* are defined using the *Variable NodeClass*, specified in Table 7. However, they restrict their use.

Properties are the leaf of any hierarchy, therefore they must 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* must point to the *PropertyType* defined in [UA Part 5].

Properties must always be defined in the context of another *Node* and must be the *TargetNode* of at least one *HasProperty Reference*. To distinguish them from *DataVariables*, they must 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 7.

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

The standard *Property NodeVersion* indicates the version of the *DataVariable*. The standard *Property VariableTimeZone* indicates the difference between the SourceTimestamp of the value and the standard time at the location in which the value was obtained. The standard *Property DataTypeVersion* is used only for *DataTypeDictionaries* and *DataTypeDescriptions* as defined in Clause 5.8. The Standard *Property DictionaryFragment* is used only for *DataTypeDescriptions* as defined in Clause 5.8. There are no additional standard *Properties* defined for *DataVariables* in this part of this document. Additional parts of this multi-part specification may define additional standard *Properties* for *DataVariables*. [UA Part 8] defines a standard 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 6.

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* must be the *TargetNode* of at least one *HasComponent Reference* coming from an *Object*, an *ObjectType*, a *DataVariable* or a *VariableType*. *DataVariables* must 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 Clause 4.5) all *Nodes* referenced with *hierarchical References* must 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 8.

Table 8 – VariableType NodeClass

Name	Use	Data Type	Description
Attributes			
Base NodeClass Attributes	M	--	Inherited from the *Base NodeClass*. See Clause 5.2
Value	O	Defined by the *DataType attribute*	The default *Value* for instances of this type.
DataType	M	NodeId	*NodeId* of the data type definition for instances of this type.
ArraySize	M	Int32	This *Attribute* indicates whether the *Value Attribute* of the *VariableType* is an array. If it is not an array, the *ArraySize* is set to -1. If it is an array, the *ArraySize* specifies the number of elements in the array. The value 0 is used to indicate an array whose size has not been allocated or is not known. For example, if a *VariableType* is defined by the following C array: Int32 myArray[346]; then this *VariableType's DataType* would point to an Int32 and the *VariableType's ArraySize* has the value 346.
IsAbstract	M	Boolean	A boolean *Attribute* with the following values: TRUE it is an abstract *VariableType*, i.e. no *Variable* of this type must exist, only of its subtypes. FALSE it is not an abstract *VariableType*, i.e. *Variables* of this type can exist.
References			
HasProperty	0..*		*HasProperty References* are used to identify the *Properties* of the *VariableType*. The referenced *Nodes* may be instantiated by the instances of this type, depending on the *ModellingRules* defined in Clause 5.11.
HasComponent	0..*		*HasComponent References* are used for complex *VariableTypes* to identify their containing *DataVariables*. Complex *VariableTypes* can only be used for *DataVariables*. The referenced *Nodes* may be instantiated by the instances of this type, depending on the *ModellingRules* defined in Clause 5.11.
HasSubtype	0..*		*HasSubtype References* identify *VariableTypes* that are subtypes of this type. The inverse *subtype of Reference* identifies the parent type of this type.
GeneratesEvent	0..*		*GeneratesEvent References* identify the type of *Events* instances of this type may generate.
References	0..*		*VariableTypes* may contain other *References* that can be instantiated by *Variables* defined by this *VariableType*. *ModellingRules* are defined in Clause 5.11.
Standard Properties			
NodeVersion	O	String	The *NodeVersion Property* is used to indicate the version of a *Node*. The *NodeVersion Property* is updated each time a *Reference* is added or deleted to the *Node* the *Property* belongs to. *Attribute* value changes except for the *DataType Attribute* do not cause the *NodeVersion* to change. Clients may read the *NodeVersion Property* or subscribe to it to determine when the structure of a *Node* has changed. Although the relationship of a *VariableType* to its *DataType* is not modelled using *References*, changes to the *DataType Attribute* of a *VariableType* lead to an update of the *NodeVersion Property*.

The *VariableType NodeClass* inherits the base *Attributes* from the *Base NodeClass* defined in Clause 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* and *ArraySize 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 Clause 5.11.

The standard *Property NodeVersion* indicates the version of the *VariableType*. There are no additional standard *Properties* defined for *VariableTypes* in this document. Additional parts of this multi-part specification may define additional standard *Properties* for *VariableTypes*. [UA Part 8] defines a standard 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 5.10. 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 6.

All *Nodes* referenced with *hierarchical References* must have unique *BrowseNames* in the context of the *VariableType* (see Clause 4.5).

5.7 Method NodeClass

Methods define callable functions. *Methods* are invoked using the *Call Service* defined in [UA 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 9.

Table 9 – Method NodeClass

Name	Use	Data Type	Description
Attributes			
Base NodeClass Attributes	M	--	Inherited from the *Base NodeClass*. See Clause 5.2
Executable	M	Boolean	The *Executable Attribute* indicates if the *Method* is currently executable ("False" means not executable, "True" means executable). The *Executable Attribute* does not take any user access rights into account, i.e. although the *Method* is executable this may be restricted to a certain user / user group.
UserExecutable	M	Boolean	The *UserExecutable Attribute* indicates if the *Method* is currently executable ("False" means not executable, "True" means executable). The *Executable Attribute* does not take any user access rights into account, i.e. although the *Method* is executable this may be restricted to a certain user / user group.
References			
HasProperty	0..*		*HasProperty References* identify the *Properties* for the *Method*.
HasModellingRule	0..1		*Methods* can point to at most one *ModellingRule Object* using a *HasModellingRule Reference* (see Clause 5.11 for details on *ModellingRules*). If no *ModellingRule* is specified, the default *ModellingRule None* is used.
References	0..*		*Methods* may contain other *References*.
Standard Properties			
NodeVersion	O	String	The *NodeVersion Property* is used to indicate the version of a *Node*. The *NodeVersion Property* is updated each time a *Reference* is added or deleted to the *Node* the *Property* belongs to. *Attribute* value changes do not cause the *NodeVersion* to change. Clients may read the *NodeVersion Property* or subscribe to it to determine when the structure of a *Node* has changed.
InputArguments	O	Argument[]	The *InputArguments Property* is used to specify the arguments that must be used by a client when calling the *Method.*
OutputArguments	O	Argument[]	The *OutputArguments Property* specifies the result returned from the *Method* call.

The *Method NodeClass* inherits the base *Attributes* from the *Base NodeClass* defined in Clause 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 [UA 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 must be set to "False" if the *Executable Attribute* is set to "False".

Properties may be defined for *Methods* using *HasProperty References*. The standard *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 Clause 7.6. An empty array a *Property* that is not provided indicates that there are no input arguments or output arguments for the *Method*. The standard *Property NodeVersion* indicates the version of the *Method*. There are no additional standard *Properties* defined for *Methods* in this document. Additional parts of this multi-part specification may define additional standard *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 Clause 5.11.

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 6.

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

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.

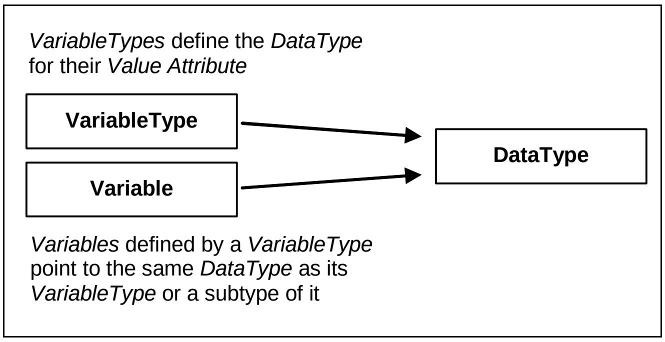

Figure 9 – Variables, VariableTypes and their DataTypes

In many cases, the *NodeId* of the *DataType Node* – the *DataTypeId* – will be well-known to clients and servers. Clause 7 defines standard *DataTypes* and [UA Part 5] defines their *DataTypeIds*. In addition, other organizations may define *DataTypes* that are well-known in the industry. Well-known *DataTypeIds* provide for commonality across 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 may 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 [UA 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*.

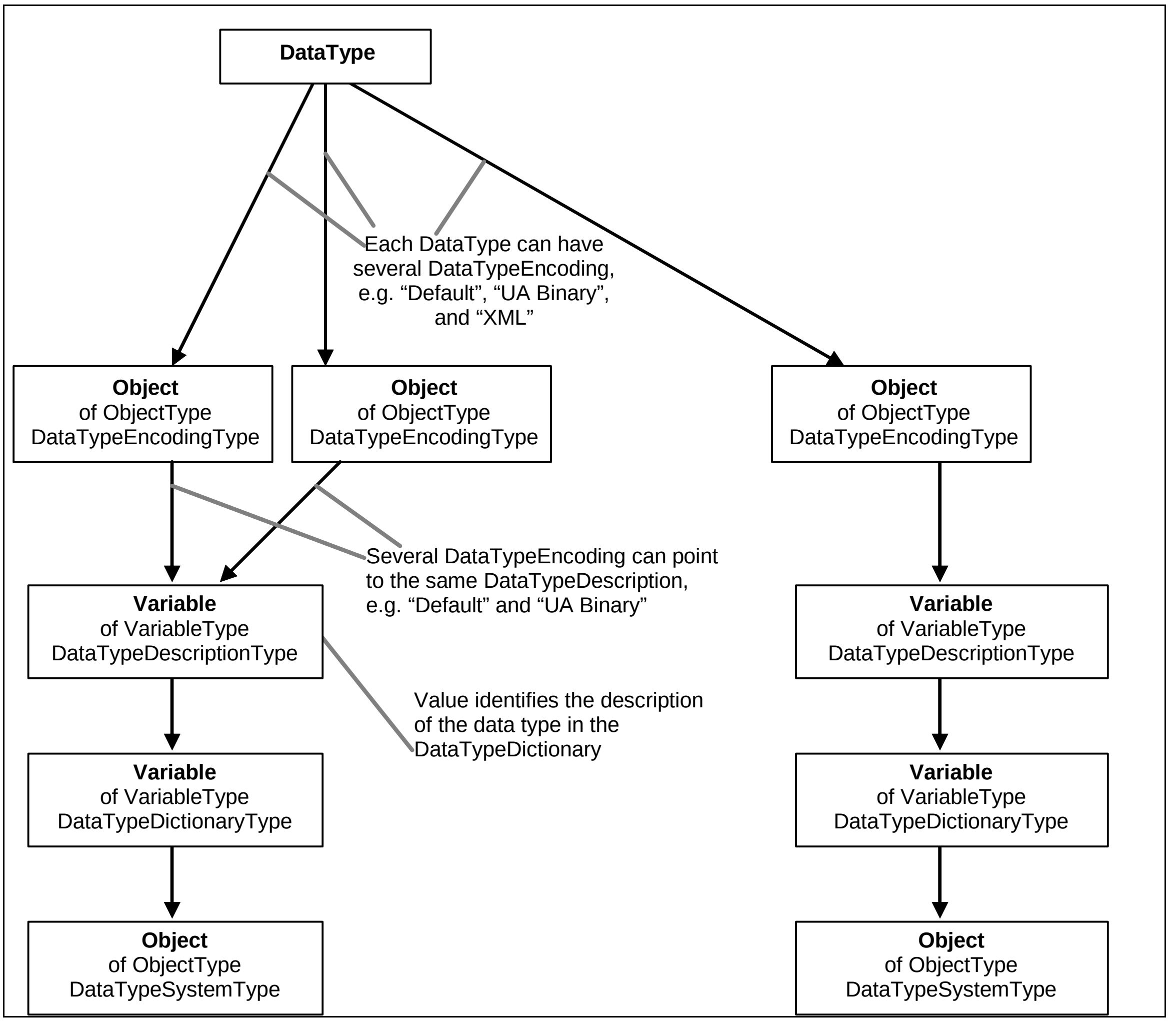

Figure 10 – DataType Model

Since the *NodeId* of the *DataTypeEncoding* will be used in some Mappings to identify the *DataType* and its encoding as defined in [UA 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 *DataTypeSystem*s. *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. 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, W3C XML Schema and Electronic Device Description Language (EDDL) are examples of *DataTypeSystems*. The OPC Binary *DataTypeSystem* is defined in Appendix 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], EDDL in [EDDL].

5.8.2 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 10.

Table 10 – Data Type NodeClass

Name	Use	Data Type	Description
Attributes			
Base NodeClass Attributes	M	--	Inherited from the *Base NodeClass*. See Clause 5.2
References			
HasProperty	0..*		*HasProperty References* identify the *Properties* for the *data type*.
HasSubtype	0..*		*HasSubtype References* may be use to span a data type hierarchy.
HasEncoding	0..*		*HasEncoding References* identify the encodings of the *DataType* represented as *Objects* of type *DataTypeEncodingType*. Each concrete *DataType* must point to at least one *DataTypeEncoding Object* with the *BrowseName* “Default Binary” or “Default XML” having the *NamespaceIndex* 0. The *BrowseName* of the *DataTypeEncoding Objects* must be unique in the context of a *DataType*, i.e. a *DataType* must not point to two *DataTypeEncodings* having the same *BrowseName*. An abstract *DataType* does not point to a *DataTypeEncoding Object*. This is the way to identify if a *DataType* is abstract.
Standard Properties			
NodeVersion	O	String	The *NodeVersion Property* is used to indicate the version of a *Node*. The *NodeVersion Property* is updated each time a *Reference* is added or deleted to the *Node* the *Property* belongs to. *Attribute* value changes do not cause the *NodeVersion* to change. Clients may read the *NodeVersion Property* or subscribe to it to determine when the structure of a *Node* has changed.

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

HasProperty References are used to identify the *Properties* of a *DataType*.

The standard *Property NodeVersion* is used to indicate the version of the *DataType*. This Version is not affect by the *DataTypeVersion Property* of *DataTypeDictionaries* and *DataTypeDescriptions*.

There are no additional standard *Properties* defined for *DataTypes* in this document. Additional parts of this multi-part specification may define additional standard *Properties* for *DataTypes*.

HasSubtype References may be used to expose a data type hierarchy in the *AddressSpace*. This hierarchy must 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 *DataType* can point to many *DataTypeEncodings*, but each *DataTypeEncoding* must 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 must not be the *SourceNode* of other types of *References*. However, they may be the *TargetNode* of other *References*.

5.8.3 DataTypeDictionary, DataTypeDescription, DataTypeEncoding and DataTypeSystem

A *DataTypeDictionary* is an entity that contains a set of type descriptions, such as an XML schema or an EDDL Device Description. *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 must 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 must 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 must re-read the *DataTypeDictionary*. This implies that the *DataTypeVersion* must 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 must 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* must be the name of the *TypeDescription*. When using "XML

Schema" dictionaries the Value must be an XPath expression [XPATH] which points to an XML element in the schema document.

