OPC UA Solution .NET Introduction

Develop OPC UA Clients and OPC UA Servers
with C# for .NET

Introduction

T

Document Control

Version	Date	Comment
1.0.8	20-MAY-2019	Initial version based on version 1.0.8
1.0.9	31-MAY-2019	Updated chapter 2
1.1.0	09-AUG-2019	Changed references to new OPC UA specification structure
1.2.0	17-NOV-2019	- Started chapter 5 about the Standard Noce Classes - Updated several chapters to OPC UA 1.04 - Changed to .NET 4.6.2 - Removed OPC UA Client Gateway (no longer supported)
1.3.0	05-JAN-2020	- Added informationen for .NET Core 2.1 - Added information for .NET 4.8
1.4.0	24-JUN-2020	Updated to version 1.4
1.4.10	27-SEP-2020	Updated general information
2.0	31-OCT-2020	Updated to version 2.0
2.1	21-DEC-2020	Updated to .NET 5.0
2.3	23-APR-2022	Updated to V2.3
2.6	17-DEC-2022	Updated to V2.6

Purpose and audience of document

Microsoft's .NET is an application development environment that supports multiple languages and provides a large set of standard programming APIs. This document defines an Application Programming Interface (API) for OPC UA Client and Server development based on the .NET programming model.

The OPC UA specification can be downloaded from the web site of the OPC Foundation. But only **Error! R eference source not found.** (Overview and Concepts) is available to the public. All other parts can only be downloaded from OPC Foundation members and may be used only if the user is an active OPC Foundation member. Because of this fact the OPC UA .NET API hides most of the OPC UA specifications to provide the possibility to delevlop OPC UA Clients and OPC UA Servers in the .NET 5.0 / .NET Core 3.1 environment without the need to be an OPC Foundation member. The API does support OPC Unified Architecture.

This document is intended as reference material for developers of OPC UA compliant Client and Server applications. It is assumed that the reader is familiar with the Microsoft's .NET 5.0 / .NET Core 3.1 and the needs of the Process Control industry.

Summary

This document gives a short overview of the functionality of the OPC UA .NET solutions family. The goal of this document is to give an introduction and can be used as base for your own implementations

T

Referenced OPC Documents

Documents
Online versions of OPC UA specifications and information models. The OPC UA Online Reference is available at: https://reference.opcfoundation.org
OPC Unified Architecture Textbook, written by Wolfgang Mahnke, Stefan-Helmut Leitner and Matthias Damm: http://www.amazon.com/OPC-Unified-Architecture-Wolfgang-Mahnke/dp/3540688986/ref=sr_1_1?ie=UTF8&s=books&qid=1209506074&sr=8-1

T

Table of Contents

T

T

T

T

Disclaimer

Trademark Notice

T

1 Overview

If your application requires access to the OPC technology, you must decide which of the OPC Specifications you want to support. The decision depends on many factors like the used OPC specifications supported by third party solutions or the system architcture you want to use. Technosoftware GmbH is specialized in software consulting and development services for technical and industrial applications based on OPC and the founder of Technosoftware GmbH was involved in the design and development of many software solutions with OPC connectivity with more than 22 years extensive knowledge about OPC.

The OPC Unified Architecture (UA) is **THE** next generation OPC standard that provides a cohesive, secure and reliable cross platform framework for access to real time and historical data and events.

It's **time to adapt** this specification for use in your applications with keeping in mind that you may need to support other OPC specifications as well. The Classic OPC specifications are widely used and it is important that your application can support these specifications. We recommend that you design your application to use the OPC Unified Architecture in the first place, with the option to have access to the Classic OPC Specifications in the second place. A short overview of the mainly used Classic OPC Specifications should give you an understanding of the requirements your application design have to fullfill.

1.1 OPC Technology

1.1.1 Classic OPC Specifications

The Classic OPC specifications are DCOM based specifications like OPC Data Access (DA), OPC Alarms&Events (AE) and OPC Historical Access (HDA). Each of these specifications is a separate specification and several more exists.

According to the different requirements within industrial applications, three major Classic OPC specifications have been developed. Support of one or more of these specifications should be provided by most of the OPC Client or OPC Server applications.

1.1.1.1 Data Access (DA)

An OPC DA Server allows OPC DA Clients to retrieve information about several objects: the server, the group and the items.

- ↗ The OPC server object maintains information about the server and acts as a container for OPC group objects.

- ↗ The OPC group object maintains information about itself and provides the mechanism for containing and logically organizing OPC items.

- ↗ The OPC items represent connections to data sources within the server.

The OPC DA Specification defines two read/write interfaces:

- ↗ Synchronous
  The client can perform a synchronous read from cache (simple and reasonably efficient). This may be appropriate for simple clients that are reading relatively small amounts of data.

- ↗ Asynchronous
  The client can 'subscribe' to cached data using IAdviseSink or IOPCDataCallback which is more complex but very efficient. Asynchronous access is recommended because it minimizes the use of CPU and NETWORK resources.

In all cases the OPC DA Server gives the client access to current values of the OPC items. The OPC DA Server only holds current information in cache. Old information is overwritten. As a result of this it cannot be guaranteed that an OPC DA Client retrieves all changes in values (also not in asynchronous mode).

T

For such cases, there exist two more OPC specifications, the OPC Alarms&Events and the OPC Historical Data Access Specification.

1.1.1.2   Alarms&Events (AE)

The OPC AE interface provides a mechanism for OPC AE clients to be notified when a specified event and/or alarm condition occurs. The browser interface also allows OPC AE clients to determine the list of events and conditions supported by an OPC AE Server as well as to get their current status.

Within OPC, an alarm is an abnormal condition and is thus a special case of a condition. A condition is a named state of the OPC Event Server or of one of its contained objects that is of interest to an OPC AE client. For example, the tag Temperature may have the following conditions associated with it: HighAlarm, HighHighAlarm, Normal, LowAlarm, and LowLowAlarm.

On the other hand, an event is a detectable occurrence that is of significance to the OPC Server, the device it represents, and its OPC AE clients. An event may or may not be associated with a condition. For example, the transition into HighAlarm and Normal conditions are events, which are associated with conditions. However, operator actions, system configuration changes, and system errors are examples of events, which are not related to specific conditions. OPC AE clients may subscribe to be notified of the occurrence of specified events.

The OPC AE specification provides methods enabling the OPC AE client to:

- ↗ Determine the types of events that are supported by the OPC AE server.
- ↗ Enter subscriptions to specified events so that OPC AE clients can receive notifications of their occurrences. Filters may be used to define a subset of desired events.
- ↗ Access and manipulate conditions implemented by the OPC AE server.

1.1.1.3   Historical Data Access (HDA)

Historical engines today produce an added source of information that should be distributed to users and software clients that are interested in this information. Currently most historical systems use their own proprietary interfaces for dissemination of data. There is no capability to augment or use existing historical solutions with other capabilities in a plug-n-play environment. This requires the developer to recreate the same infrastructure for their solutions, as all other vendors have had to develop independently with no interoperability with any other systems.

In keeping with the desire to integrate data at all levels of business, historical information can be another type of data.

There are several types of Historian servers. Some key types supported by the HDA specification are:

- ↗ Simple Trend data servers.

These servers provided little else then simple raw data storage. (Data would typically be the types of data available from an OPC Data Access server, usually provided in the form of a duple [Time Value & Quality])

- ↗ Complex data compression and analysis servers. These servers provide data compression as well as raw data storage. They can provide summary data or data analysis functions, such as average values, minimums and maximums etc. They can support data updates and history of the updates. They can support storage of annotations along with the actual historical data storage.

1.1.2 OPC XML-DA Specification

The OPC XML-DA specification is a web services-based specification and is a step between Classic OPC and OPC Unified Architecture. The functionality is restricted to OPC DA.

Technosoftware GmbH does not offer any solutions supporting the OPC XML-DA Specification.

1.1.3 OPC .NET 4.0 (WCF)

OPC .NET 4.0 (WCF) is not a specification like the others mentioned here, it bridges the gap between Microsoft.NET and the world of Classic OPC. OPC .NET 4.0 is an OPC standard C# Application Programming Interface (API) designed to simplify client access to OPC Classic servers. It also includes a formal WCF Interface; however, there is no compliance or certification program for this interface. The Classic OPC interfaces are still the primary means to ensure multi-vendor interoperability.

Technosoftware GmbH does not offer any solutions supporting the OPC .NET 4.0 API.

T

1.1.4 OPC Unified Architecture Specification

OPC Unified Architecture (UA) is a platform-independent standard through which various kinds of systems and devices can communicate by sending Messages between Clients and Servers over various types of networks. It supports robust, secure communication that assures the identity of Clients and Servers and resists attacks. OPC UA defines standard sets of Services that Servers may provide, and individual Servers specify to Clients what Service sets they support. Information is conveyed using standard and vendor- defined data types, and Servers define object models that Clients can dynamically discover. Servers can provide access to both current and historical data, as well as Alarms and Events to notify Clients of important changes. OPC UA can be mapped onto a variety of communication protocols and data can be encoded in various ways to trade off portability and efficiency.

The OPC Foundation provides deliverables for its member companies. These include a .NET based OPC UA Stack, an ANSI C based OPC UA Stack and a Java based OPC UA Stack. The .NET based OPC UA Stack is the base of all Technosoftware GmbH's .NET based solutions.