Like *DataTypeDictionaries* each *DataTypeDescription* provides the standard *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 must provide its *DataTypeEncodings* and if a *DataTypeDictionary* is exposed, it should expose all its *DataTypeDescriptions*. Both of these *References* must be bi-directional.

The *VariableTypes DataTypeDictionaryType* and *DataTypeDescriptionType* and the *ObjectTypes DataTypeSystemType* and *DataTypeEncodingType* are formally defined in [UA Part 5].

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

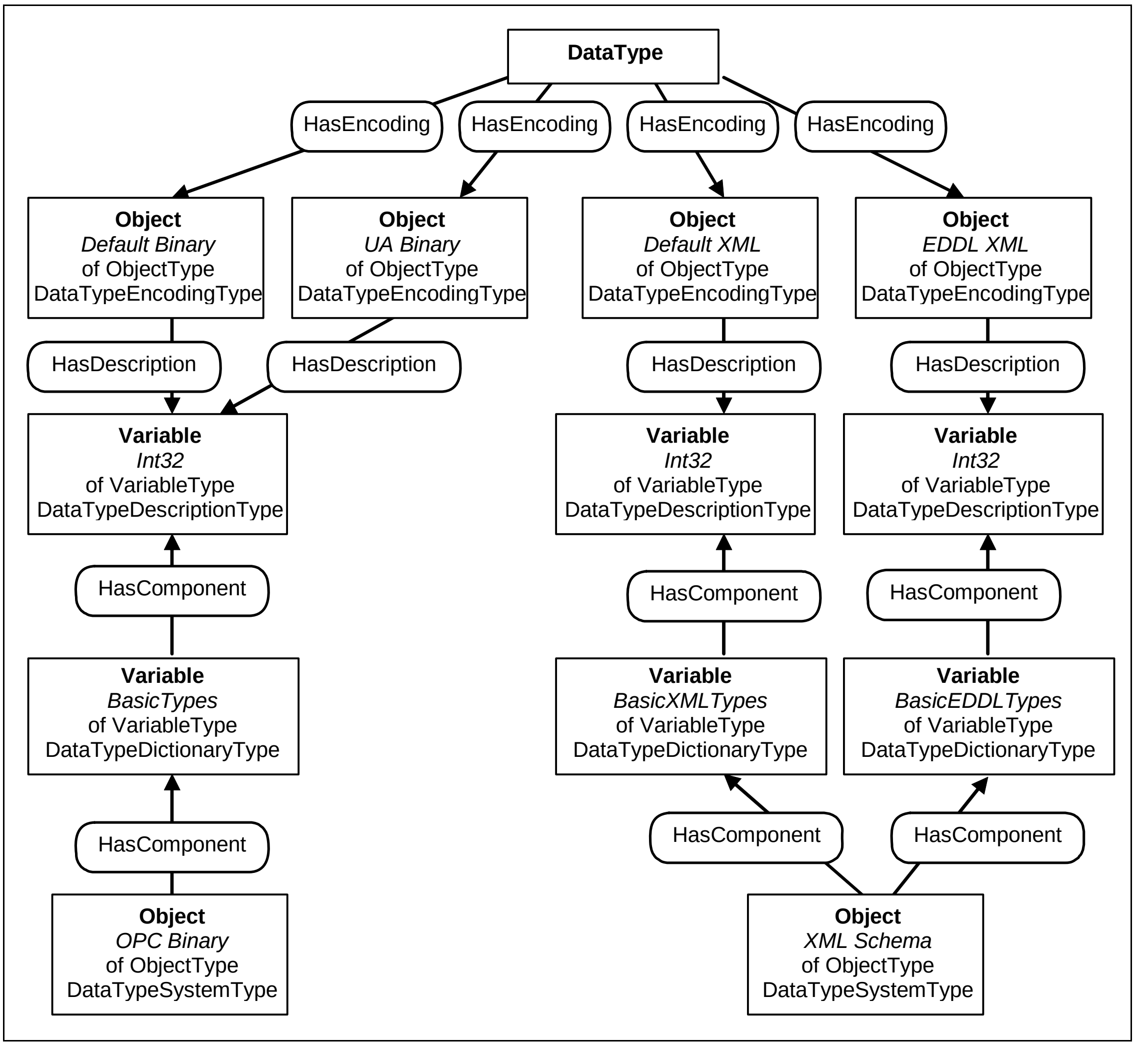

Figure 11 – Example of DataType Modelling

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, UA defines a standard *Property* for the *DataTypeDescription* called *DictionaryFragment* (see Clause 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. Clients can typically act more effective reading the whole *DataTypeDictionary* at once instead of reading several parts and building their own *DataTypeDictionary* over a period of time.

All *DataTypeDictionaries* must be uniquely identified by a URI that is usually assigned by the organization that created the dictionary and may be used by multiple servers. For this reason the canonical *NodeId* for each *DataTypeDictionary* must be this URI. As result, clients may use the URI to read the dictionary from its own configuration or even fetch it from a website (if the URI is a URL). Clients must use the URI to locate the dictionary if the server does not provide either the *Value* of a *DataTypeDictionary* or *DictionaryFragment Properties*.

5.9 Summary of Attributes of the NodeClasses

Table 11 summarises all *Attributes* defined in this document and points out which *NodeClasses* use them either optional (O) or mandatory (M).

Table 11 – Overview about Attributes

Attribute	**Variable**	**Variable Type**	**Object**	**Object Type**	**Reference Type**	**DataType**	**Method**	**View**
AccessLevel	M							
ArraySize	M	M						
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	
InverseName					O			
IsAbstract		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	
Value	M	O						

5.10 Subtyping of ObjectTypes and VariableTypes

The *HasSubtype* standard *ReferenceType* defines subtypes of types. Subtyping can only occur between *Nodes* of the same *NodeClass*. Subtypes do not inherit the parent type's *NodeId* or the parent type's *BrowseName*. Rules for subtyping *ReferenceTypes* are described in Clause 5.3.3.3. There is no common definition for subtyping *DataTypes*, as described in Clause 5.8.2. This Clause specifies subtyping rules for *ObjectTypes* and *VariableTypes*.

The rules for single inheritance from the parent type are:

a) Subtypes inherit the fully-inherited parent type's *Attribute* values, except for the *NodeId* and the *BrowseName*. Inherited *Attribute* values may be overridden by the subtype unless restricted by the *NodeClass* definition. Optional *Attributes*, not provided by the parent type, may be added to the subtype. In this context, fully-inherited means that inheritances for parent types are done first, recursively down from the top-level parent.

b) Subtypes inherit the fully-inherited parent type's *References* to *Nodes* used for instantiation. This depends on the *ModellingRule* of the referenced *Node*. *ModellingRules* are defined in Clause 5.11. In general, inheritance of *References* means that the same *References* are defined for the subtypes. The *TargetNode* for each of these *References* may be the *TargetNode* of the parent type's *Reference*, or another *Node* of the same *NodeClass*. If the *Node* "A" referenced in the parent type has a type definition "Type_A", the *Node* "B" referenced in the subtype must have the same type definition "Type_A" or its type definition must be derived from "Type_A".

c) Changing the values of the *Attributes* of a supertype is not reflected in its subtypes. Whether changing *References* is reflected in the subtypes is server-dependent.

5.11 Instantiation of ObjectTypes and VariableTypes

5.11.1 General

OPC UA requires servers to provide type definitions for *Objects* and *Variables*. If an *Object* or *Variable* is created based on a type, the following rules apply:

a) The fully-inherited type, as defined in Clause 5.10, is used for the instantiation.

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

c) *Nodes* that are referenced from the *TypeDefinitionNode* are instantiated depending on the *ModellingRule* of the *TargetNode*. The *TargetNode ModellingRule* is specified by using the *HasModellingRule Reference* pointing to a *ModellingRule*. Each *Node* may have at most one *ModellingRule*. If no *ModellingRule* is provided, the default *ModellingRule None* is used. If and how the *Node* is instantiated depends on the *ModellingRule*. Clause 5.11.3 defines standard *ModellingRules*.

5.11.2 Ownership due to ModellingRules

OPC UA does not provide a general concept of ownership for its *Nodes*. *DataType Nodes*, for example, may exist in the *AddressSpace* without being referenced and owned by other *Nodes*. The same is true for *ObjectTypes* and *VariableTypes*. It is vendor-specific whether a deletion of such a type leads to the deletion of its subtypes or not.

However, OPC UA provides the concept of ownership for *Variables* and *Methods*, since they must always be part of at least one other *Node*. It also provides the concept of ownership for *Objects*, but the ownership of *Objects* is optional, that is, not every *Object* must be owned.

For each *Method*, *Variable* and *Object* exactly one *ModellingRule* is defined, either using the *HasModellingRule Reference* or using the default *ModellingRule*. Clause 5.11.3 defines standard *ModellingRules*. However, *ModellingRules* are extensible and therefore vendors may define *ModellingRules* having an ownership semantic for other *NodeClasses*, too. Each *ModellingRule* applicable for *Methods* or *Variables* must specify their ownership semantic since they must be referenced by at least one *Aggregates Reference*.

Independent of any ownership semantic and assigned *ModellingRule*, any *Node* may be deleted as result of the deletion of another *Node* or other *Service* invocations. OPC UA does not forbid this, but any rules behind that are server-specific. The same apply for any changes in the *AddressSpace*, e.g. a *Node* may be added when a client connects to the system.

5.11.3 Standard ModellingRules

5.11.3.1 General

OPC UA defines standard *ModellingRules*. The following subclauses define standard *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*. *ModellingRules* are represented in the *AddressSpace* as *Objects* of the *ObjectType ModellingRuleType* or one of its subtypes. [UA Part 5] specifies the representation of the *ModellingRule Objects* and its type in the *AddressSpace*.

5.11.3.2 None

The standard *ModellingRule None* indicates that the *Node* marked with this rule is neither considered for instantiation of a type nor created due to *InstanceDeclarations* of a type.

If a *Node* referenced by a *TypeDefinitionNode* is marked with the *ModellingRule None* 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*.

If a *Node* referenced by an instance "A" is marked with the *ModellingRule None*, it indicates that this *Node* was not created due to the *InstanceDeclarations* of the *TypeDefinitionNode* of "A". For example, a *DataVariable* representing a heat sensor instantiated from a heat sensor *VariableType* may have an additional *Property* not defined by its type, containing the latest maintenance report of the *DataVariable*. However, the *Property* is based on the *PropertyType*.

If the *None ModellingRule* is used for a *Method* or *Variable*, it implies an ownership for those *Nodes*. Any *Method* or *Variable* having a *None ModellingRule* or no *ModellingRule* assigned to it must be the *TargetNode* of exactly one *HasComponent or HasProperty Reference*. If the *SourceNode* of this *Reference* is deleted, the *TargetNode* must also be deleted.

There is no ownership semantic assigned for other *NodeClasses* having no *ModellingRule* or *None* assigned to them.

Since *None* is the default *ModellingRule*, omitting the *ModellingRule* has the same semantic.

Clause 5.11.3.5 gives another example how this standard *ModellingRule* is used.

5.11.3.3 New

The standard *ModellingRule New* indicates that the *Node* referenced by a *TypeDefinitionNode* is newly-created for each instance. This *ModellingRule* applies only to *Methods*, *Objects*, and *Variables*, and affects only *HasProperty and HasComponent References* or their subtypes.

If a *Node* contained by a *TypeDefinitionNode* is marked with the *ModellingRule New* it indicates that a copy of this *Node* will be newly-created for each instance of the type. 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", created as a copy of the "SP1" of the type. Figure 12 illustrates the example. "AI_BLK_1" has no *ModellingRule* that is similar to the *None ModellingRule* since it was directly created based on a *TypeDefinitionNode* and not based on an *InstanceDeclaration* of a *TypeDefinitionNode*.

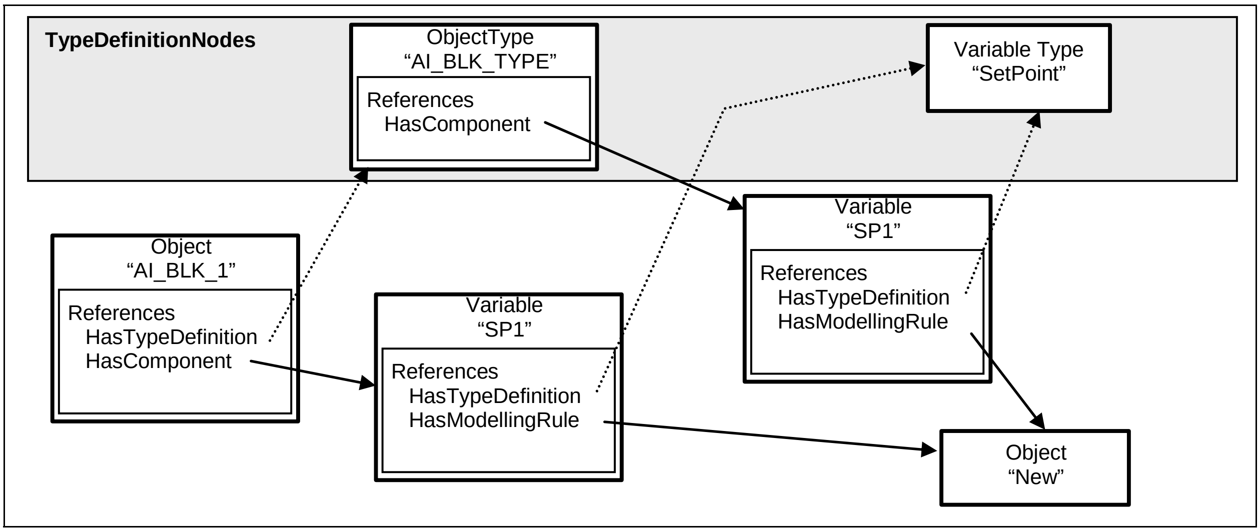

Figure 12 – Use of the Standard ModellingRule New

The following rules for creating a copy apply:

1. A new *Node* will be created having the same *NodeClass* and a different *NodeId*. All other *Attributes* have initially the same values; the *BrowseName* must always be the same.

2. All referenced *Nodes* will be instantiated for the copy depending on their *ModellingRules*.

If a *Node* contained by an instance is marked with the *ModellingRule New*, one of the following cases apply: It indicates that the referenced *Node* was either created due to the *ModellingRule* or that it indirectly belongs to a *TypeDefinitionNode*.

When a *TypeDefinitionNode referencing* a *Node* marked as *New* is subtyped, its subtype either references the same *Node* or another *Node* of the same *NodeClass*. If another *Node* is referenced, either the same type definition or a subtype of it must be used. The *BrowseName* must be the same.

There are two different cases affecting how the ownership of *Nodes* having the *New ModellingRule* applies. The *Node* is either referenced by a *HasComponent* or *HasProperty* of a *TypeDefinitionNode*, or it is not.

In the first case, it may be the *TargetNode* of either one or more *HasComponent References* or one or more *HasProperty References*. The *SourceNodes* of these *References* must all be *TypeDefinitionNodes*, of the same type hierarchy. The server must delete the *Node* if the last *Node* referencing it with a *HasComponent* or *HasProperty Reference* is deleted.

In the second case, it must be the *TargetNode* of exactly one *HasComponent* or one *HasProperty Reference*. If the *SourceNode* of this *Reference* is deleted, the *TargetNode* must also be deleted.

5.11.3.4 Shared

The standard *ModellingRule Shared* specifies that this *Node* can be shared by many other *Nodes*, that is, having many *Nodes* pointing with an *Aggregates Reference* to it.

In general, every *Node* can be marked as *Shared*, that is, many *Aggregates References* can have this *Node* as *TargetNode*. There is no ownership defined for shared *Nodes*; a server may or may not delete a *Node* if a shared *Node* has no *Aggregates Reference* pointing to it.

If the *Node* is a *Variable* or *Method*, the server must delete the *Node* if the last *Node* referencing it with a *HasComponent* or *HasProperty Reference* is deleted, since *Variables* and *Methods* may never stand alone.

If a *TypeDefinitionNode* references a *Node* with the *ModellingRule Shared*, each instance of this type must have the same *Reference* referencing the same *Node*. For example, an *ObjectType* may have a *HasProperty Reference* to a *Property* pointing to an icon. Each instance of the *ObjectType* would also have a *HasProperty Reference* to the same *Property*. However, it is not specified if deleting the *Reference* from the *ObjectType* to the shared *Node* will lead to the same behaviour on the instances.

When a type definition is subtyped, its subtype either references the same *Shared Node* or another *Shared Node* of the same *NodeClass*. If another *Node* is referenced, either the same type definition or a subtype of it must be used.

5.11.3.5 Examples using standard ModellingRules

In Figure 13 an example using the standard *ModellingRules* is shown. The *ObjectType* “AI_BLK_TYPE” contains three *Variables*. “SP1” represents a setpoint and has the *ModellingRule New*, since it must be instantiated for each instance. The “UseCaseScenario” describes how to use the type; therefore it is not needed for the instances and has the *ModellingRule None*. The “Icon” is used to represent each instance of the type in a graphical display, thus it has the *ModellingRule Shared* because all instances can use the same instance.

An instance of “AI_BLK_TYPE” is also shown in Figure 13, called “AI_BLK_1”. Due to the *ModellingRules* it has a *Reference* to the shared “Icon” and a newly created “SP1”. The “UseCaseScenario” was not considered for instantiation purposes. A new *Variable*, “MaintenanceReport” was added to the instance. Since this was not the result of the instantiation of “AI_BLK_1”, but directly instantiated due to a *VariableType* not shown in the figure, it has no *ModellingRule* – having the same semantic as the *None ModellingRule*. “AI_BLK_1” itself has no *ModellingRule*, since it was directly created based on a *TypeDefinitionNode* and not based on an *InstanceDeclaration* of a *TypeDefinitionNode*.

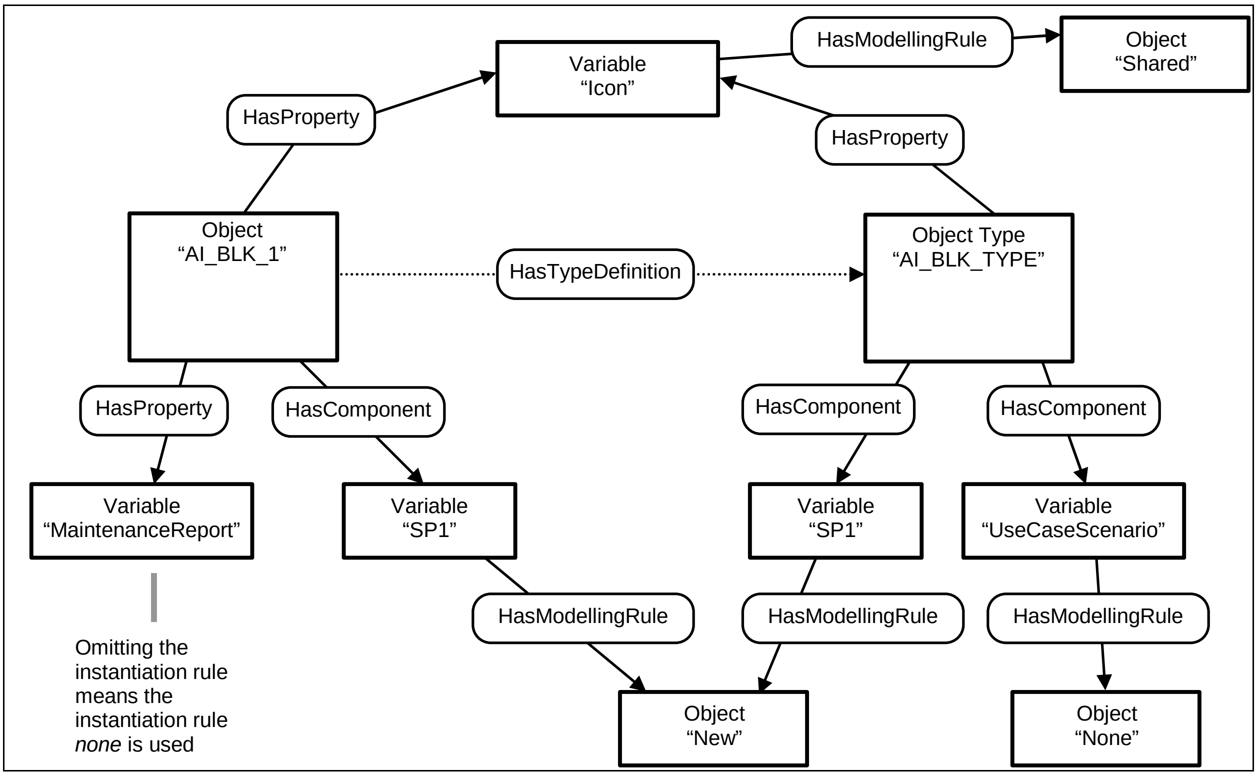

Figure 13 – Example of usage of Standard ModellingRules

6 Standard ReferenceTypes

6.1 General

OPC UA defines standard *ReferenceTypes* as an inherent part of the OPC UA Address Space Model. Figure 14 informally describes the hierarchy of these standard *ReferenceTypes*. Other parts of this multi-part specification may specify additional *ReferenceTypes*. The following subclauses define the standard *ReferenceTypes*. [UA Part 5] defines their representation in the *AddressSpace*.

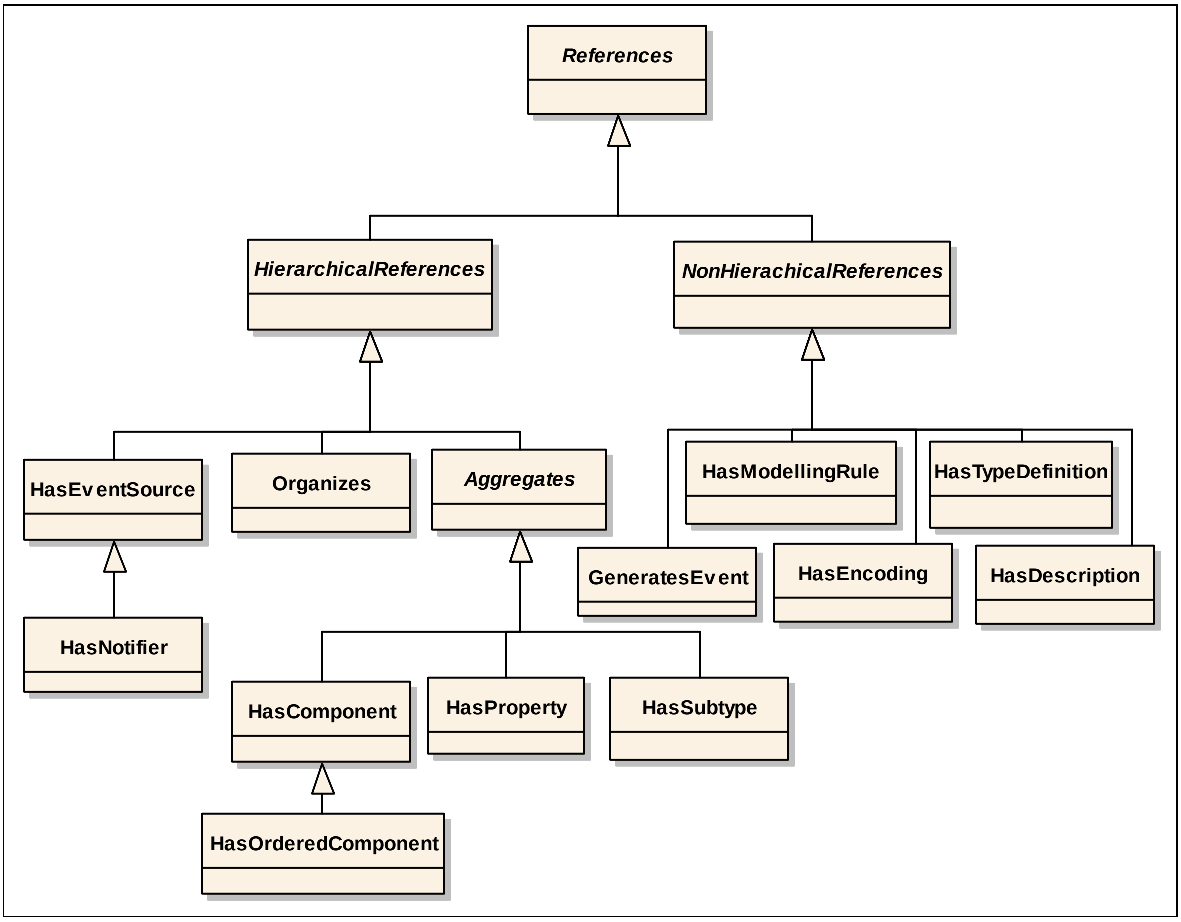

Figure 14 – Standard ReferenceType Hierarchy

6.2 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* must 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 [UA Part 4].

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

6.3 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. It does not forbid loops, that is, 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*.

6.4 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* must inherit from either *hierarchical References* or *non-hierarchical References*, either direct or indirect.

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

6.5 Aggregates ReferenceType

The *Aggregates 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 *Aggregates ReferenceType* must 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”.

6.6 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 *Variable*s or *VariableTypes* with their *DataVariables*.

Like all other *ReferenceTypes*, this *ReferenceType* does not specify anything about the ownership of the parts, although it represents a part-of relationship semantic. That is, it is not specified if the *TargetNode* of a *Reference* of the *HasComponent ReferenceType* is deleted when the *SourceNode* is deleted. *ModellingRules*, as defined in Clause 5.11, can be used for this purpose.

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

If the *TargetNode* is a *Variable*, the *SourceNode* must 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* must be an *Object* or *ObjectType*.

6.7 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 Clause 4.4.2.

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

6.8 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 [UA Part 4] in a well-defined order. The order is server-specific, but the client can assume that the server always returns them in the same order.

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

6.9 HasSubtype ReferenceType

The *HasSubtype ReferenceType* is a concrete *ReferenceType* that can be used directly. It is a subtype of the *Aggregates 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 Clause 5.3.3.3; a *DataType* hierarchy as specified in Clause 5.8.2, as well as other subtype hierarchies as specified in Clause 5.10.

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

6.10 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 must be an *Object;* it should be an *Object* of the *ObjectType FolderType* or one of its subtypes (see Clause 5.5.3).

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

6.11 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 Clause 5.11.

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

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

6.12 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 Clause 4.5.

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

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

6.13 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 must be a *DataType*.

The *TargetNode* of this *ReferenceType* must be an *Object* of the *ObjectType DataTypeEncodingType* or one of its subtypes (see Clause 5.8.3).

6.14 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 must be an *Object* of the *ObjectType DataTypeEncodingType* or one of its subtypes.

The *TargetNode* of this *ReferenceType* must be an *Object* of the *ObjectType DataTypeDescriptionType* or one of its subtypes (see Clause 5.8.3).

6.15 GeneratesEvent

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

The semantic of this *ReferenceType* is to identify the types of *Events* instances of *ObjectTypes* or *VariableTypes* may generate.

The *SourceNode* of *References* of this type must be an *ObjectType* or a *VariableType*.

The *TargetNode* of this *ReferenceType* must be an *ObjectType* representing *EventTypes*, i.e. the *BaseEventType* or one of its subtypes.

6.16 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 [UA 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* must 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 must 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".

6.17 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 Clause 6.16.

The *SourceNode* and *TargetNode* of this *ReferenceType* must 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* must also be provided in the *SourceNode* of the *Reference*.

An example of a possible organization of *Event References* is represented in Figure 15. 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".

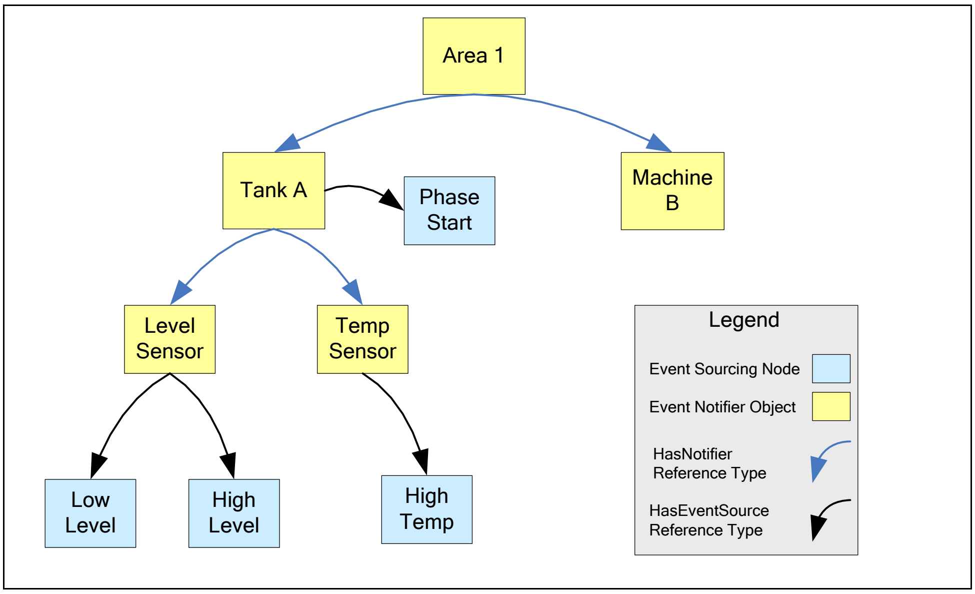

Figure 15 – Event Reference Example

A second example of a more complex organization of *Event References* is represented in Figure 16. 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".