Technosoftware GmbH offers the following solutions supporting the OPC Unified Architecture Specification:

↗ **OPC UA Client .NET**
The OPC UA Client .NET offers a fast and easy access to the OPC UA Client technology. Develop OPC UA 1.00, 1.01, 1.02, 1.03 and 1.04 compliant UA Clients with C#/VB.NET targeting .NET 6.0, .NET 5.0, .NET Core 3.1 or .NET Standard 2.1.

↗ **OPC UA Server .NET**
The OPC UA Server .NET offers a fast and easy access to the OPC Unified Architecture (UA) technology. Develop OPC UA 1.00, 1.01, 1.02, 1.03 and 1.04 compliant Servers with C#/VB.NET targeting .NET 6.0, .NET 5.0, .NET Core 3.1 or .NET Standard 2.1.

T

2 OPC UA Solutions .NET

The OPC UA Solution .NET offers a fast and easy access to the OPC UA Client & Server technology. Develop OPC compliant UA Clients and Servers with C# targeting .NET7.0, .NET 6.0 or .NET Core 3.1. For backward compatibility we also provide .NET 4.8, .NET 4.7.2 and .NET 4.6.2 assemblies.
You can download it from https://github.com/technosoftware-gmbh/opcua-solution-net-samples

This GitHub repository is automatically tested with the following environments:

    a. Linux Ubuntu 22.04
        i.   .NET 6.0.404
        ii. Mono 6.12
        iii. MS Build 17.4.0
    b. Mac OS X 11
        i.   .NET 6.0.404
        ii. Mono 6.12
        iii. MS Build 17.4.0
    c. Windows Server 202
        i.   .NET 6.0.404
        ii. MS Build 17.4.0

Important:

An installation guide is available with the solution. Please read that one first and then follow this guide.

2.1 .NET based Solutions

2.1.1 OPC UA Client .NET

The OPC UA Client .NET offers a fast and easy access to the OPC Unified Architecture (UA) technology. Develop OPC UA compliant Clients with C# targeting .NET 7, .NET 6.0 or .NET Core 3.1. For backward compatibility we also provide .NET 4.8, .NET 4.7.2 and .NET 4.6.2 support.

.NET 7, .NET 6 and .NET Core 3.1 allows you develop applications that run on all common platforms available today, including Linux, macOS and Windows 8.1/10/11 (including embedded/IoT editions) without requiring platform-specific modifications.

The OPC Unified Architecture (UA) is THE next generation OPC standard that provides a cohesive, secure and reliable cross platform framework for access to real time and historical data and events.

It's Time to Adapt this specification for use in your applications and we recommend considering designing your application to use the OPC Unified Architecture.

The OPC UA Client .NET API defines classes which can be used to implement an OPC client capable to access OPC servers supporting different profiles with the same API. These classes manage client side state information; provide higher level abstractions for OPC tasks such as managing sessions and subscriptions or saving and restoring connection information for later use.

T
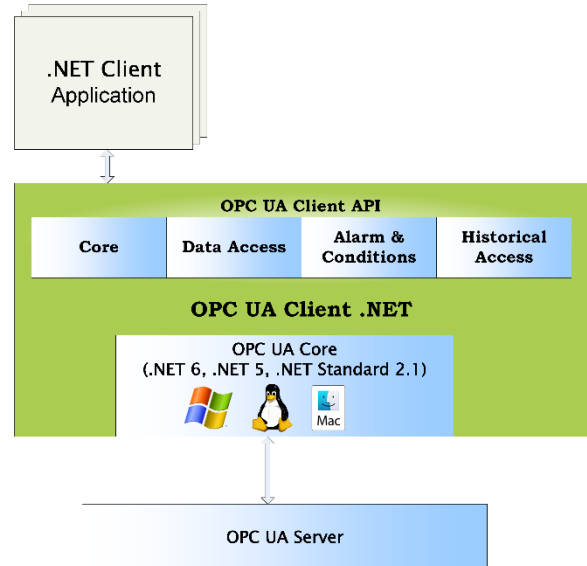
2.1.2   OPC UA Server .NET

The OPC UA Server NET offers a fast and easy access to the OPC Unified Architecture (UA) technology. Develop OPC UA compliant Servers with C# targeting .NET 7, .NET 6.0 or .NET Core 3.1. For backward compatibility we also provide .NET 4.8, .NET 4.7.2 and .NET 4.6.2 support.

.NET 7, .NET 6 and .NET Core 3.1 allows you develop applications that run on all common platforms available today, including Linux, macOS and Windows 8.1/10/11 (including embedded/IoT editions) without requiring platform-specific modifications.

The OPC Unified Architecture (UA) is THE next generation OPC standard that provides a cohesive, secure and reliable cross platform framework for access to real time and historical data and events.

It's Time to Adapt this specification for use in your applications and we recommend considering designing your application to use the OPC Unified Architecture.

We recommend considering designing your application to use the OPC Unified Architecture in the first place by using our new **OPC UA Server .NET** toolkit which allows development of server applications supporting OPC UA.

The developer can concentrate on his application and servers can be developed fast and easily without the need to spend a lot of time learning how to implement the OPC Unified Architecture specification. The server API is easy to use and many OPC specific functions are handled by the framework.

The included Model Compiler can be used to create the necessary C# classes of Information Model's specified in XML and CSV based files. The XML files must be edited by a text editor.
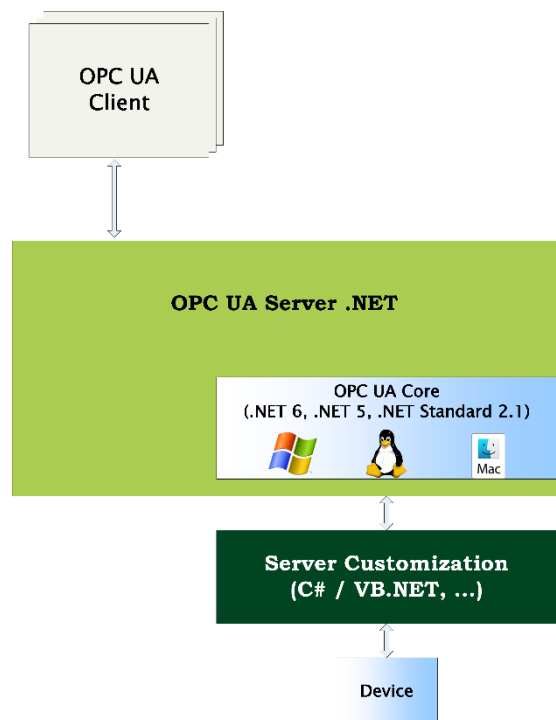
3 Introduction

OPC Unified Architecture (UA) is a platform-independent standard through which various kinds of systems and devices can communicate by sending Messages between Clients and Servers over various types of networks. It supports robust, secure communication that assures the identity of Clients and Servers and resists attacks. OPC UA defines standard sets of Services that Servers may provide, and individual Servers specify to Clients what Service sets they support. Information is conveyed using standard and vendor-defined data types, and Servers define object models that Clients can dynamically discover. Servers can provide access to both current and historical data, as well as Alarms and Events to notify Clients of important changes. OPC UA can be mapped onto a variety of communication protocols and data can be encoded in various ways to trade off portability and efficiency.

OPC UA provides a consistent, integrated AddressSpace and service model. This allows a single OPC UA Server to integrate data, Alarms and Events, and history into its AddressSpace, and to provide access to them using an integrated set of Services. These Services also include an integrated security model.

OPC UA also allows Servers to provide Clients with type definitions for the Objects accessed from the AddressSpace. This allows standard information models to be used to describe the contents of the AddressSpace. OPC UA allows data to be exposed in many different formats, including binary structures and XML documents. The format of the data may be defined by OPC, other standard organizations or vendors. Through the AddressSpace, Clients can query the Server for the metadata that describes the format for the data. In many cases, Clients with no pre-programmed knowledge of the data formats will be able to determine the formats at runtime and properly utilize the data.

OPC UA adds support for many relationships between Nodes instead of being limited to just a single hierarchy. In this way, an OPC UA Server may present data in a variety of hierarchies tailored to the way a set of Clients would typically like to view the data. This flexibility, combined with support for type definitions, makes OPC UA applicable to a wide array of problem domains. As illustrated below, OPC UA is not targeted at just the telemetry server interface, but also to provide greater interoperability between higher level functions.

OPC UA is designed to provide robustness of published data. A major feature of all OPC servers is the ability to publish data and Event Notifications. OPC UA provides mechanisms for Clients to quickly detect and recover from communication failures associated with these transfers without having to wait for long timeouts provided by the underlying protocols.