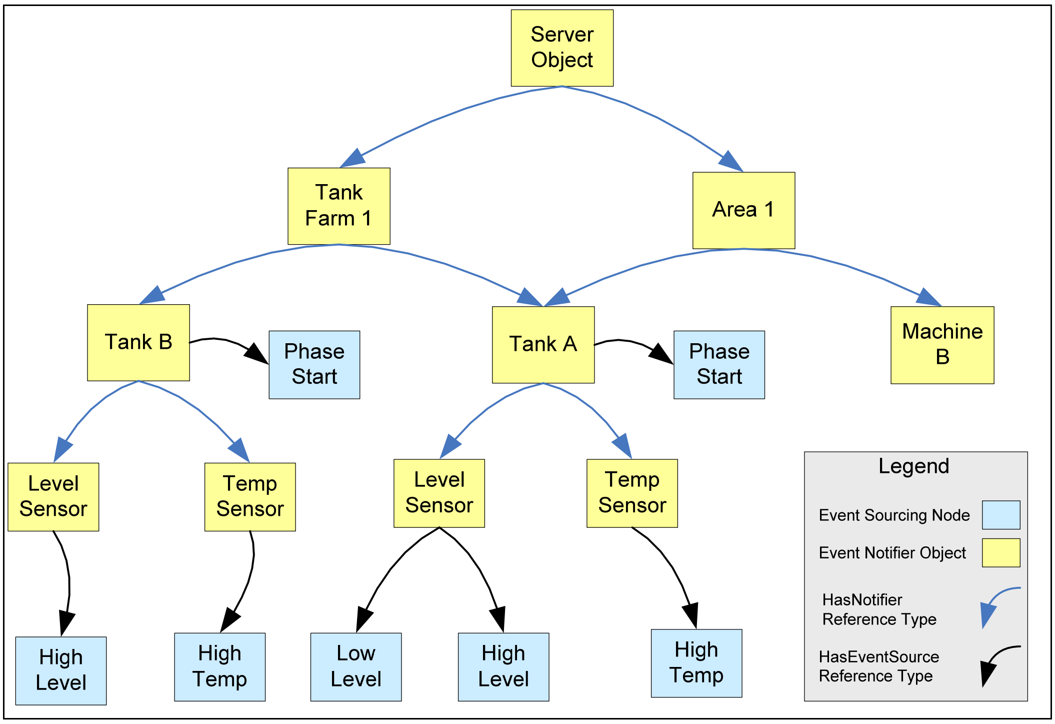

Figure 16 – Complex Event Reference Example

7 Standard DataTypes

7.1 General

The following subclauses define standard *DataTypes* of OPC UA. Their representation in the *AddressSpace* and the *DataType* hierarchy is specified in [UA Part 5]. Other parts of this multi-part specification may specify additional *DataTypes*.

7.2 NodeId

7.2.1 General

NodeIds are composed of three elements that identify a *Node* within a server. They are defined in Table 12.

Table 12 – NodeId Definition

Name	Type	Description
NodeId	structure	
namespaceIndex	UInt32	The index for a namespace URI (see Clause 7.2.2).
identifierType	Enum	The format and data type of the identifier (see Clause 7.2.3).
identifier	*	The identifier for a *Node* in the *AddressSpace* of a UA server (see Clause 7.2.4).

See [UA Part 6] for a description of the encoding of the identifier into OPC UA Messages.

7.2.2 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 [UA 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. Index 1 is reserved to identify the local server.

7.2.3 IdType

The IdType element identifies the type of the *NodeId*, its format and its scope. Its values are defined in Table 13.

Table 13 – IdType Values

Value	Description
NUMERIC	Numeric value
STRING	String value
URI	Universal Resource ID format. Support for this IdType is required.
GUID	Globally Unique Identifier
OPAQUE	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.

NodeIds of the type GUID and URI should always be globally unique, therefore no namespace is provided for them.

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

7.2.4 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.

A Null *NodeId* has special meaning. For example, many services defined in [UA 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 14.

Table 14 – NodeId Null Values

IdType	Identifier
NUMERIC	0
STRING	A Null or Empty String (“”)
URI	A Null or Empty String (“”)
GUID	A Guid initialised with zeros (e.g. 00000000-0000-0000-0000-000000)
OPAQUE	A ByteString with Length=0

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

A *Node* in the *AddressSpace* may not have a Null as its *NodeId*.

7.3 QualifiedName

This primitive *DataType* contains a qualified name. It is, for example, used as *BrowseName*. Its elements are defined in Table 15.

Table 15 – QualifiedName Definition

Name	Type	Description
QualifiedName	structure	
NamespaceIndex	UInt32	Index that identifies the namespace that defines the name. This index is the index of that namespace in the local server’s *NamespaceArray*. The client may read the *NamespaceArray Variable* to access the string value of the namespace.
name	String	The unqualified name.

7.4 LocaleId

This primitive *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 for OPC UA as follows:

d) OPC UA permits only zero or one <country/region> component to follow the <language> component,

e) OPC UA also permits the "-CHS" and "-CHT" three-letter <country/region> codes for "Simplified" and "Traditional" Chinese locales.

f) OPC UA also allows the use of other <country/region> codes as deemed necessary by the client or the server.

Table 16 shows examples of OPC UA *LocaleIds*. Clients and servers always provide *LocaleIds* that explicitly identify the language and the country/region.

Table 16 –LocaleId Examples

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

This DataType defines a special value NULL indicating that the LocaleId is unknown.

7.5 LocalizedText

This primitive *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 17.

Table 17 – LocalizedText Definition

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

7.6 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 Node*. Its elements are described in Table 18.

Table 18 – Argument Definition

Name	Type	Description
Argument	structure	
name	String	The name of the argument
dataType	NodeId	The *NodeId* of the *DataType* of this argument
arraySize	Int32	If the dataType is an array it specifies the number of elements in the array. The value 0 is used to indicate an array whose size is not known. If the dataType is not an array, the value -1 is used.
description	LocalizedText	A localised description of the argument

7.7 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.

7.8 Boolean

This primitive *DataType* defines a value that is either TRUE or FALSE.

7.9 Byte

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

7.10 ByteString

This primitive *DataType* defines a value that is a sequence of Byte values.

7.11 Date

This primitive *DataType* defines a Gregorian calendar date.

7.12 Double

This primitive *DataType* defines a value that adheres to the IEEE 754 Double Precision data type definition.

7.13 Float

This primitive *DataType* defines a value that adheres to the IEEE 754 Single Precision data type definition.

7.14 Guid

This primitive *DataType* defines a value that is a 128-bit Globally Unique Identifier.

7.15 SByte

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

7.16 IdType

This *DataType* is an enumeration that identifies the IdType of a *NodeId*. Its values are defined in Table 13. See Clause 7.2.3 for a description of the use of this *DataType* in *NodeIds*.

7.17 Integer

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

7.18 Int16

This primitive *DataType* defines a value that is a signed integer between -32,768 and 32,767 inclusive.

7.19 Int32

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

7.20 Int64

This primitive *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.

7.21 NodeClass

This *DataType* is an enumeration that identifies a *NodeClass*. Its values are defined in Table 19.

Table 19 – NodeClass Values

Name
DataType
Method
Object
ObjectType
ReferenceType
Variable
VariableType
View

7.22 Number

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

7.23 String

This primitive *DataType* defines a Unicode character string that should exclude control characters that are not whitespaces (0x00 - 0x08, 0x0E-0x1F or 0x7F).

7.24 Time

This primitive *DataType* defines a time in terms of hours, minutes, seconds and fractions of a second. Its granularity is specified by its encoding in [UA Part 6].

7.25 UInteger

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

7.26 UInt16

This primitive *DataType* defines a value that is an unsigned integer between 0 and 65,535 inclusive.

7.27 UInt32

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

7.28 UInt64

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

7.29 UtcTime

This primitive *DataType* is used to define Coordinated Universal Time (UTC) values. All time values conveyed between servers and clients in OPC UA are UTC values. Clients must provide any conversions between UTC and local time.

This *DataType* is represented as a 64-bit signed integer which represents the number of 100 nanosecond intervals since January 1, 1601. [UA Part 6] defines details about this *DataType*.

7.30 XmlElement

This primitive *DataType* is used to define XML elements. [UA Part 6] defines details about this *data type*.

8 Standard EventTypes

8.1 General

The following subclauses define standard *EventTypes* of OPC UA. Their representation in the *AddressSpace* is specified in [UA Part 5]. Other parts of this multi-part specification may specify additional *EventTypes*. Figure 17 informally describes the hierarchy of these standard *EventTypes.*

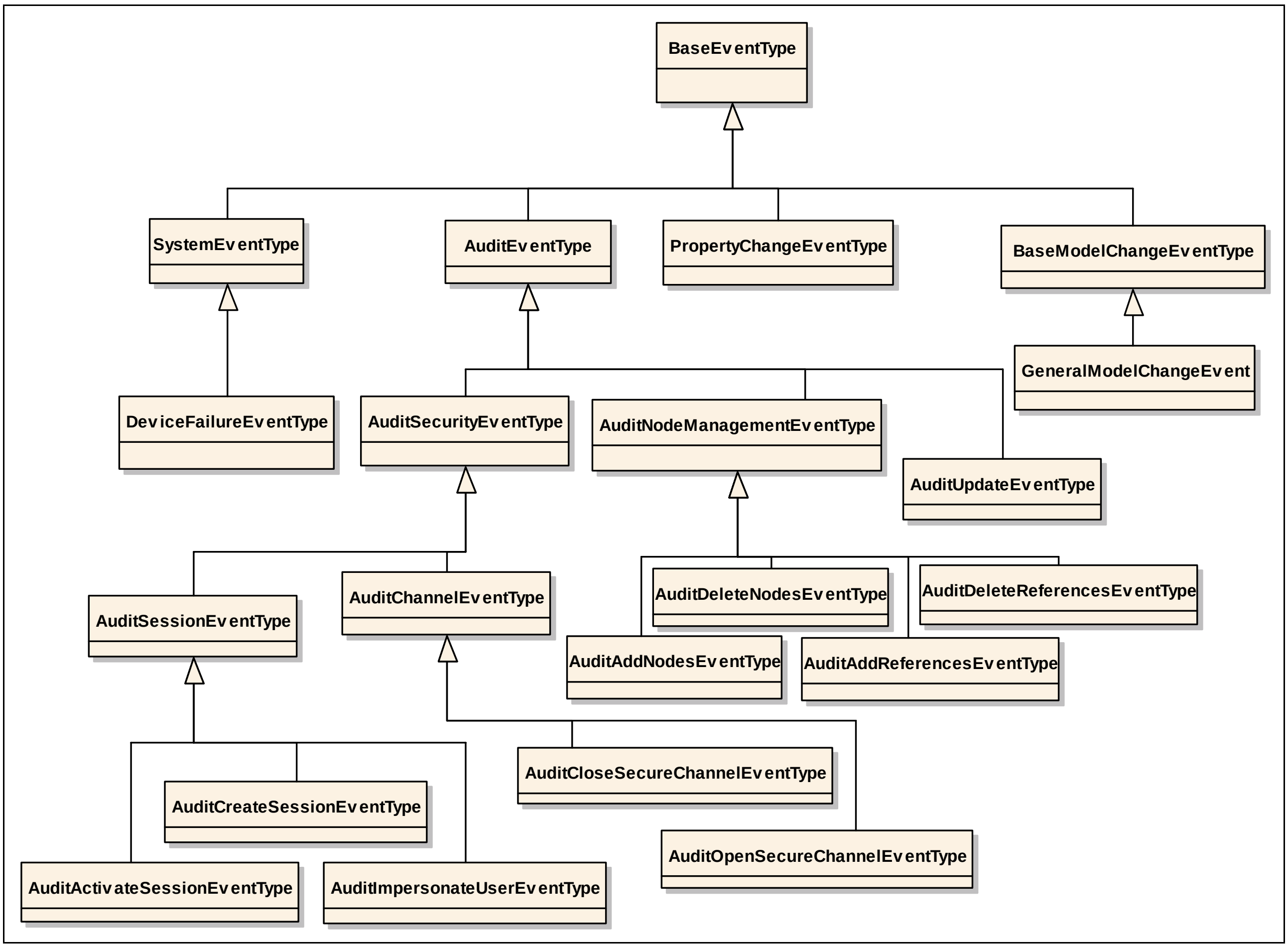

Figure 17 – Standard EventType Hierarchy

8.2 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.

8.3 SystemEventType

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

8.4 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 18 illustrates the OPC UA defined behaviour of a 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 *AuditEventId* 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.

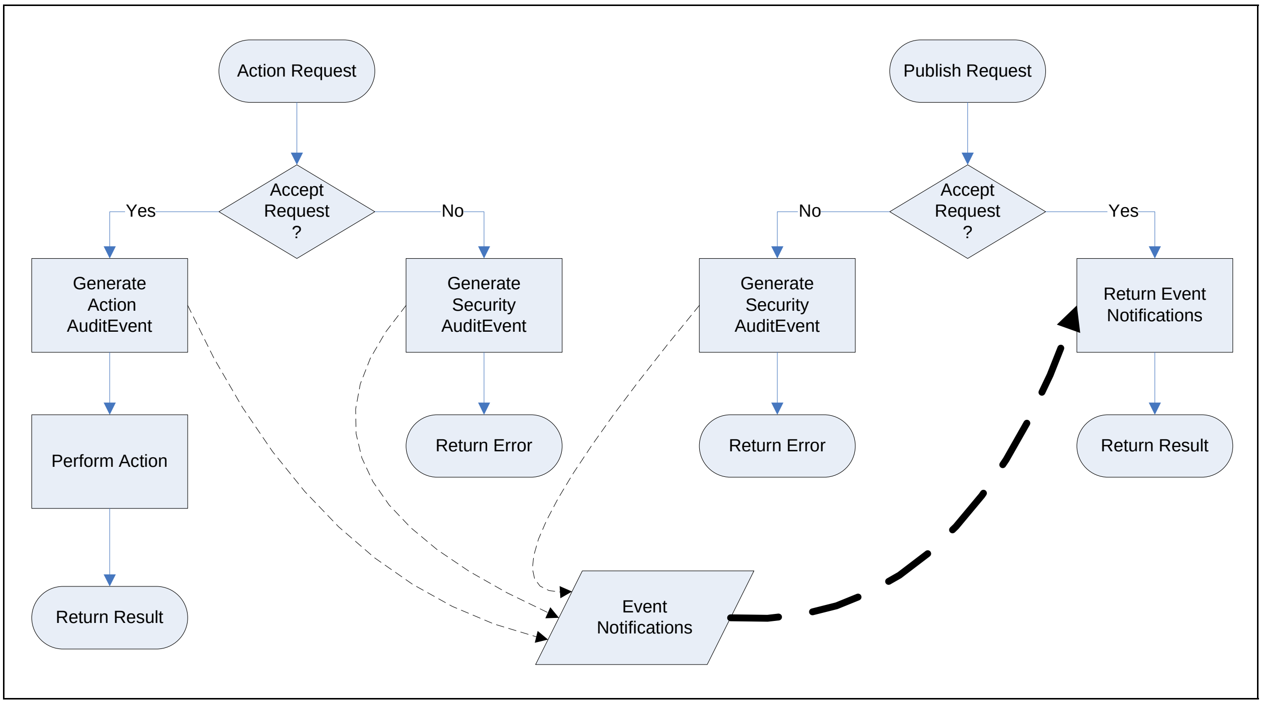

Figure 18 – Audit Behaviour of a Server

Figure 19 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 must, 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.