OPC UA is designed to support a wide range of Servers, from plant floor PLCs to enterprise Servers. These Servers are characterized by a broad scope of size, performance, execution platforms and functional capabilities. Therefore, OPC UA defines a comprehensive set of capabilities, and Servers may implement a subset of these capabilities. To promote interoperability, OPC UA defines standard subsets, referred to as Profiles, to which Servers may claim conformance. Clients can then discover the Profiles of a Server, and tailor their interactions with that Server based on the Profiles. Profiles are defined in [UA Part 7].
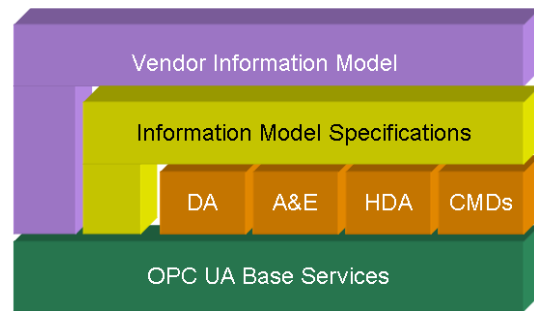
T

The OPC UA specifications are layered to isolate the core design from the underlying computing technology and network transport. This allows OPC UA to be mapped to future technologies as necessary, without negating the basic design. Mappings and data encodings are described in [UA Part 6]. Two data encodings are defined in this part:

- XML/text

- UA Binary

In addition, two transport mappings are defined in this part:

- TCP

- SOAP Web services over HTTP

Clients and Servers that support multiple transports and encodings will allow the end users to make decisions about tradeoffs between performance and XML Web service compatibility at the time of deployment, rather than having these tradeoffs determined by the OPC vendor at the time of product definition.

OPC UA is designed as the migration path for OPC clients and servers that are based on Microsoft COM technology. Care has been taken in the design of OPC-UA so that existing data exposed by OPC COM servers (DA, HDA and A&E) can easily be mapped and exposed via OPC UA. Vendors may choose to migrate their solutions natively to OPC UA or use external wrappers to convert from OPC COM to OPC UA and vice-versa. Each of the previous OPC specifications defined its own address space model and its own set of Services. OPC UA unifies the previous models into a single integrated address space with a single set of Services.

4 OPC UA Basics

This chapter describes the base concepts of the OPC UA specification. It is not intended to replace studying the OPC UA specification; its purpose is to introduce the base concepts of the OPC UA specification.

4.1 Overview

4.1.1 Specifications

The OPC UA specification is organized as a multi-part specification, splitted into 3 sections:

1. Core Specification Parts
   The first seven parts specify the core capabilities of OPC UA. These core capabilities define the structure of the OPC AddressSpace and the Services that operate on it.

2. Access Type Specification Parts
   Parts 8 through 11 apply these core capabilities to specific types of access previously addressed by separate OPC COM specifications, such as Data Access (DA), Alarms and Events (A&E) and Historical Data Access (HDA).

3. Utility Specification Parts
   Part 12: describes Discovery mechanisms for OPC UA and Part 13 describe ways of aggregating data. Part 14 defines the PubSub communication model. The PubSub communication model defines an OPC UA publish subscribe pattern instead of the client server pattern defined by the Services in Part 4.

Unfortunatley most of the OPC UA specification documents can only be downloaded by OPC Foundation members from the OPC Foundation web site. Only part 1 (Overview and Concepts) is available to the public. All other parts are available only to OPC Foundation members and may be used only if the user is an active OPC Foundation member.

The following sections give the reader an introduction in OPC UA. Most of these parts are summaries of the OPC UA specifications and are reduced to those parts which are required for the reader to understand the basics of OPC UA. Mappings between definitions in the OPC UA specifications and those used in the OPC UA .NET Standard Solutions are given as soon as they are used the first time.

In case you are interested in an Overview of OPC UA please consider downloading part 1 (Overview and Concepts) through the OPC Foundation web site. The direct link is
http://www.opcfoundation.org/DownloadFile.aspx?CM=3&RI=414&CN=KEY&CI=283&CU=7.

4.1.2 Graphical Notation

The OPC UA specification **Error! Reference source not found.** defines a graphical notation for an OPC UA *A ddressSpace*. It defines graphical symbols for all NodeClasses and how References of different types can be visualized. This notation is also used in this documentation and therefore explained in this chapter.

4.1.2.1 Overview

The OPC UA specification **Error! Reference source not found.** defines a graphical notation for OPC UA data. I t is normative, that is, the notation is used in the OPC UA specification to expose examples of OPC UA data. However, it is not required to use this notation to expose OPC UA data.
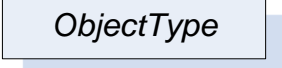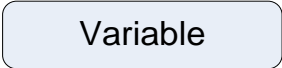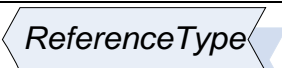
The graphical notation can expose all structural data of OPC UA. *Nodes*, their *Attributes* including their current value and *References* between the *Nodes* including the *ReferenceType* can be exposed. The graphical notation provides no mechanism to expose events or historical data.

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

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

4.1.2.2 Simple Notation

Depending on their *NodeClass Nodes* are represented by different graphical forms as defined in the following table:

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

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

Node1	ReferenceName	Node2
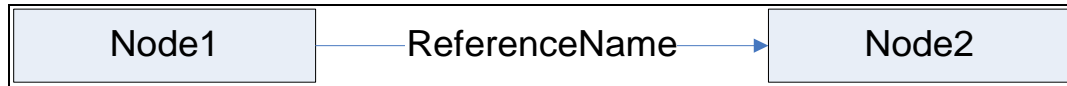
The next table defines how symmetric and asymmetric *References* are represented in general, and also defines shortcuts for some *ReferenceTypes*. Although it is recommended to use those shortcuts, it is not required. Thus, instead of using the shortcut also the generic solution can be used.

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

4.1.2.3 Extended Notation

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

The following rules define some special handling of structures.

- In general, values of all *DataTypes* should be represented by an appropriate string representation. Whenever a *NamespaceIndex* or *LocaleId* is used in those structures they can be omitted.

- The *DisplayName* contains a *LocaleId* and a *String*. Such a structure can be exposed as [<LocaleId>:]<String>; where the *LocaleId* is optional. For example, a *DisplayName* can be "en:MyName". Instead of that, also "MyName" can be used. This rule applies whenever a *DisplayName* is shown, including the text used in the graphical representation of a *Node*.

- The *BrowseName* contains the *NamespaceIndex* and a *String*. Such a structure can be exposed as [<NamespaceIndex>:]<String>; where the *NamespaceIndex* is optional. For example, a *BrowseName* can be "1:MyName". Instead of that, also "MyName" can be used. This rule applies whenever a *BrowseName* is shown, including the text used in the graphical representation of a *Node*.

Instead of using the *HasTypeDefinition* reference to point from an *Object* or *Variable* to its *ObjectType* or *VariableType* the name of the *TypeDefinition* can be added to the text used in the *Node*. The *TypeDefinition* has to be prefixed with "::". The next figure gives an example, where "Node1" uses a *Reference* and "Node2" the shortcut. A figure can contain *HasTypeDefinition References* for some *Nodes* and the shortcut for other *Nodes*. It is not allowed that a *Node* uses the shortcut and additionally is the *SourceNode* of a *HasTypeDefinition*.
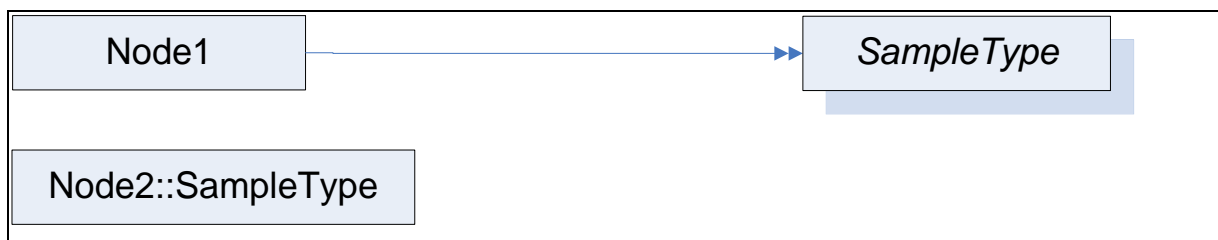
To display *Attributes* of a *Node* additional text can be put inside the form representing the *Node* under the text representing the *DisplayName*. The *DisplayName* and the text describing the *Attributes* have to be separated using a horizontal line. Each *Attribute* has to be set into a new text line. Each text line shall contain the *Attribute* name followed by an "=" and the value of the *Attribute*. On top of the first text line containing an *Attribute* shall be a text line containing the underlined text "Attribute". It is not required to expose all *Attributes* of a *Node*. It is allowed to show only a subset of *Attributes*. If an optional *Attribute* is not provided, the *Attribute* can be marked by a strike-through line, for example "~~Description~~". Examples of exposing *Attributes* are shown in the figure below:
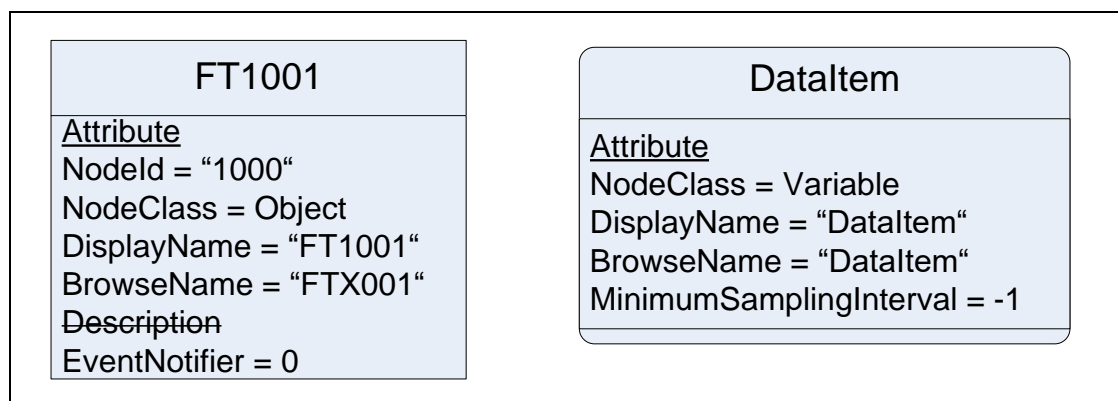
T

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

FT1001	DataItem
Attribute NodeId = "1000" DisplayName = "FT1001" BrowseName = "FTX001" ~~Description~~ EventNotifier = 0 Property Prop1 = 12 Prop2 = "PropValue"	Attribute NodeClass = Variable DisplayName = "DataItem" BrowseName = "DataItem" MinimumSamplingInterval = -1 Property Prop1 = 12 Prop2 = "PropValue"

FT1002	DataItemX
Property Prop1 = 12 Prop2 = "PropValue"	Property Prop1 = 12 Prop2 = "PropValue"

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

T

4.1.3 Terms, definitions and abbreviations

For the purposes of this specification, the following definitions apply. Please note that the following sections are an excerpt of **Error! Reference source not found.** of the OPC UA specification.

4.1.3.1 OPC UA

4.1.3.1.1 Terms

AddressSpace

The collection of information that an OPC UA *Server* makes visible to its *Clients*. See **Error! Reference source not found.** for a description of the contents and structure of the *Server AddressSpace*.

Alarm

A type of *Event* associated with a state condition that typically requires acknowledgement. See **Error! Reference source not found.** for a description of *Alarms*.

Attribute

A primitive characteristic of a *Node*. All *Attributes* are defined by OPC UA, and may not be defined by *Clients* or *Servers*. *Attributes* are the only elements in the *AddressSpace* permitted to have data values.

Certificate

A digitally signed data structure that describes capabilities of a *Client* or *Server*.

Client

A software application that sends *Messages* to OPC UA *Servers* conforming to the *Services* specified in this set of specifications.

Condition

A generic term that is an extension to an *Event*. A *Condition* represents the conditions of a system or one of its components and always exists in some state.

Communication Stack

A layered set of software modules between the application and the hardware that provides various functions to encode, encrypt and format a *Message* for sending, and to decode, decrypt and unpack a *Message* that was received.

Complex Data

Data that is composed of elements or more than one primitive data type, such as a structure.

Discovery

The process by which OPC UA Clients obtain information about OPC UA Servers, including endpoint and security information.

Event

A generic term used to describe an occurrence of some significance within a system or system component.

EventNotifier

A special *Attribute* of a *Node* that signifies that a *Client* may subscribe to that particular *Node* to receive *Notifications* of *Event* occurrences.

T

Information Model

An organizational framework that defines, characterizes and relates information resources of a given system or set of systems. The core address space model supports the representation of *Information Models* in the *AddressSpace*. See **Error! Reference source not found.** for a description of the base OPC UA *Information Model*.

Message

The data unit conveyed between *Client* and *Server* that represents a specific *Service* request or response.

Method

A callable software function that is a component of an *Object*.

MonitoredItem

A *Client*-defined entity in the *Server* used to monitor *Attributes* or *EventNotifiers* for new values or *Event* occurrences and generate *Notifications* for them.

Node

The fundamental component of an *AddressSpace*.

NodeClass

The class of a *Node* in an *AddressSpace*. *NodeClasses* define the metadata for the components of the OPC UA Object Model. They also define constructs, such as *Views,* that are used to organize the *AddressSpace*.

Notification

The generic term for data that announces the detection of an *Event* or of a changed *Attribute* value. *Notifications* are sent in *NotificationMessages*.

NotificationMessage

A *Message* published from a *Subscription* that contains one or more *Notifications*.

Object

A *Node* that represents a physical or abstract element of a system. *Objects* are modelled using the OPC UA Object Model. Systems, subsystems and devices are examples of *Objects*. An *Object* may be defined as an instance of an *ObjectType*.

Object Instance

A synonym for *Object*. Not all *Objects* are defined by *ObjectTypes*.

ObjectType

A *Node* that represents the type definition for an *Object*.

Profile

A specific set of capabilities, defined in **Error! Reference source not found.**, to which a *Server* may claim conformance. Each *Server* may claim conformance to more than one *Profile*.

Program

An executable *Object* that, when invoked, immediately returns a response to indicate that execution has started, and then returns intermediate and final results through *Subscriptions* identified by the *Client* during invocation.

T

Reference

An explicit relationship (a named pointer) from one *Node* to another. The *Node* that contains the *Reference* is the source *Node,* and the referenced *Node* is the target *Node*. All *References* are defined by *ReferenceTypes*.

ReferenceType

A *Node* that represents the type definition of a *Reference*. The *ReferenceType* specifies the semantics of a *Reference*. The name of a *ReferenceType* identifies how source *Nodes* are related to target *Nodes* and generally reflects an operation between the two, such as "A *Contains* B".

RootNode

The beginning or top *Node* of a hierarchy. The *RootNode* of the OPC UA *AddressSpace* is defined in **Error! R eference source not found.**.

Server

A software application that implements and exposes the *Services* specified in this set of specifications.

Service

A *Client*-callable operation in an OPC UA *Server. Services* are defined in **Error! Reference source not found.**. A *Service* is similar to a method call in a programming language or an operation in a Web services WSDL contract.

Service Set

A group of related *Services*.

Session

A logical long-running connection between a *Client* and a *Server.* A *Session* maintains state information between *Service* calls from the *Client* to the *Server*.

Subscription

A *Client*-defined endpoint in the *Server,* used to return *Notifications* to the *Client*. Generic term that describes a set of *Nodes* selected by the *Client* (1) that the *Server* periodically monitors for the existence of some condition, and (2) for which the *Server* sends *Notifications* to the *Client* when the condition is detected.

Variable

A *Variable* is a *Node* that contains a value.

View

A specific subset of the *AddressSpace* that is of interest to the *Client*.

T

4.1.3.1.2 Abbreviations and symbols

A&E   Alarms and Events

API   Application Programming Interface

COM   Component Object Model

DA   Data Access

DCS   Distributed Control System

DX   Data Exchange

HDA   Historical Data Access

HMI   Human-Machine Interface

LDAP   Lightweight Directory Access Protocol

MES   Manufacturing Execution System

OPC   OPC Foundation (a non-profit industry association)

PLC   Programmable Logic Controller

SCADA Supervisory Control And Data Acquisition

SOAP   Simple Object Access Protocol

UA   Unified Architecture

UDDI   Universal Description, Discovery and Integration

UML   Unified Modelling Language

WSDL Web Services Definition Language

XML   Extensible Mark-up Language

T

4.1.3.2 OPC UA Security

4.1.3.2.1 Terms

Application Instance

an individual installation of a program running on one computer.

NOTE: There can be several *Application Instances* of the same application running at the same time on several computers or possibly the same computer.

Application Instance Certificate

a *Digital Certificate* of an individual *Application Instance* that has been installed in an individual host.

NOTE: Different installations of one software product would have different *Application Instance Certificates*.

Asymmetric Cryptography

a *Cryptography* method that uses a pair of keys, one that is designated the *Private Key* and kept secret, the other called the *Public Key* that is generally made available.

NOTE: '*Asymmetric Cryptography*, also known as "public-key cryptography". In an *Asymmetric Encryption* algorithm when an entity A wants to ensure *Confidentiality* for data it sends to another entity B, entity A encrypts the data with a *Public Key* provided by entity B. Only entity B has the matching *Private Key* that is needed to decrypt the data. In an asymmetric *Digital Signature* algorithm when an entity A wants to ensure *Integrity* or provide *Authentication* for data it sends to an entity B, entity A uses its *Private Key* to sign the data. To verify the signature, entity B uses the matching *Public Key* that entity A has provided. In an asymmetric key agreement algorithm, entity A and entity B each send their own *Public Key* to the other entity. Then each uses their own *Private Key* and the other's *Public Key* to compute the new key value.' according to **Error! Reference source not found.**

Asymmetric Encryption

the mechanism used by *Asymmetric Cryptography* for encrypting data with the *Public Key* of an entity and for decrypting data with the associated *Private Key*.

NOTE: See **Error! Reference source not found.** for details.

Asymmetric Signature

the mechanism used by *Asymmetric Cryptography* for signing data with the *Private Key* of an entity and for verifying the data's signature with the associated *Public Key*.

NOTE: See **Error! Reference source not found.** for details.

Auditability

a security objective that assures that any actions or activities in a system can be recorded.

T

Auditing

the tracking of actions and activities in the system, including security related activities where the *Audit* records can be used to review and verify system operations.

Authentication

a security objective that assures that the identity of an entity such as a *Client, Server*, or user can be verified.

Authorization

a security objective that assures the control to grant the right or the permission to a system entity to access a system resource.

Availability

a security objective that assures that the system is running normally. that is, no services have been compromised in such a way to become unavailable or severely degraded.