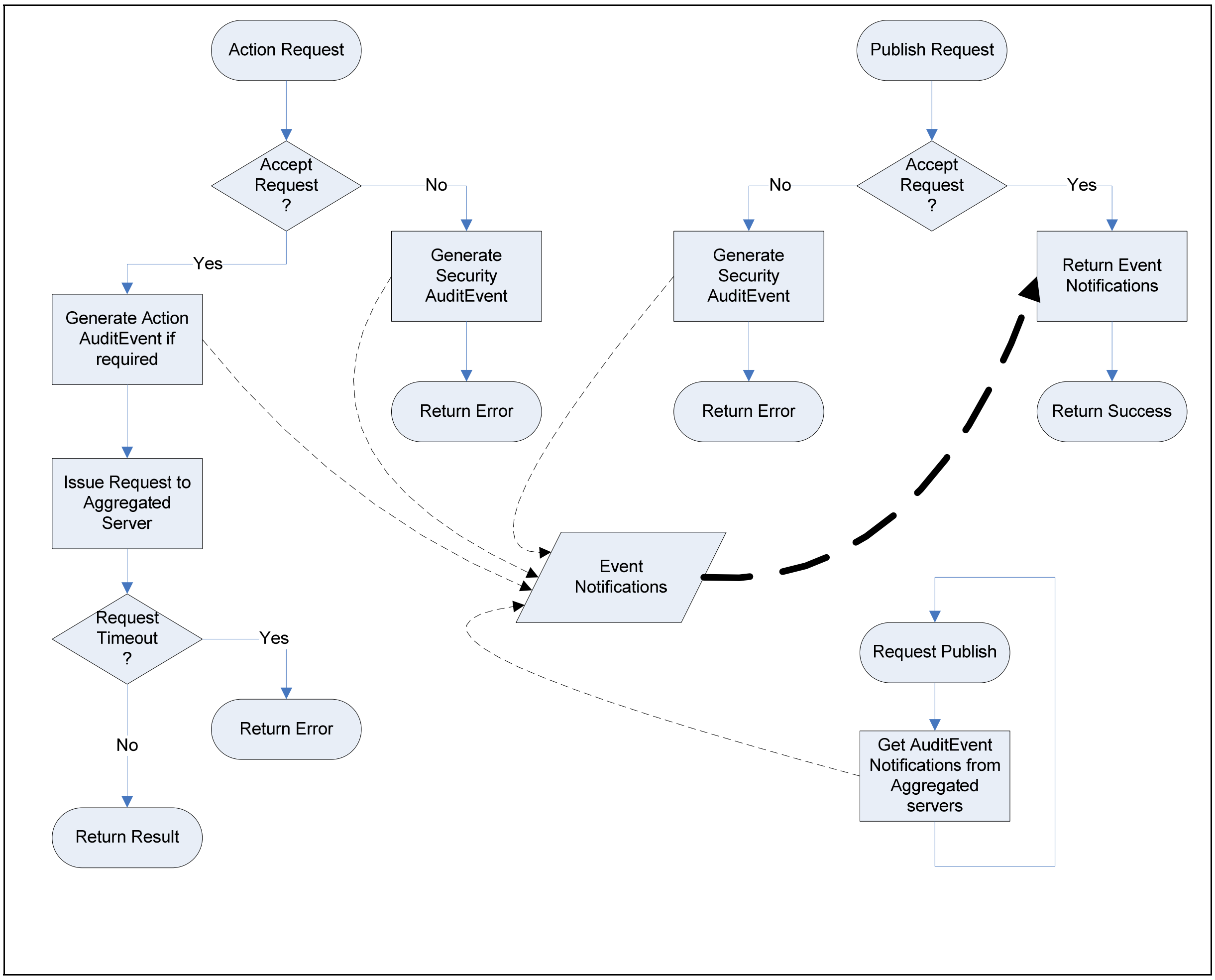

Figure 19 – Audit Behaviour of an Aggregating Server

8.5 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.

8.6 AuditChannelEventType

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

8.7 AuditOpenSecureChannelEventType

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

8.8 AuditCloseSecureChannelEventType

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

8.9 AuditSessionEventType

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

8.10 AuditCreateSessionEventType

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

8.11 AuditActivateSessionEventType

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

8.12 AuditImpersonateUserEventType

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

8.13 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.

8.14 AuditAddNodesEventType

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

8.15 AuditDeleteNodesEventType

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

8.16 AuditAddReferencesEventType

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

8.17 AuditDeleteReferencesEventType

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

8.18 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.

8.19 DeviceFailureEventType

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

8.20 ModelChangeEvents

8.20.1 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.

8.20.2 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* must be changed, and every time the *NodeVersion* is changed, a *ModelChangeEvent* must be generated. A server must support both the *ModelChangeEvent* and the *NodeVersion Property* or neither, but never only one of the two mechanisms.

8.20.3 Views

A *ModelChangeEvent* is always generated in the context of a *View* including the default *View* where the whole *AddressSpace* is considered. Thus 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.

8.20.4 Event Compression

An implementation is not required to issue an *Event* for every update as it occurs. A 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.

8.20.5 BaseModelChangeEventType

The *BaseModelChangeEventType* is the base type for *ModelChangeEvents* and does not contain information about the changes but only indicates that changes occurred. Therefore the client must assume that any or all of the *Nodes* may have changed.

8.20.6 GeneralModelChangeEventType

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

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

8.20.7 Guidelines for ModelChangeEvents

Two types of standard *ModelChangeEvents* are defined: the *BaseModelChangeEvent* that does not contain any information about the changes and the *GeneralModelChangeEvent* that identifies the changed *Nodes* via an array. The precision used depends on both the capability of the UA server and the nature of the update. A 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 UA server should emit exactly one *ModelChangeEvent* for an update or series of updates. It should not issue multiple types of *ModelChangeEvent* for the same update.
- Any client that responds to *ModelChangeEvents* should respond to any *Event* of the *BaseModelChangeEventType* including its subtypes like the *GeneralModelChangeEventType*.

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

8.21 PropertyChangeEventType

8.21.1 General

PropertyChangeEvents are generated to indicate a change of the *AddressSpace* semantics. The change consists of a change to the *Value Attribute* of a *Property*.

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

PropertyChangeEvents are not generated by every change of a *Property*, but only when the *Property* describes the semantic of the *Node* that owns the *Property*. For example, a change in a *Property* describing the engineering unit of a *DataVariable* will issue a *PropertyChangeEvent*, whereas the change of a *Property* containing an Icon of the *DataVariable* will not. For *Variables* and *VariableTypes* this behaviour is exactly the same as described by the *SemanticsChanged* bit of the *StatusCode* defined in [UA 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*.

8.21.2 ViewVersion and NodeVersion Properties

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

8.21.3 Views

PropertyChangeEvents are handled in the context of a *View* the same way as *ModelChangeEvents*. This is defined in Clause 8.20.3.

8.21.4 Event Compression

PropertyChangeEvents can be compressed the same way as *ModelChangeEvents*. This is defined in Clause 8.20.4.

Appendix A: 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 server 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 standards 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 [UA Part 5] gives an overview over the design decisions made when modelling the information about the server defined in [UA 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

Clause 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 [UA 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[3] 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 "Int32VaraibleType".

[3] *VariableTypes* other then 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 [UA 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*.

If none of the above applies and the semantic described in Clause 4 does not make it clear that it has to be a *DataVariable*, it is useful making it a *Property* since *Properties* are easier to handle for the client (e.g. with the BrowseProperties *Service* defined in [UA Part 4]).

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 may 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 must be able to provide the native format and also interpret it to be able to provide the information in simple *DataTypes*.

It is recommended to use the first approach. When a transactional context is needed or the client should be able to get a large amount of data instead of subscribing to several individual values, the third approach is suitable. However, the server may 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 may 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 [UA 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* may not be sufficient to specify the needed behaviour for instantiation and subtyping.

Appendix B: 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 20).

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 [UA Part 5].

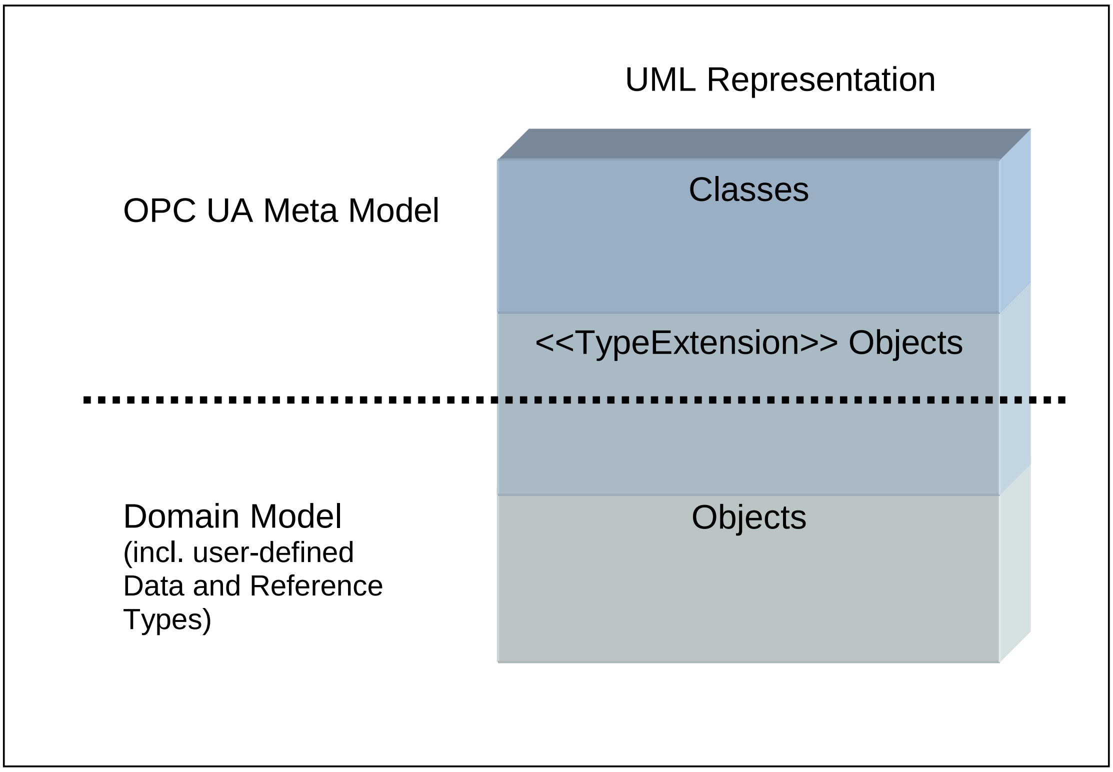

Figure 20 – Background of OPC UA Meta Model

B.2 Notation

An example of a UML class representing the OPC UA concept *BaseNode* is given in the UML class diagram in Figure 21. 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 *BaseNode* in Figure 21.

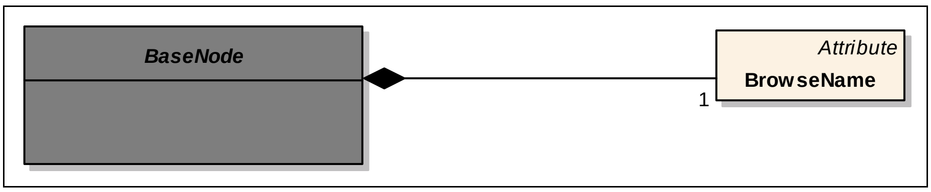

Figure 21 – Notation (I)

UML object diagrams are used to display <<TypeExtension>> objects (e.g. *HasComponent* in Figure 22). 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*).

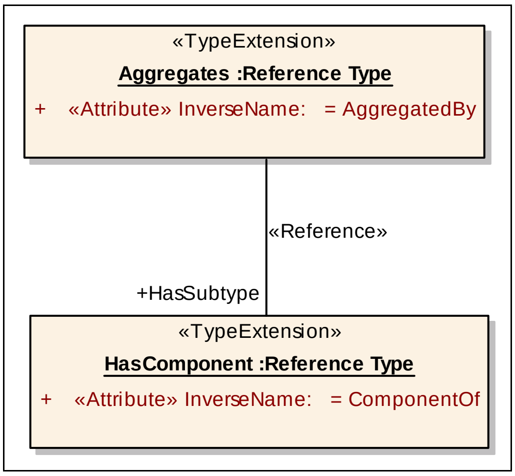

Figure 22 – Notation (II)

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 22 is "HasComponent".