CertificateAuthority

an entity that can issues *Digital Certificate*s, also known as a CA.

Note: The *Digital Certificate* certifies the ownership of a *Public Key* by the named subject of the *Certificate*. This allows others (relying parties) to rely upon signatures or assertions made by the *Private Key* that corresponds to the *Public Key* that is certified. In this model of trust relationships, a CA is a [trusted third party](#) that is trusted by both the subject (owner) of the *Certificate* and the party relying upon the *Certificate*. CAs are characteristic of many [*Public Key* infrastructure](#) (PKI) schemes

CertificateStore

persistent location where *Certificate*s and *Certificate* revocation lists (CRLs) are stored.

Note: It maybe a disk resident file structure or on Windows platforms, it maybe a Windows registry location.

Confidentiality

a security objective that assures the protection of data from being read by unintended parties.

Cryptogrophy

transforming clear, meaningful information into an enciphered, unintelligible form using an algorithm and a key.

Cyber Security Management System (CSMS)

a program designed by an organization to maintain the security of the entire organization's assets to an established level of *Confidentiality*, *Integrity*, and *Availability*, whether they are on the business side or the industrial automation and control systems side of the organization.

Digital Certificate

a structure that associates an identity with an entity such as a user, a product or an *Application Instance* where the *Certificate* has an associated asymmetric key pair which can be used to authenticate that the entity does, indeed, possess the *Private Key*.

Digital Signature

a value computed with a cryptographic algorithm and appended to data in such a way that any recipient of the data can use the signature to verify the data's origin and *Integrity*.

T

Hash Function

'an algorithm such as SHA-1 for which it is computationally infeasible to find either a data object that maps to a given hash result (the "one-way" property) or two data objects that map to the same hash result (the "collision-free" property).' according to **Error! Reference source not found.**

Hashed Message Authentication Code (HMAC)

a *MAC* that has been generated using an iterative *Hash Function*.

Integrity

a security objective that assures that information has not been modified or destroyed in a unauthorized manner.

NOTE: definition from **Error! Reference source not found.**.

Key Exchange Algorithm

a protocol used for establishing a secure communication path between two entities in an unsecured environment whereby both entities apply a specific algorithm to securely exchange secret keys that are used for securing the communication between them.

NOTE: A typical example of a *Key Exchange Algorithm* is the SSL Handshake Protocol specified in **Error! R eference source not found.**.

Message Authentication Code (MAC)

a short piece of data that results from an algorithm that uses a secret key (see *Symmetric Cryptography*) to hash a *Message* whereby the receiver of the *Message* can check against alteration of the *Message* by computing a *MAC* that should be identical using the same *Message* and secret key.

Message Signature

a *Digital Signature* used to ensure the *Integrity* of *Message*s that are sent between two entities.

NOTE: There are several ways to generate and verify *Message Signatures* however they can be categorized as symmetric (See **Error! Reference source not found.**) and asymmetric (See **Error! Reference source not fo und.**) approaches.

T

Non-Repudiation

strong and substantial evidence of the identity of the signer of a *Message* and of *Message Integrity*, sufficient to prevent a party from successfully denying the original submission or delivery of the *Message* and the *Integrity* of its contents.

Nonce

a random number that is used once, typically by algorithms that generate security keys.

OPC UA Application

an OPC UA *Client*, which calls OPC UA services, or an OPC UA *Server*, which performs those services.

Private Key

the secret component of a pair of cryptographic keys used for *Asymmetric Cryptography*.

Public Key

the publicly-disclosed component of a pair of cryptographic keys used for *Asymmetric Cryptography*. **Error! R eference source not found.**

Public Key Infrastructure (PKI)

the set of hardware, software, people, policies, and procedures needed to create, manage, store, distribute, and revoke *Digital Certificates* based on *Asymmetric Cryptography*.

NOTE: The core *PKI* functions are to register users and issue their public-key *Certificate*s, to revoke *Certificate*s when required, and to archive data needed to validate *Certificate*s at a much later time. Key pairs for data *Confidentiality* may be generated by a *Certificate* authority (CA), but requiring a *Private Key* owner to generate its own key pair improves security because the *Private Key* would never be transmitted. **Error! R eference source not found.**. See **Error! Reference source not found.** and **Error! Reference source not found.** for more details on *Public Key Infrastructure*s.

Rivest-Shamir-Adleman (RSA)

an algorithm for *Asymmetric Cryptography*, invented in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman. **Error! Reference source not found.**

Secure Channel

in OPC UA, a communication path established between an OPC UA *Client* and *Server* that have authenticated each other using certain OPC UA services and for which security parameters have been negotiated and applied.

T

Symmetric Cryptography

A branch of cryptography involving algorithms that use the same key for two different steps of the algorithm (such as encryption and decryption, or Signature creation and signature verification). **Error! Reference source not found.**

Symmetric Encryption

the mechanism used by *Symmetric Cryptography* for encrypting and decrypting data with a cryptographic key shared by two entities.

Symmetric Signature

the mechanism used by *Symmetric Cryptography* for signing data with a *cryptographic key* shared by two entities.

NOTE: The signature is then validated by generating the signature for the data again and comparing these two signatures. If they are the same then the signature is valid, otherwise either the key or the data is different from the two entities. **Error! Reference source not found.** defines a typical example for an algorithm that generates *Symmetric Signature*s.

TrustList

a list of *Certificate*s that an application has been configured to trust.

Transport Layer Security (TLS)

 a standard protocol for creating *Secure Channel*s over IP based networks.

X.509 Certificate

a *Digital Certificate* in one of the formats defined by X.509 v1, 2, or 3. An *X.509 Certificate* contains a sequence of data items and has a *Digital Signature* computed on that sequence.

4.1.3.2.2 Abbreviations and symbols

AES   Advanced Encryption Standard

CA   Certificate Authority

CRL   Certificate Revocation List

CSMS   Cyber Security Management System

DNS   Domain Name System

DSA   Digital Signature Algorithm

ECDH   Elliptic Curve Diffie-Hellman

ECDSA   Elliptic Curve Digital Signature Algorithm

HMAC   Hash-based Message Authentication Code

PKI   Public Key Infrastructure

RSA   public key algorithm for signing or encryption, Rivest, Shamir, Adleman

SHA   Secure Hash Algorithm (Multiple versions exist SHA1, SHA256,…)

SOAP   Simple Object Access Protocol

SSL   Secure Sockets Layer

TLS   Transport Layer Security

UA   Unified Architecture

URI   Uniform Resource Identifier

XML   Extensible Mark-up Language

T

4.1.3.3 OPC UA Address Space

4.1.3.3.1 Terms

DataType

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

DataTypeId

NodeId of a DataType Node.

DataVariable

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

EventType

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

Hierarchical Reference

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

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

InstanceDeclaration

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

ModellingRule

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

Property

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

SourceNode

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

TargetNode

Node that is referenced by another *Node;* For example, in the *Reference* "A Contains B", "B" is the *TargetNode*

TypeDefinitionNode

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

VariableType

Node that represents the type definition for a *Variable*

T

4.1.3.3.2 Abbreviations

UA   Unified Architecture

UML   Unified Modeling Language

URI   Uniform Resource Identifier

W3C   World Wide Web Consortium

XML   Extensible Markup Language

T

4.1.3.4   OPC UA Services

4.1.3.4.1   Terms

Deadband

permitted range for value changes that will not trigger a data change *Notification*

NOTE   *Deadband* can be applied as a filter when subscribing to *Variables* and is used to keep noisy signals from updating the *Client* unnecessarily. This standard defines *AbsoluteDeadband* as a common filter. **Error! R eference source not found.** defines an additional *Deadband* filter.

Endpoint

physical address available on a network that allows *Clients* to access one or more *Services* provided by a *Server*

NOTE   Each *Server* may have multiple *Endpoints*. The address of an *Endpoint* must include a *HostName*.

Gateway Server

Server that acts as an intermediary for one or more *Servers*

NOTE   *Gateway Servers* may be deployed to limit external access, provide protocol conversion or to provide features that the underlying *Servers* do not support.

HostName

unique identifier for a machine on a network

NOTE   This identifier shall be unique within a local network; however, it may also be globally unique. The identifier can be an IP address.

Security Token

identifier for a cryptographic key set

NOTE   All *Security Tokens* belong to a security context which is in case of OPC UA the *Secure Channel*.

SoftwareCertificate

digital certificate for a software product that can be installed on several hosts to describe the capabilities of the software product

NOTE   Different installations of one software product could have the same software certificate. Software certificates are not relevant for security. They are used to identify a software product and its supported features. SoftwareCertificates are described in **Error! Reference source not found.**.

4.1.3.4.2 Abbreviations and symbols

API   Application Programming Interface

BNF   Backus-Naur Form

CA   Certificate Authority

CRL   Certificate Revocation List

CTL   Certificate Trust List

DA   Data Access

UA   Unified Architecture

URI   Uniform Resource Identifier

URL   Uniform Resource Locator

T

4.1.3.5 OPC UA Information Model

4.1.3.5.1 Terms

ClientUserId

String that identifies the user of the client requesting an action

NOTE: The *ClientUserId* is obtained directly or indirectly from the *UserIdentityToken* passed by the *Client* in the *ActivateSession Service* call. See **Error! Reference source not found.**, Chapter 6.4.3 for details.

4.1.3.5.2 Abbreviations and symbols

UA   Unified Architecture

XML   Extensible Markup Language

T

4.1.3.6 OPC UA Mappings

4.1.3.6.1 Terms

Data Encoding

Is a way to serialize OPC UA *Messages* and data structures.

Mapping

Specifies how to implement an OPC UA feature with a specific technology.

NOTE For example, the OPC UA Binary Encoding is a *Mapping* that specifies how to serialize OPC UA data structures as sequences of bytes.

Security Protocol

Ensures the integrity and privacy of UA *Messages* that are exchanged between OPC UA applications

Stack

Is a collection of software libraries that implement one or more *Stack Profiles*; *Stacks* have an API which hides the implementation details from the *Application* developer

Stack Profile

Is a combination of DataEncodings, SecurityProtocol and TransportProtocol Mappings

NOTE OPC UA applications implement one or more *StackProfiles* and can only communicate with OPC UA applications that support a *StackProfile* that they support.

Transport Protocol

Represents a way to exchange serialized OPC UA *Messages* between OPC UA applications

T

4.1.3.6.2 Abbreviations and symbols

API   Application Programming Interface

ASN.1   Abstract Syntax Notation #1

BP   WS-I Basic Profile Version

BSP   WS-I Basic Security Profile

CSV   Comma Separated Value (File Format)

HTTP   Hypertext Transfer Protocol

HTTPS   Secure Hypertext Transfer Protocol

IPSec   Internet Protocol Security

RST   Request Security Token

OID   Object Identifier (used with ASN.1)

RSTR   Request Security Token Response

SCT   Security Context Token

SHA1   Secure Hash Algorithm

SOAP   Simple Object Access Protocol

SSL   Secure Sockets Layer (Defined in **Error! Reference source not found.**)

TCP   Transmission Control Protocol

TLS   Transport Layer Security (Defined in **Error! Reference source not found.**)

UTF8   Unicode Transformation Format (8-bit)

UA   Unified Architecture

UASC   OPC UA Secure Conversation

WS-*   XML Web Services Specifications

WSS   WS Security

WS-SC   WS Secure Conversation

XML   Extensible Markup Language

T

4.1.3.7 OPC UA Profiles

4.1.3.7.1 Terms

Application

a software program that executes or implements some aspect of OPC UA.

Note: The application could run on any machine and perform any function. The application could be software or it could be a hardware application, the only requirement is that it implements OPC UA

ConformanceUnit

a specific set of OPC UA features that can be tested as a single entity.

Note: A *ConformanceUnit* can cover a group of services, portions of services or information models. For additional detail see **Error! Reference source not found.**.

ConformanceGroup

a group of *ConformanceUnits* that is given a name.

Note: This grouping is only to assist in organizing *ConformanceUnits*. Typical *ConformanceGroups* include groups for each of the service sets in OPC UA and each of the Information Model standards.

Facet

a Profile dedicated to a specific feature that a Server or Client may require.

Note: Facets are typically combined to form higher-level Profiles. The use of the term Facet in the title of a Profile indicates that the given Profile is not a standalone Profile.

ProfileCategory

arranges Profiles into application classes, such as Server or Client.

Note: These categories help determine the type of Application that a given Profile would be used for. For additional details see **Error! Reference source not found.** section 4.4

TestCase

a technical description of a set of steps required to test a particular function or information model.

Note: TestCases provide sufficient details to allow a developer to implement them in code. TestCases also provide a detailed summary of the expected result(s) from the execution of the implemented code and any precondition(s) that must be established before the TestCase can be executed.

TestLab

a facility that is designated to provide testing services.

Note: These services include but are not limited to personal that directly perform testing, automated testing and a formal repeatable process. The OPC Foundation has provided detailed standard describing OPC UA TestLabs and the testing they are to provided (see Compliance Part 8 UA Server, Compliance Part 9 UA Client,)

T

4.1.3.7.2 *Abbreviations*

API Application Programming Interface

DA Data Access

HA Historical Access

HMI Human Machine Interface

PKI Public Key Infrastructure

RSA Rivest-Shamir-Adleman

UA Unified Architecture

T

4.1.3.8   OPC UA Data Access

4.1.3.8.1   Terms

DataItem

link to arbitrary, live automation data, that is, data that represents currently valid information

NOTE   Examples of such data are

- device data (such as temperature sensors),
- calculated data,
- status information (open/closed, moving),
- dynamically-changing system data (such as stock quotes),
- diagnostic data.

AnalogItem

*DataItem*s that represent continuously-variable physical quantities (e.g., length, temperature), in contrast to the digital representation of data in discrete items

NOTE   Typical examples are the values provided by temperature sensors or pressure sensors. OPC UA defines a specific *VariableType* to identify an *AnalogItem*. *Properties* describe the possible ranges of *AnalogItem*s.

DiscreteItem

*DataItem*s that represent data that may take on only a certain number of possible values (e.g., OPENING, OPEN, CLOSING, CLOSED)

NOTE   Specific *VariableTypes* are used to identify *DiscreteItem*s with two states or with multiple states. *Properties* specify the string values for these states.

ArrayItem

*DataItem*s that represent continuously-variable physical quantities and where each individual data point consists of multiple values represented by an array (e.g., the spectral response of a digital filter)

NOTE   Typical examples are the data provided by analyser devices. Specific *VariableTypes* are used to identify *ArrayItem* variants.

EngineeringUnits

Units of measurement for *AnalogItem*s that represent continuously-variable physical quantities (e.g., length, mass, time, temperature)

NOTE   This standard defines *Properties* to inform about the unit used for the *DataItem* value and about the highest and lowest value likely to be obtained in normal operation.

T

4.1.3.8.2 Abbreviations and symbols

DA   Data Access

EU   Engineering Unit

UA   Unified Architecture

4.1.3.9   OPC UA Alarms & Conditions

4.1.3.9.1   Terms

Acknowledge

"*Operator* action that indicates recognition of a new *Alarm*"

Note: as defined in **Error! Reference source not found.**. The term "Accept" is another common term used to d escribe *Acknowledge*. They can be used interchangeably. This document will use *Acknowledge.*

Active

a state for an Alarm that indicates that the situation the *Alarm* is representing currently exists.

Note: Other common terms defined by **Error! Reference source not found.** are "Standing" for an *Active A larm* and "Cleared" when the *Condition* has returned to normal and is no longer *Active*.

ConditionClass

a Condition grouping that indicates in which domain or for what purpose a certain *Condition* is used.

Note: Some top-level *ConditionClasses* are defined in this specification. Vendors or organisations may derive more concrete classes or define different top-level classes.

ConditionBranch

a specific state of a *Condition*.

Note: The *Server* can maintain *ConditionBranches* for the current state as well as for previous states.

ConditionSource

Element which a specific *Condition* is based upon or related to.

Note: Typically, it will be a *Variable* representing a process tag (e.g. FIC101) or an *Object* representing a device or subsystem.

In *Events* generated for *Conditions*, the *SourceNode Property* (inherited from the *BaseEventType*) will contain the *NodeId* of the *ConditionSource*.

Confirm

Operator action informing the *Server* that a corrective action has been taken to address the cause of the *Alarm*.

Disable

"System is configured such that the *Alarm* will not be generated even though the base *Alarm Condition* is present"

Note: This concept is further described in EEMUA

Operator

Special user who is assigned to monitor and control a portion of a process

Note: "A Member of the operations team who is assigned to monitor and control a portion of the process and is working at the control system's Console" as defined in **Error! Reference source not found.**. In this standard a n Operator is a special user. All descriptions that apply to general users also apply to Operators.

Refresh

T

the act of providing an update to an *Event Subscription* that provides all *Alarms* which are considered to be *Retained*.

Note: This concept is further described in **Error! Reference source not found.**.

Retain

Alarm in a state that is interesting for a *Client* wishing to synchronize its state of *Conditions* with the *Server*'s state.

Shelving

"Facility where the *Operator* is able to temporarily prevent an *Alarm* from being displayed to the *Operator* when it is causing the *Operator* a nuisance.

Note: A Shelved *Alarm* will be removed from the list and will not re-annunciate until un-shelved" as defined in **Error! Reference source not found.**.

Suppress

the act of determining whether an *Alarm* should not occur

Note: "An *Alarm* is suppressed when logical criteria are applied to determine that the *Alarm* should not occur, even though the base *Alarm Condition* (e.g. *Alarm* setting exceeded) is present" as defined in **Error! R eference source not found.**.

Abbreviations and symbols

DA   Data Access

UA   Unified Architecture

UML   Unified Modelling Language

XML   Extensible Mark-up Language

4.1.3.10   OPC UA Programs

4.1.3.10.1   *Terms*

Function

programmatic task performed at a server or device, usually accomplished by computer code execution

Finite State Machine

sequence of states and valid state transitions along with the causes and effects of those state transitions that define the actions of a *Program* in terms of discrete stages

ProgramType

ObjectType Node that represents the type definition of a *Program* and is a subtype of the *FiniteStateMachineType*

Program Control Method

Method specified by this specification having specific semantics designed for the control of a *Program* by causing a state transition

Program Invocation

unique *Object* instance of a *Program* existing on a *Server*

NOTE   The Program Invocation is distinguished from other Object instances of the same ProgramType by the object node's unique browse path.