To highlight the classes explained in a class diagram, they are marked grey (e.g. *BaseNode* in Figure 21). 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 BaseNode

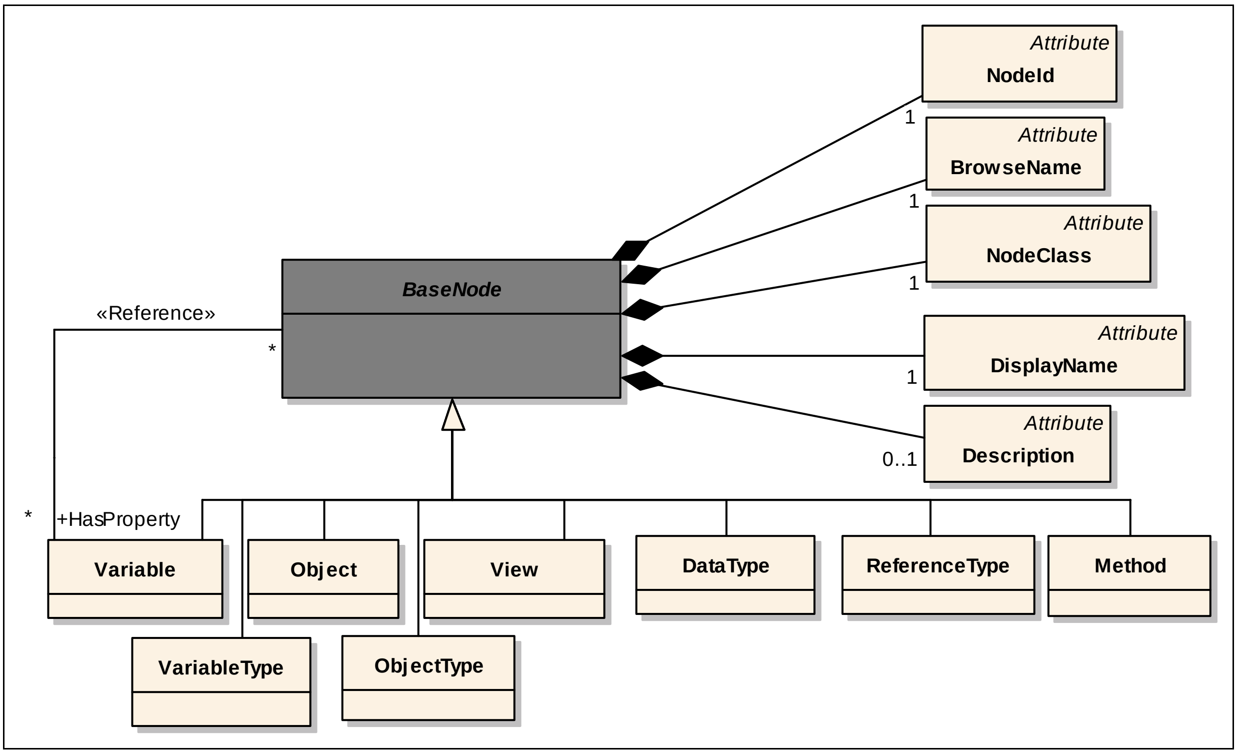

Figure 23 – BaseNode

B.3.2 ReferenceType

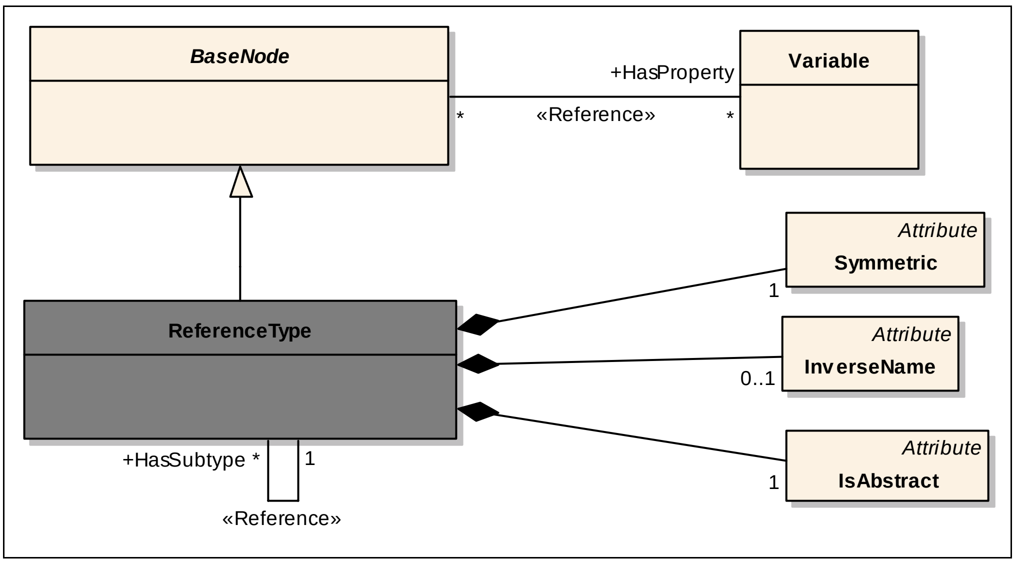

Figure 24 – Reference and ReferenceType

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

B.3.3 Predefined ReferenceTypes

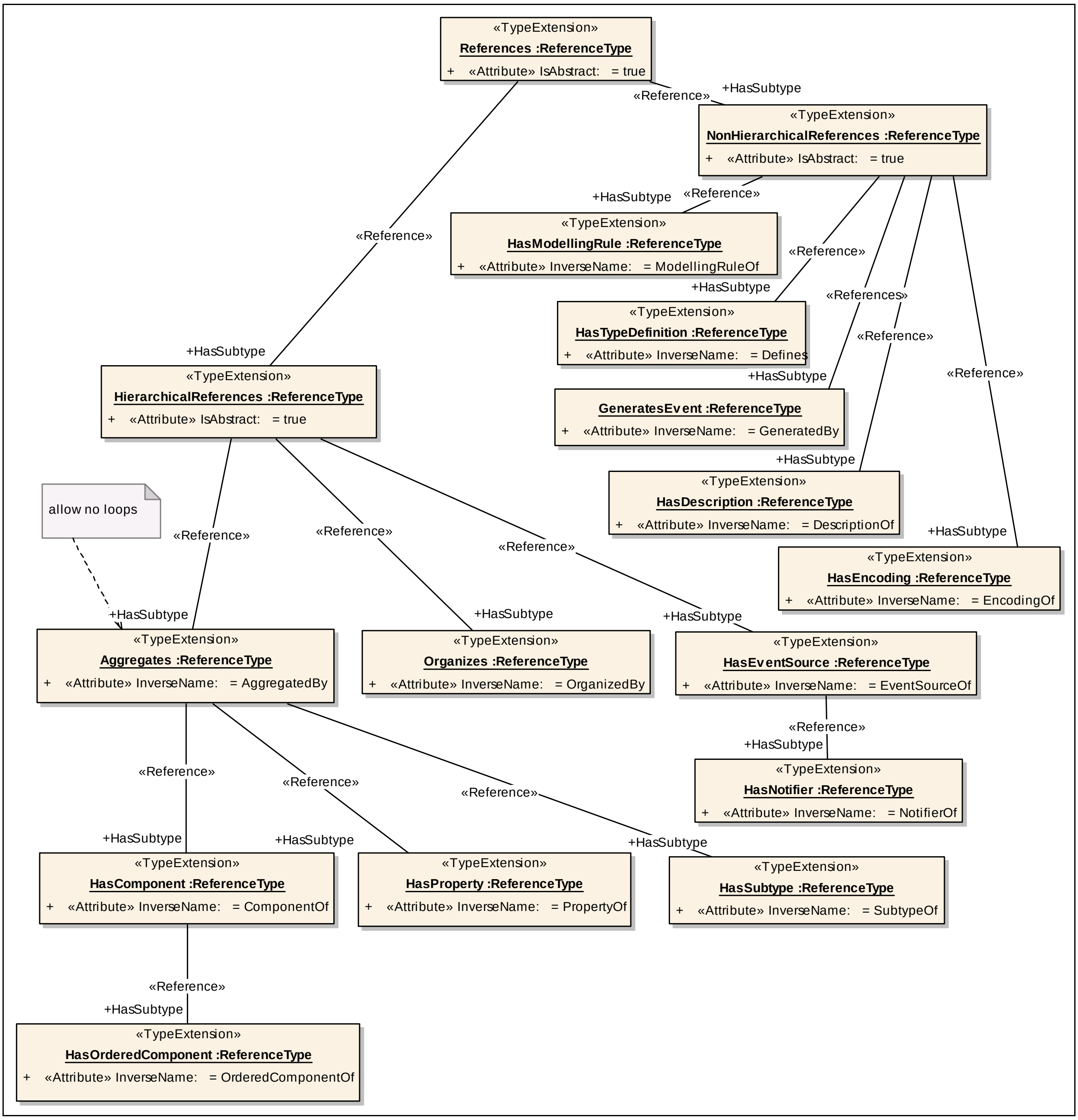

Figure 25 – Predefined ReferenceTypes

B.3.4 Attributes

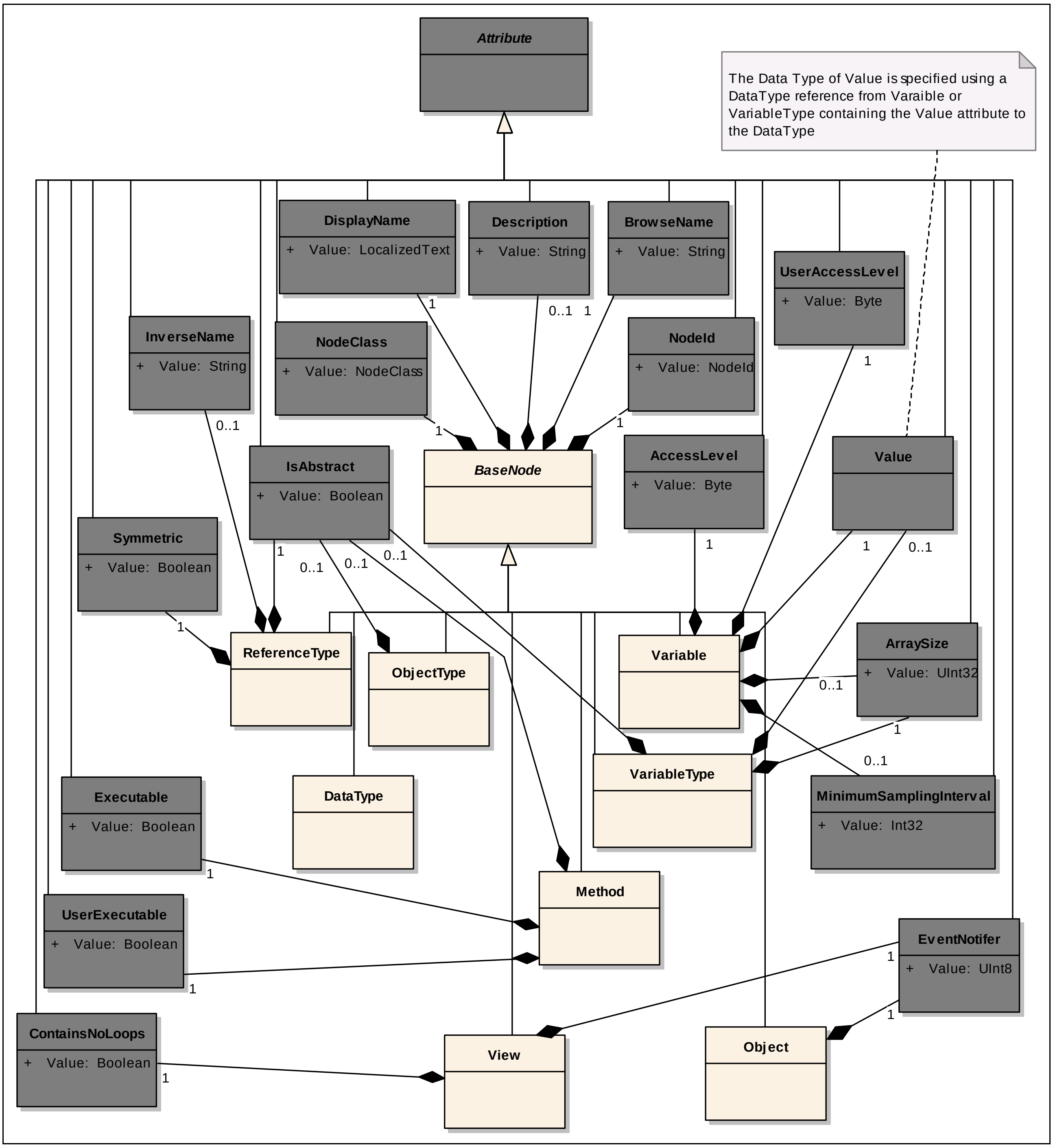

Figure 26 – Attributes

There may be more *Attributes* defined in other parts of the standard.