4.1.3.10.2   *Abbreviations*

API   Application Programming Interface

DA   Data Access

FSM   Finite State Machine

HMI   Human Machine Interfaces

PCM   Program Control Method

PGM   Program

PI   Program Invocation

T

4.1.3.11 OPC UA Historical Access

4.1.3.11.1 Terms

Annotation

metadata associated with an item at a given instance in time

NOTE An *Annotation* is metadata that is associated with an item at a given instance in time. There does not have to be a value stored at that time.

BoundingValues

values associated with the starting and ending time

NOTE *BoundingValues* are the values that are associated with the starting and ending time of a *ProcessingInterval* specified when reading from the historian. *BoundingValues* may be required by *Clients* to determine the starting and ending values when requesting *raw data* over a time range. If a *raw data* value exists at the start or end point, it is considered the bounding value even though it is part of the data request. If no *raw data* value exists at the start or end point, then the server will determine the boundary value, which may require data from a data point outside of the requested range. See **Error! Reference source not found.** f or details on using *BoundingValues*.

HistoricalNode

Object, *Variable*, *Property* or *View* in the *AddressSpace* where a *Client* can access historical data or *Events*

NOTE A **Error! Unknown document property name.** is a term used in this document to represent any *Object*, *Variable, Property* or *View* in the *AddressSpace* for which a *Client* may read and/or update historical data or *Events*. The terms "**Error! Unknown document property name.**'s history" or "history of a **Error! Unknown document property name.**" will refer to the time series data or *Events* stored for this *HistoricalNode* where *HistoricalNode* is an *Object*, *Variable*, *Property* or *View*. The term **Error! Unknown document property name.** refers to both **Error! Unknown document property name.**s and **Error! Unknown document property name.**s, and is used when referencing aspects of the specification that apply to accessing historical data and *Events*.

HistoricalDataNode

Variable or *Property* in the *AddressSpace* where a *Client* can access historical data

NOTE A **Error! Unknown document property name.** represents any Variable or Property in the AddressSpace for which a Client may read and/or update historical data. The terms "**Error! Unknown document property name.**'s history" or "history of a **Error! Unknown document property name.**" will refer to the time series data stored for this HistoricalNode where HistoricalNode is an Object, Variable, or Property. Some examples of such data are:

- device data (like temperature sensors)
- calculated data
- status information (open/closed, moving)
- dynamically changing system data (like stock quotes)
- diagnostic data

The term **Error! Unknown document property name.**s is used when referencing aspects of the specification that apply to accessing historical data only.

T

HistoricalEventNode

Object or *View* in the *AddressSpace* where a *Client* can access historical *Events*

NOTE   A **Error! Unknown document property name.** represents any Object or View in the AddressSpace for which a Client may read and/or update historical Events. The terms "**Error! Unknown document property name.**'s history" or "history of a **Error! Unknown document property name.**" will refer to the time series Events stored in some historical system. Some examples of such data are:

- Notifications
- system Alarms
- operator action Events
- system triggers (such as new orders to be processed)

The term **Error! Unknown document property name.** is used when referencing aspects of the specification that apply to accessing historical Events only.

Modified values

Error! Unknown document property name.'s value that has been changed

NOTE   A *modified value* is a ***Error! Unknown document property name.**'s* value that has been changed (or manually inserted or deleted) after it was stored in the historian. For some servers, a lab data entry value is not a *modified value*, but if a user corrects a lab value, the original value would be considered a *modified value*, and would be returned during a request for *modified values*. Also manually inserting a value that was missed by a standard collection system may be considered a *modified value*. Unless specified otherwise, all historical *Services* operate on the current, or most recent, value for the specified ***Error! Unknown document property name.*** at the specified timestamp. Requests for *modified values* are used to access values that have been superseded, deleted or inserted. It is up to a system to determine what is considered a *modified value*. Whenever a server has modified data available for an entry in the historical collection it shall set the *ExtraData* bit in the *StatusCode.*

Raw data

data that is stored within the historian for a **Error! Unknown document property name.**

NOTE   Raw data is data that is stored within the historian for a **Error! Unknown document property name.** . The data may be all data collected for the DataValue or it may be some subset of the data depending on the historian and the storage rules invoked when the item's values were saved.

StartTime / EndTime

the bounds of a history request and define the time domain

NOTE   The *StartTime* and *EndTime* specify the bounds of a history request and define the time domain of the request. For all requests, a value falling at the end time of the time domain is not included in the domain, so that requests made for successive, contiguous time domains will include every value in the historical collection exactly once.

TimeDomain

interval of time covered by a particular request, or response

NOTE   The interval of time covered by a particular request, or by a particular response. In general, if the start time is earlier than or the same as the end time, the time domain is considered to begin at the start time and end just before the end time; if the end time is earlier than the start time, the time domain still begins at the start time and ends just before the end time, with time "running backward" for the particular request and response. In both cases, any value which falls exactly at the end time of the *TimeDomain* is not included in the

T

TimeDomain. See the examples in the **Error! Reference source not found.**. *BoundingValues* effect the time d omain as described in the **Error! Reference source not found.**.

All timestamps which can legally be represented in an *UtcTime DataType* are valid timestamps, and the server may not return an invalid argument result code due to the timestamp being outside of the range for which the server has data. See **Error! Reference source not found.** for a description of the range and granularity of this *D ataType*. Servers are expected to handle out-of-bounds timestamps gracefully, and return the proper *StatusCodes* to the *Client.*

Structured History Data

structured data stored in a history collection where parts of the structure are used to uniquely identify the data

NOTE *Structured History Data* is structured data stored in a history collection where one or more parameters of the structure are used to uniquely identify the data within the data collection. Most historical data applications assume only one current value per timestamp. Therefore the timestamp of the data is considered the unique identifier for that value. Some data or meta data such as *Annotations* may permit multiple values to exist at a single timestamp. In such cases the Server would use one or more parameters of the *Structured History Data* entry to uniquely identifiy each within the history collection. *Annotations* are examples of *Structured History Data*.

4.1.3.11.2   Abbreviations and symbols

DA   Data Access

HDA   Historical Data Access

UA   Unified Architecture

4.1.3.12 OPC UA Discovery

4.1.3.12.1 Terms

CertificateManagement Server

a software application that manages the *Certificates* used by *Applications* in an enterprise.

DirectoryService

a software application, or a set of applications, that stores and organizes information about network resources such as computers or services.

DiscoveryServer

an *Application* that maintains a list of OPC UA *Servers* that are available on the network and provides mechanisms for Clients to obtain this list.

DiscoveryUrl

a URL for a network *Endpoint* that provides the information required to connect to a *Server*.

GlobalDiscoveryServer (GDS)

a *DiscoveryServer* that maintains a list of OPC UA *Applications* available in an enterprise.

Note: a GDS may also provide certificate management services.

IPAddress

a unique number assigned to a network interface that allows Internet Protocol (IP) requests to be routed to that interface.

Note: An *IPAddress* for a host may change over time.

LocalDiscoveryServer (LDS)

a *DiscoveryServer* that maintains a list of all *Servers* that have registered with it.

Note: *Servers* normally register with the LDS on the same host.

LocalDiscoveryServer-ME (LDS-ME)

a Local*DiscoveryServer* that includes the MulticastExtension.

MulticastExtension

an extension to a *LocalDiscoveryServer* that adds support for the mDNS protocol.

MulticastSubnet

a network that allows multicast packets to be sent to all nodes connected to the network.

Note: a *MulticastSubnet* is not necessarily the same as a TCP/IP subnet.

ServerCapabilityIdentifier

a short identifier which uniquely identifies a set of discoverable capabilities supported by a *Server*.

Note: the OPC Foundation maintains a list of the currently defined *ServerCapabilityIdentifiers*.

T

4.1.3.12.2 Abbreviations and symbols

API   Application Programming Interface

UA   Unified Architecture

GDS   Global Directory Server

IANA   The Internet Assigned Numbers Authority

LDS   Local Discovery Server

LDS-ME   Local Discovery Server with the Multicast Extension.

TLS   Transport Layer Security

DCHP   Dynamic Host Configuration Protocol

DNS   Domain Name System

UDDI   Universal Description, Discovery and Integration

LDAP   Lightweight Directory Access Protocol

mDNS   Multicast Domain Name System

DER   Distinguished Encoding Rules

SHA1   Secure Hash Algorithm

CRL   Certificate Revocation List

PEM   Privacy Enhanced Mail

PFX   Personal Information Exchange

4.1.3.13 OPC UA Aggregates

4.1.3.13.1 Terms

Aggregate

way to produce derived values from *Raw data*

NOTE Provides summarized data values. An *Aggregate* is a way to produce a set of values derived from the *Raw data*. This *Raw data* may be from a historian or buffered real time data. Clients may specify an *Aggregate* when using the *HistoryRead* service or as part of a *Subscription* as a filter. Complete details of the various standard *Aggregates* and their behaviour are outlined in **Error! Reference source not found.**. Common *A ggregates* include averages over a given time range, minimum over a time range and maximum over a time range.