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

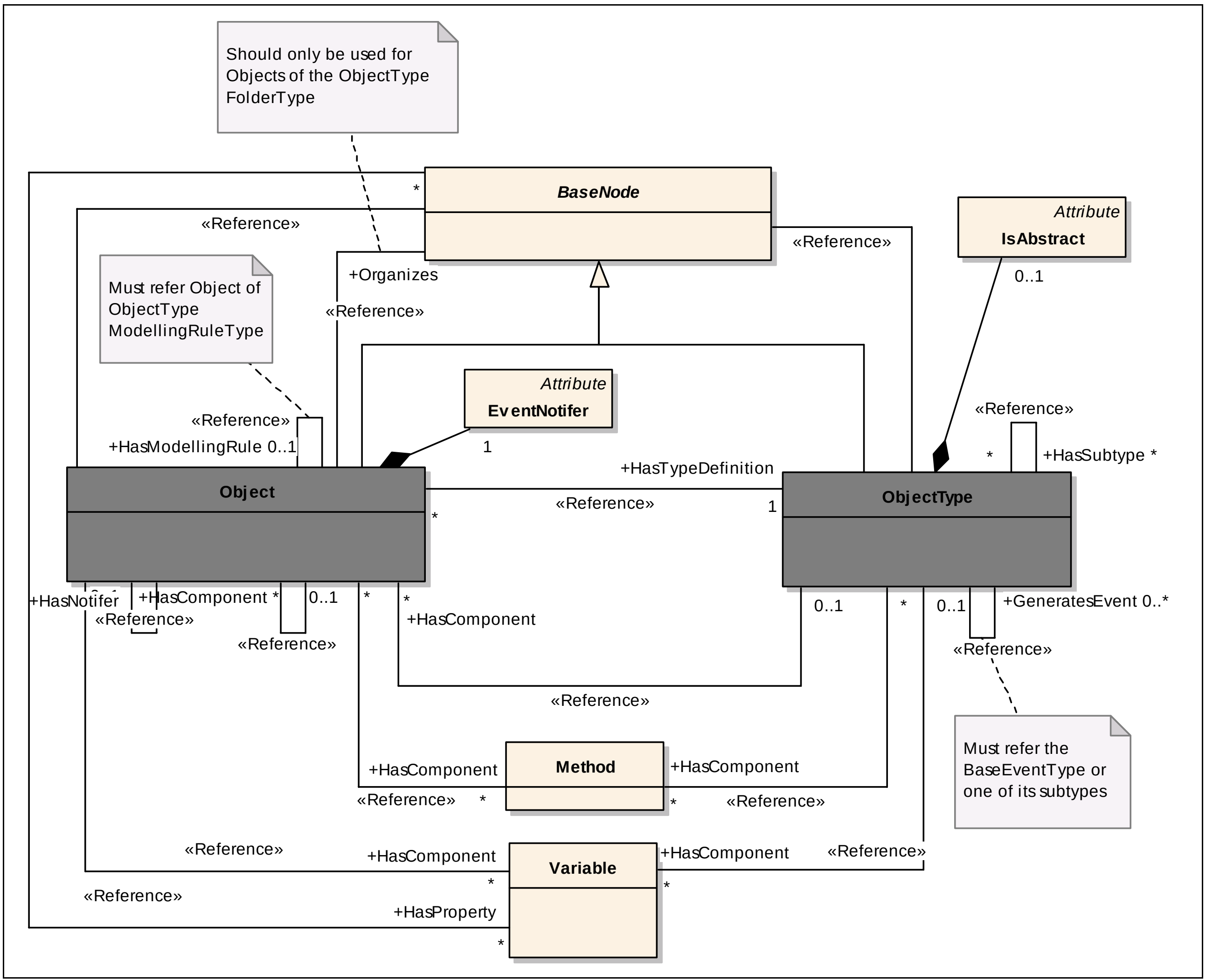

Figure 27 – Object and ObjectType

B.3.6 Variable and VariableType

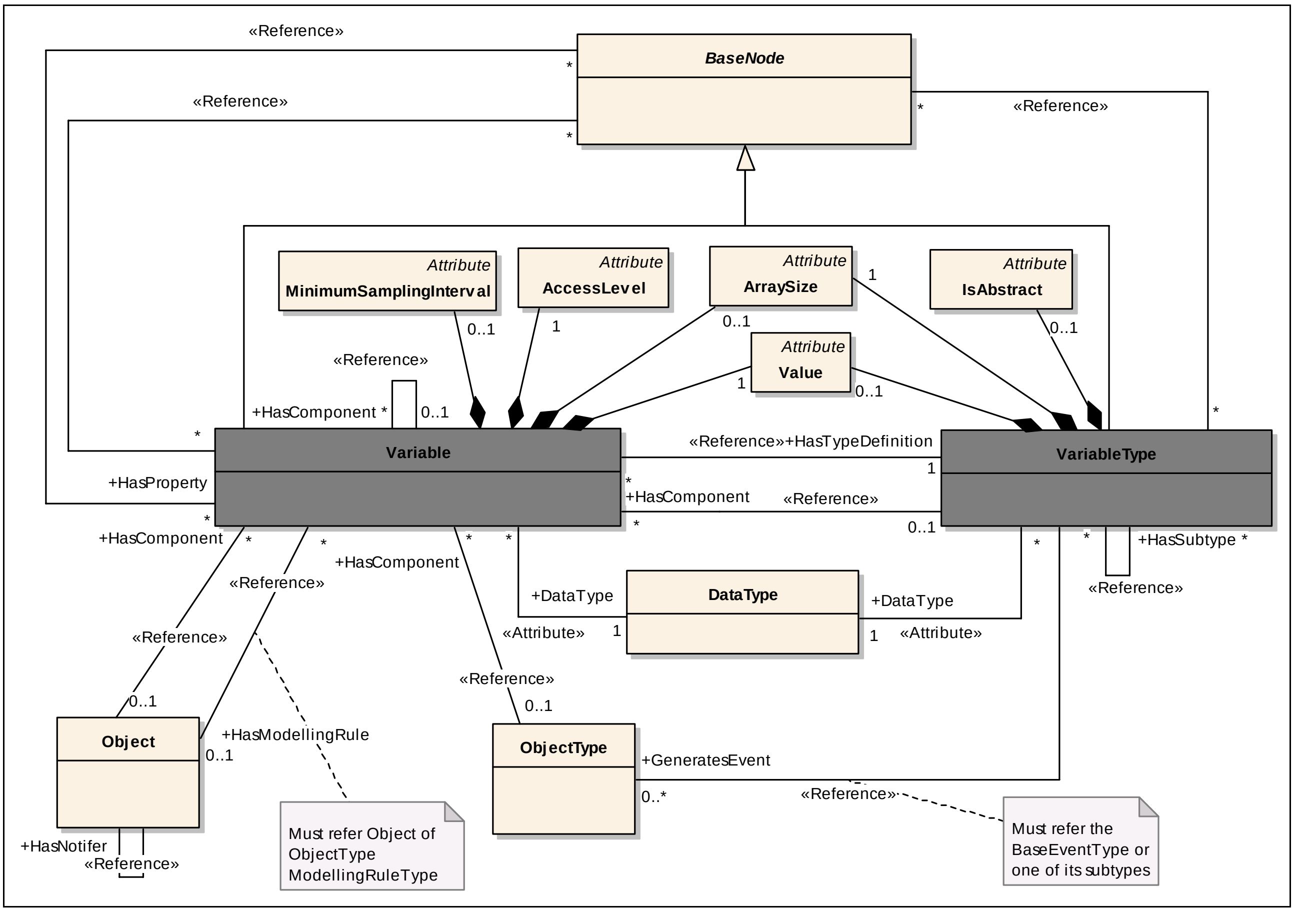

Figure 28 – Variable and VariableType

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

If a *HasProperty* points to a *Variable* from a *BaseNode* “A” the following constraints apply:

The *Variable* must not be the *SourceNode* of a *HasProperty* or any other *HierarchicalReferences Reference*.

All *Variables* having “A” as the *SourceNode* of a *HasProperty Reference* must have a unique *BrowseName* in the context of “A”.

B.3.7 Method

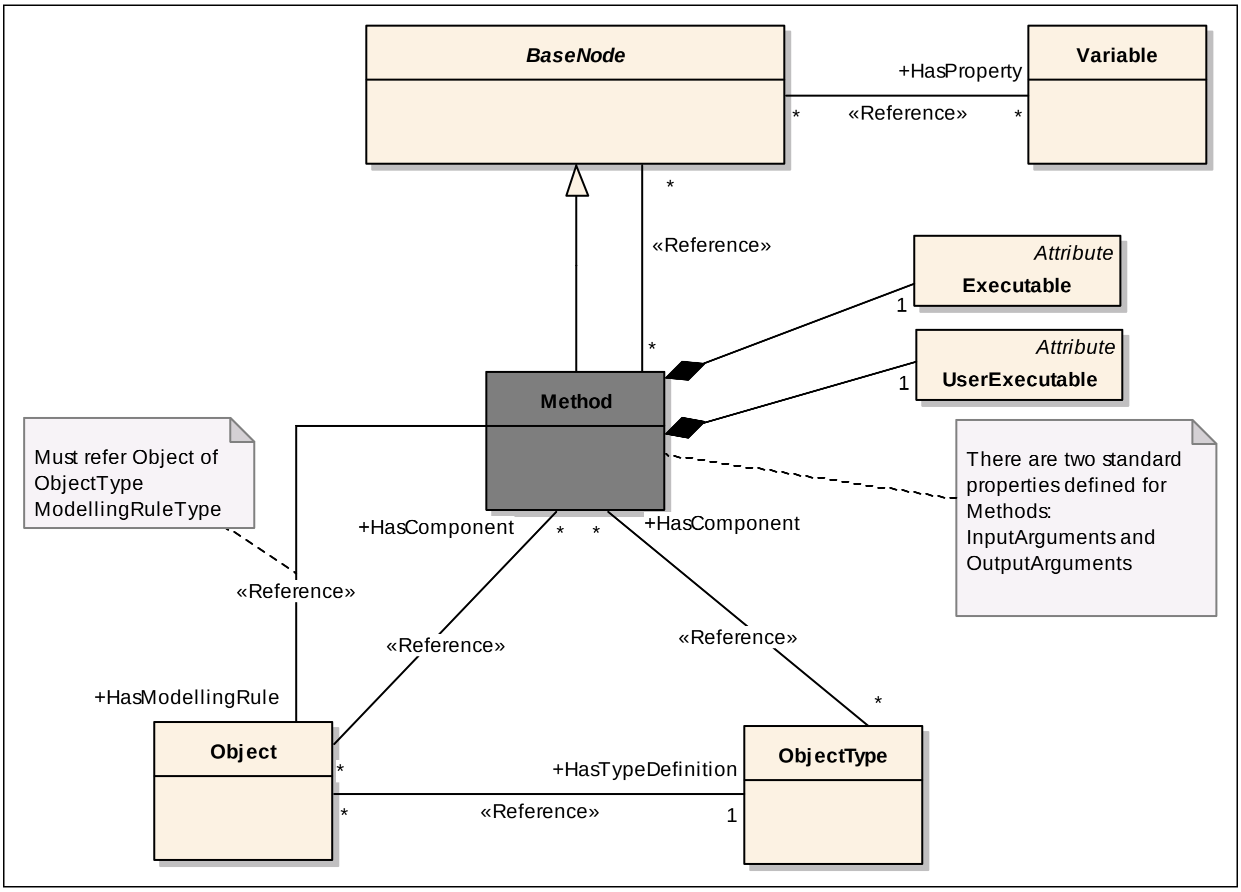

Figure 29 – Method

B.3.8 EventNotifier

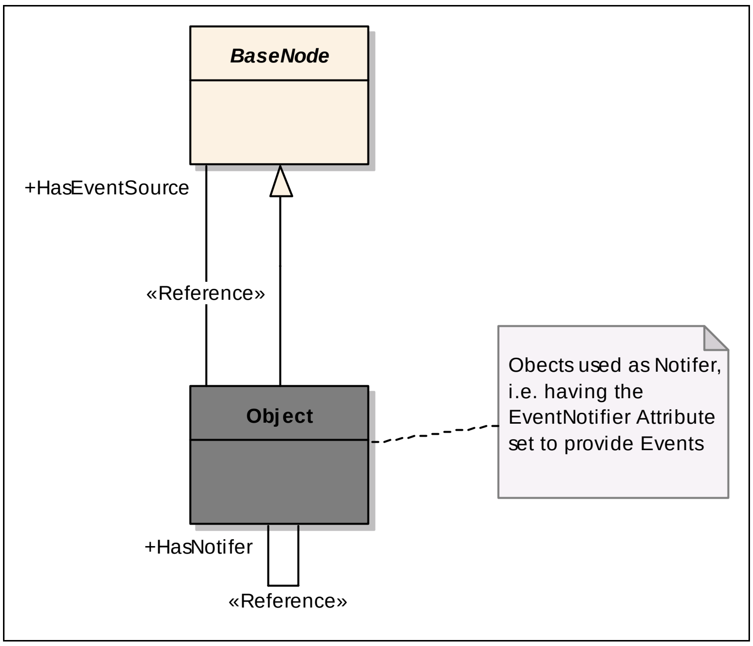

Figure 30 – EventNotifier

B.3.9 DataType

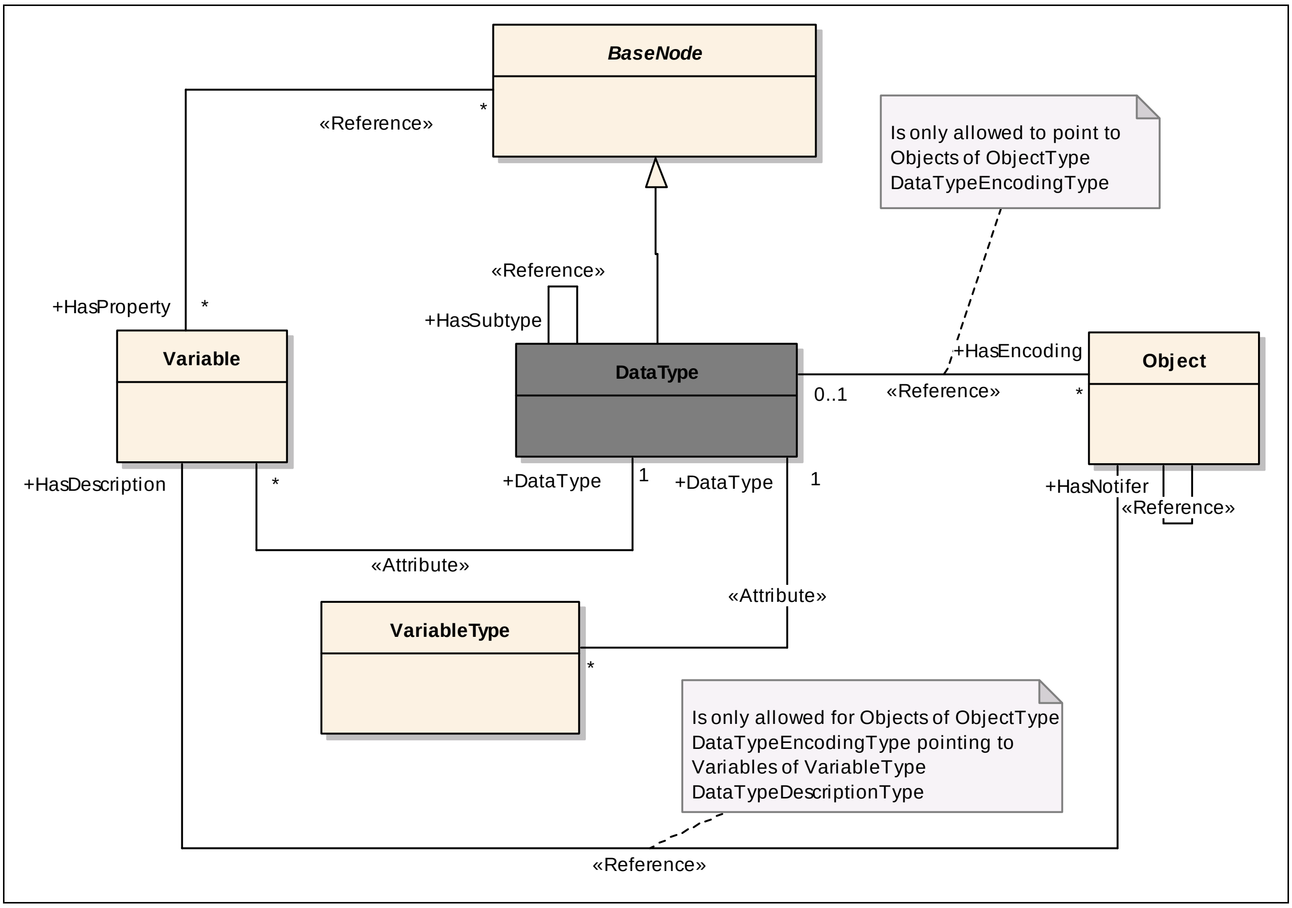

Figure 31 – DataType

B.3.10 View

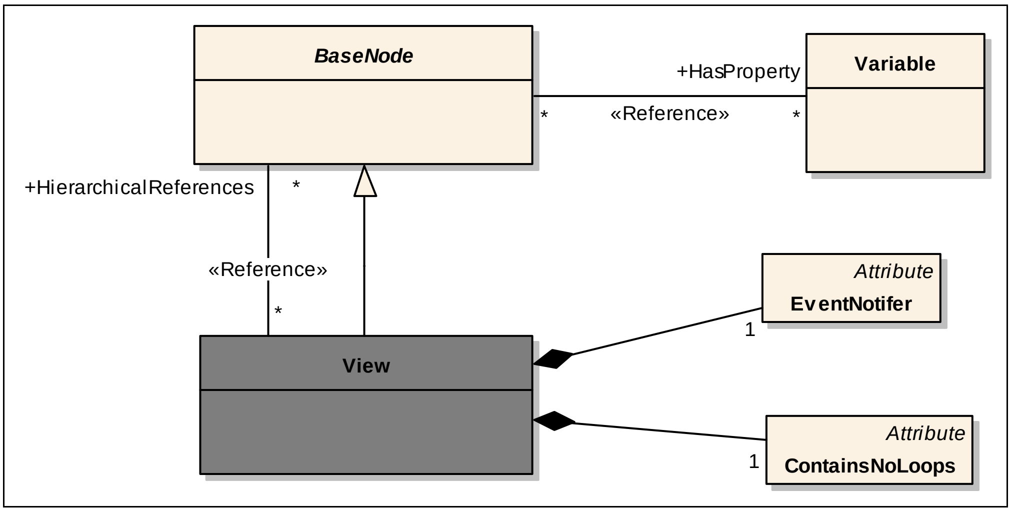

Figure 32 – View

Appendix C: OPC Binary Type Description System

C.1 Concepts

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

Figure 33 below illustrates the structure of an OPC Binary *TypeDictionary*.

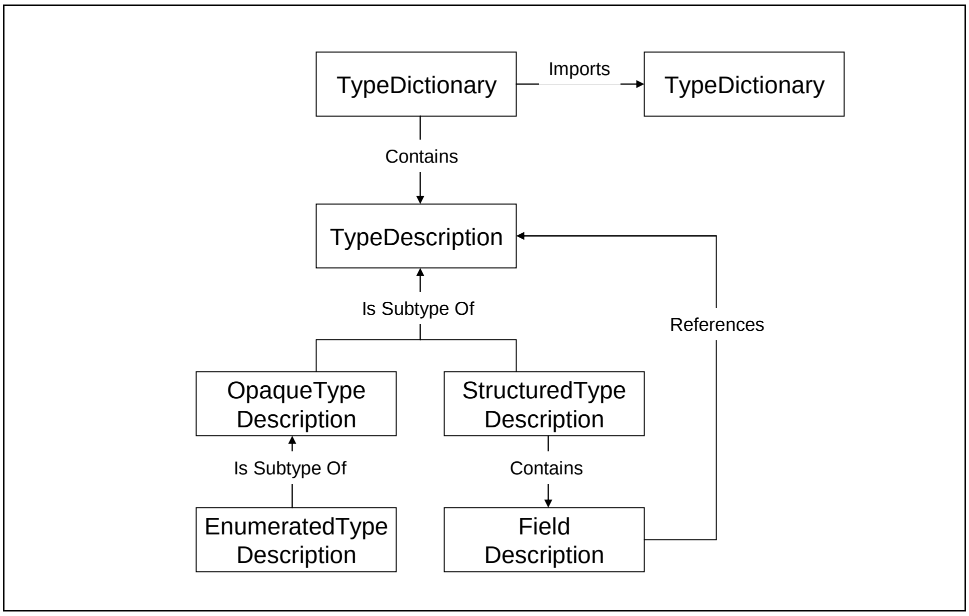

Figure 33 – OPC Binary Dictionary Structure

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 *OpaqueTypeDescription*. An instance of one of these building blocks is a binary-encoded value.

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

In some cases, the binary encoding described by an *OpaqueTypeDescription* 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 *OpaqueTypeDescription.* If authors of *TypeDictionaries* need to define new *OpaqueTypeDescriptions* 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 *StructuredTypeDescription* breaks a complex value into a sequence of values that are described by a *FieldDescription*. Each *FieldDescriptions* 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 *FieldDescription* is described completely in Clause C.2.6.

An *EnumeratedTypeDescription* describes a numeric value that has a limited set of possible values, each of which has a descriptive name. *EnumeratedTypeDescriptions* 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 20

Table 20 – TypeDictionary Components

Name	**Type**	**Description**
Documentation	Documentation	An element that contains human-readable text and XML that provides an overview of what is contained in the dictionary.
Import	ImportDirective[]	Zero or more elements that specify other *TypeDictionaries* that are referenced by *StructuredTypeDescriptions* defined in the dictionary. Each import element specifies the *NamespaceURI* of the *TypeDictionary* being imported. The *TypeDictionary* element must declare an XML namespace prefix for each imported namespace.
TargetNamespace	xs:string	Specifies the URI that qualifies all *TypeDescriptions* defined in the dictionary.
DefaultByteOrder	ByteOrder	Specifies the default *ByteOrder* for all *TypeDescriptions* that have the *ByteOrderSignificant* attribute set to “true”. This value overrides the setting in any imported *TypeDictionary*. This value is overridden by the *DefaultByteOrder* specified on a *TypeDescription*.
TypeDescription	TypeDescription[]	One or more elements that describe the structure of a binary encoded value. A TypeDescription is an abstract type. A dictionary may only contain the *OpaqueTypeDescription*, *EnumeratedTypeDescription* and *StructuredTypeDescription* 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 21

Table 21 – 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 *TypeDescription* that is unique within the dictionary. The fields of structured types reference *TypeDescriptions* by using this name qualified with the dictionary namespace URI.
DefaultByteOrder	ByteOrder	An attribute that specifies the default *ByteOrder* for the type description. This value overrides the setting in any *TypeDictionary* or in any *StructuredTypeDescription* that references the type description.

C.2.3 OpaqueTypeDescription

An *OpaqueTypeDescription* 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 *OpaqueTypeDescriptions* 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 *OpaqueTypeDescription.* The OPC Binary Type Description System defines many standard *OpaqueTypeDescriptions* that should allow authors to describe most binary encoded values as *StructuredTypeDescriptions*.

The components of an *OpaqueTypeDescription* are described in Table 22.

Table 22 – OpaqueTypeDescription Components

Name	Type	Description
TypeDescription	TypeDescription	An *OpaqueTypeDescription* inherits all elements and attributes defined for a *TypeDescription* in Table 21.
LengthInBits	xs:string	An attribute which specifies the length of the *OpaqueTypeDescription* in bits. This value should always be specified. If this value is not specified the *Documentation* element should describe the encoding in a way that a human understands.
ByteOrderSignificant	xs:boolean	An attribute that indicates whether byte order is significant for the type. If byte order is significant then the application must determine the byte order to use for the current context before interpreting the encoded value. The application determines the byte order by looking for the *DefaultByteOrder* attribute specified for containing *StructuredTypeDescriptions* or the *TypeDictionary*. If StructuredTypeDescriptions are nested the inner *StructuredTypeDescriptions* override the byte order of the outer descriptions. If the *DefaultByteOrder* attribute is specified for the *OpaqueTypeDescription*, then the *ByteOrder* is fixed and does not change according to context. If this attribute is “true”, then the *LengthInBits* attribute must be specified and it must be an integer multiple of 8 bits.

C.2.4 EnumeratedTypeDescription

An *EnumeratedTypeDescription* describes a binary-encoded numeric value that has a fixed set of valid values. The encoded binary value described by an *EnumeratedTypeDescription* 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 *EnumeratedTypeDescription* are described in Table 23.

Table 23 – EnumeratedTypeDescription Components

Name	Type	Description
OpaqueTypeDescription	OpaqueTypeDescription	An *EnumeratedTypeDescription* inherits all elements and attributes defined for a *TypeDescription* in Table 21 and for an *OpaqueTypeDescription* defined in Table 22. The *LengthInBits* attribute must always be specified.
EnumeratedValue	EnumeratedValue	One or more elements that describe the possible values for the instances of the type.

C.2.5 StructuredTypeDescription

A *StructuredTypeDescription* 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 *StructuredTypeDescription*.

Authors of type dictionaries should use *StructuredTypeDescriptions* 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 must be fixed and it must be a multiple of 8 bits. Any field which does not have a fixed length must 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 *StructuredTypeDescription* are described in Table 24.

Table 24 – StructuredTypeDescription Components

Name	Type	Description
TypeDescription	TypeDescription	A *StructuredTypeDescription* inherits all elements and attributes defined for a *TypeDescription* in Table 21.
Field	FieldDescription	One or more elements that describe the fields of the structure. Each field must have a name that is unique within the *StructuredTypeDescription*. Some fields may reference other fields in the *StructuredTypeDescription* by using this name.
anyAttribute	*	Authors of a *TypeDictionary* may add their own attributes to any *StructuredTypeDescription* that must 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 FieldDescription

A *FieldDescription* describes a binary encoded value that appears in sequence within a *StructuredTypeDescription*. Every *FieldDescription* must reference a *TypeDescription* that describes the encoded value for the field.

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

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

The components of a *FieldDescription* are described in Table 25.

Table 25 – FieldDescription 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 *Field* that is unique within the *StructuredTypeDescription*. Other fields in the structured type reference a *Field* by using this name.
TypeName	xs:QName	An attribute that specifies the *TypeDescription* that describes the contents of the field. A field may contain zero or more instances of this type depending on the settings for the other attributes and the values in other fields
Length	xs:unsignedInt	An attribute that indicates length of the field. This value may be the total number of encoded bytes or it may be the number of instances of the type referenced by the field. The *IsLengthInBytes* attributes specifies which of these definitions applies.
LengthField	xs:string	An attribute that indicates which other field in the *StructuredTypeDescription* specifies the length of the field. The length of the field may be in bytes or it may be the number of instances of the type referenced by the field. The *IsLengthInBytes* attributes specifies which of these definitions applies. If this attribute refers to a field that is not present in an encoded value, then the default value for the length is 1. This situation could occur if the field referenced is an optional field (see the *SwitchField* attribute). The length field must 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 *FieldDescription* referenced by this attribute must precede the field with the *StructuredTypeDescription*.
IsLengthInBytes	xs:boolean	An attribute that indicates whether the *Length* or *LengthField* attributes specify the length of the field in bytes or in the number of instances of the type referenced by the field.
SwitchField	xs:string	If this attribute is specified, then the field is optional and many not appear in every instance of the encoded value. This attribute specifies the name of another *Field* that controls whether this field is present in the encoded value. The field referenced by this attribute must be an integer value (see the *LengthField* attribute). The current value of the switch field is compared to the *SwitchValue* attribute using the *SwitchOperand.* If the condition evaluates to true then the field appears in the stream. If the *SwitchValue* attribute is not specified, then this field is present if the value of the switch field is non-zero. The *SwitchOperand* field is ignored if it is present. If the *SwitchOperand* attribute is missing, then the field is present if the value of the switch field is equal to the value of the *SwitchValue* attribute*.* The *Field* referenced by this attribute must precede the field with the *StructuredTypeDescription*.
SwitchValue	xs:unsignedInt	This attribute specifies when the field appears in the encoded value. The value of the field referenced by the *SwitchName* attribute is compared using the SwitchOperand attribute to this value. The field is present if the expression evaluates to true. The field is not present otherwise.
SwitchOperand	xs:string	This attribute specifies how the value of the switch field should be compared to the switch value attribute. This field is an enumeration with the following values: Equal — *SwitchField* is equal to the *SwitchValue*. GreaterThan — *SwitchField* is greater than the *SwitchValue*. LessThan — *SwitchField* is less than the *SwitchValue*. GreaterThanOrEqual — *SwitchField* is greater than or equal to the *SwitchValue*. LessThanOrEqual — *SwitchField* is less than or equal to the *SwitchValue*. NotEqual — *SwitchField* is not equal to the *SwitchValue*. In each case the field is present if the expression is true.
Terminator	xs:hexBinary	This attribute indicates that the field contains one or more instances of *TypeDescription* referenced by this field and that the last value has the binary

		encoding specified by the value of this attribute. If this attribute is specified then the *TypeDescription* referenced by this field must either have a fixed byte order (i.e. byte order is not significant or explicitly specified) or the containing StructuredTypeDescription must explicitly specify the byte order. Examples: Field Data Type / Terminator / Byte Order / Hexadecimal String Char / tab character / not applicable / 09 WideChar: / tab character / BigEndian / 0009 WideChar: / tab character / LittleEndian / 0900 Int16 / 1 / BigEndian / 0001 Int16 / 1 / LittleEndian / 0100
anyAttribute	*	Authors of a *TypeDictionary* may add their own attributes to any *FieldDescription* which must be qualified with a namespace defined by the authors. Applications should not be required to understand these attributes in order to interpret a binary encoded field value.

C.2.7 EnumeratedValueDescription

An *EnumeratedValueDescription* describes a possible value for an *EnumeratedTypeDescription*.

The components of an *EnumeratedValue* are described in Table 26.

Table 26 – EnumeratedValueDescription 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 27.

Table 27 – ImportDirective Components

Name	Type	Description
Namespace	xs:string	This attribute specifies the *TargetNamespace* for the *TypeDictionary* being imported. This may be a well-known URI which means applications need not have access to the physical file to recognise types that are referenced.
Location	xs:string	This attribute specifies the physical location of the XML file containing the *TypeDictionary* to import. This value could be a URL for a network resource, a NodeId in a 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 *StructuredTypeDescription*. The standard type descriptions are described in

Table 28 – 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:OpaqueTypeDescription Name="Int128" LengthInBits="128">
  <opc:Documentation>A 128-bit signed integer.</opc:Documentation>
</opc:OpaqueTypeDescription>
```

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

```
<opc:StructuredTypeDescription 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:StructuredTypeDescription>
```

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

3. A structured type with optional fields.

```
<opc:StructuredTypeDescription 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:StructuredTypeDescription>
```

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

4. An array of integers.

```
<opc:StructuredTypeDescription 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:StructuredTypeDescription>
```

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

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

```
<opc:StructuredTypeDescription 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:StructuredTypeDescription>
```

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

6. A simple union.

```
<opc:StructuredTypeDescription 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:StructuredTypDescriptione>
```

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

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

7. An enumerated type.

```
<opc:EnumeratedTypeDescription 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:EnumeratedTypeDescription>
```

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

8. A nillable array.

```
<opc:StructuredTypeDescription 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:StructuredTypDescription>
```

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" ?>
```

```xml
<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="OpaqueTypeDescription">
    <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="EnumeratedValueDescription">
    <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="EnumeratedTypeDescription">
    <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="FieldDescription">
    <xs:sequence>
      <xs:element ref="Documentation" minOccurs="0" maxOccurs="1" />
    </xs:sequence>
    <xs:attribute name="Name" type="xs:string" use="required" />
```

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

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

</xs:schema>
```

C.6 OPC Binary Standard TypeDictionary

```xml
<?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:OpaqueTypeDescription Name="Bit" LengthInBits="1">
    <opc:Documentation>A single bit.</opc:Documentation>
  </opc:OpaqueTypeDescription>

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

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

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

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

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

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

```xml
  </opc:OpaqueTypeDescription>

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

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

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

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

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

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

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

  <opc:StructuredTypeDescription 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:StructuredTypeDescription>

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

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

  <opc:StructuredTypeDescription 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:StructuredTypeDescription>

  <opc:StructuredTypeDescription 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:StructuredTypeDescription>

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

  <opc:StructuredTypeDescription 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:StructuredTypeDescription>

</opc:TypeDictionary>
```