ProcessingInterval

the amount of time that lies between two bounding timestamps

NOTE A *ProcessingInterval* is the amount of time that lies between two bounding timestamps, and represents the subsection of a given time range an *Aggregate* is applied over. For example, performing a 10 minute Average over the time range 12:00-12:30 would result in a set of three intervals of *ProcessingInterval* length, with each interval having a start time of 12:00, 12:10 and 12:20 respectively. The rules used to determine interval *Bounds* are discussed in **Error! Reference source not found.**.

Interpolated

calculated from data samples

NOTE *Interpolated* data is data that is calculated from data samples; this may be historical data or buffered real time data. An *interpolated* value is calculated from the data points on either side of the requested timestamp. The document refers to two types of *interpolation*, depending on how the data is represented.

EffectiveEndTime

time immediately before *endTime*

NOTE All Aggregate calculations include the *startTime* but exclude the *endTime*. However, it is sometimes necessary to return an *Interpolated* End Bound as the value for an *Interval* with a timestamp that is in the interval. *Servers* are expected to use the time immediately before *endTime* where the time resolution of the *Server* determines the exact value (do not confuse this with hardware or operating system time resolution). For example, if the *endTime* is 12:01:00, the time resolution is 1 second then the *EffectiveEndTime* is 12:00:59. If the Server time resolution is 1 millisecond the *EffectiveEndTime* is 12:00:59.999. See **Error! Reference s ource not found.**.

If time is flowing backwards, *Servers* are expected to use the time immediately after *endTime* where the time resolution of the *Server* determines the exact value.

Extrapolated

data constructed from a discrete data set but is outside of the discrete data set

NOTE *Extrapolated* data is data that is constructed from a discrete data set but is outside of the discrete data set. It is similar to the process of interpolation, which constructs new points between known points, but its result is subject to greater uncertainty. *Extrapolated* data is used in cases where the requested time period falls farther into the future than the data available in the underlying system. See example in Table 1.

SlopedInterpolation

simple linear interpolation

NOTE *SlopedInterpolation* and/or *SlopedExtrapolation* refer to simple linear interpolation, as in the method of curve fitting using linear polynomials. See example in Table 1.

SteppedInterpolation

interpolating the value based on a horizontal line fit

NOTE *SteppedInterpolation* and/or *SteppedExtrapolation* refers to holding the last data point constant, or interpolating the value based on a horizontal line fit.

For example, consider the following table of raw and *Interpolated/Extrapolated* values:

Table 1 – Interpolation Examples

Timestamp	Raw Value	Sloped Interpolation	Stepped Interpolation
12:00:00	10		
12:00:05		15	10
12:00:08		18	10
12:00:10	20		
12:00:15		25	20
12:00:20	30		
		SlopedExtrapolation	**SteppedExtrapolation**
12:00:25		35	30
12:00:27		37	30

Bounding Values

values at the *startTime* and *endTime*

NOTE Many *Aggregates* require values at the *startTime* and *endTime* in order to compute the result. These values are called *Bounding Values*. If *Raw data* does not exist at these times a value must be estimated. There are two ways to determine *Bounding Values* for an interval. One way (called *Interpolated Bounding Values*) uses the first non-Bad data points found before and after the timestamp to estimate the bound. The other (called *Simple Bounding Values*) uses the data points immediately before and after the boundary timestamps to estimate the bound even if these points are bad.

In all cases the *TreatUncertainAsBad* (see **Error! Reference source not found.**) flag is used to determine w hether uncertain values are Bad or non-Bad.

If a Raw value was not found and a non-Bad bounding value exists the Aggregate Bits (see **Error! Reference s ource not found.**) are set to 'Interpolated'.

When calculating bounding values; the value portion of *Raw data* that has Bad status is set to null. This means the value portion is not used in any calculation and a null is returned if the raw value is returned. The status portion is determined by the rules specified by the bound or aggregate.

The *Interpolated Bounding Values* approach (see **Error! Reference source not found.**) is the same as what is u sed in Classic OPC Historical Data Access (HDA) and is important for applications such as advanced process

control where having useful values at all times is important. The *Simple Bounding Values* approach (see **Error! Reference source not found.**) is new in this specification and is important for applications which must p roduce regulatory reports and cannot use estimated values in place of Bad data.

Interpolated Bounding Values

Bounding Values determined by a calculation using the nearest good value

NOTE Interpolated Bounding Values using SlopedInterpolation are calculated as follows:

- If a non-Bad Raw value exists at the timestamp then it is the bounding value;

- Find the first non-Bad Raw value before the timestamp;

- Find the first non-Bad Raw value after the timestamp;

- Draw a line between before value and after value;

- Use point where the line crosses the timestamp as an estimate of the bounding value;

The calculation can be expressed with the following formula:

$$V_{bound} = (T_{bound} - T_{before}) * (V_{after} - V_{before}) / (T_{after} - T_{before}) + V_{before}$$

Where Vx is a value at 'x' and Tx is the timestamp associated with Vx.

If no non-Bad values exist before the timestamp the *StatusCode* is Bad_NoData. The *StatusCode* is Uncertain_DataSubNormal if any Bad values exist between the before value and after value. If either the before value or the after value are Uncertain the *StatusCode* is Uncertain_DataSubNormal. If the after value does not exist the before value must be extrapolated using *SlopedExtrapolation* or *SteppedExtrapolation*.

The period of time that is searched to discover the good values before and after the timestamp is server dependent, but if a good value is not found within some reasonable time range then the server will assume it does not exist. The server as a minimum should search a time range which is at least the size of the ProcessingInterval.

Interpolated Bounding Values using SlopedExtrapolation are calculated as follows:

- Find the first non-Bad Raw value before timestamp;

- Find the second non-Bad Raw value before timestamp;

- Draw a line between these two values;

- Extend the line to where it crosses the timestamp;

- Use the point where the line crosses the timestamp as an estimate of the bounding value;

The formula is the same as the one used for SlopedInterpolation.

The *StatusCode* is always Uncertain_DataSubNormal. If only one non-Bad raw value can be found before the timestamp then SteppedExtrapolation is used to estimate the bounding value.

 Interpolated Bounding Values using SteppedInterpolation are calculated as follows:

- If a non-Bad Raw value exists at the timestamp then it is the bounding value;

- Find the first non-Bad Raw value before timestamp;

- Use the value as an estimate of the bounding value;

The *StatusCode* is Uncertain_DataSubNormal if any Bad values exist between the before value and the timestamp. If no non-Bad *Raw data* exists before the timestamp then the *StatusCode* is Bad_NoData. If the value before the timestamp is Uncertain the *StatusCode* is Uncertain_DataSubNormal. The value after the

---

timestamp is not needed when using SteppedInterpolation; however, if the timestamp is after the end of the data then the bounding value is treated as extrapolated and the *StatusCode* is Uncertain_DataSubNormal.

SteppedExtrapolation is a term that describes SteppedInterpolation when a timestamp is after the last value in the history collection.

Simple Bounding Values

Bounding Values determined by a calculation using the nearest value

NOTE   Simple Bounding Values using SlopedInterpolation are calculated as follows:

- If any Raw value exists at the timestamp then it is the bounding value;

- Find the first Raw value before timestamp;

- Find the first Raw value after timestamp;

- If the value after the timestamp is Bad then the before value is the bounding value;

- Draw a line between before value and after value;

- Use point where the line crosses the timestamp as an estimate of the bounding value;

The formula is the same as the one used for SlopedInterpolation.

If a Raw value at the timestamp is Bad the *StatusCode* is Bad_NoData. If the value before the timestamp is Bad the *StatusCode* is Bad_NoData. If the value before the timestamp is Uncertain the *StatusCode* is Uncertain_DataSubNormal. If the value after the timestamp is Bad or Uncertain the *StatusCode* is Uncertain_DataSubNormal.

Simple Bounding Values using SteppedInterpolation are calculated as follows:

- If any Raw value exists at the timestamp then it is the bounding value;

- Find the first Raw value before timestamp;

- If the value before timestamp is non-Bad then it is the bounding value;

If a Raw value at the timestamp is Bad the *StatusCode* is Bad_NoData. If the value before the timestamp is Bad the *StatusCode* is Bad_NoData. If the value before the timestamp is Uncertain the *StatusCode* is Uncertain_DataSubNormal.

If either bounding time of an interval is beyond the last data point then extrapolation may be used if the Server feels it is appropriate for the data.

In some Historians, the last Raw value does not necessarily indicate the end of the data. Based on the Historian's knowledge of the data collection mechanism, i.e. frequency of data updates and latency, the Historian may extend the last value to a time known by the Historian to be covered. When calculating Simple Bounding Values the Historian will act as if there is another Raw value at this timestamp.

In the same way, if the earliest time of an interval starts before the first data point in history and the latest time is after the first data point in history, then the interval will be treated as if the interval extends from the first data point in history to the latest time of the interval and the *StatusCode* of the interval will have the Partial bit set [See **Error! Reference source not found.**].

The period of time that is searched to discover the values before and after the timestamp is server dependent, but if a value is not found within some reasonable time range then the server will assume it does not exist. The server as a minimum should search a time range which is at least the size of the ProcessingInterval.

T

4.1.3.13.2 Abbreviations and symbols

DA   Data Access

HDA   Historical Data Access

UA   Unified Architecture

4.2 Security

4.2.1 Overview

4.2.1.1 Environment

OPC UA is a protocol used between components in the operation of an industrial facility at multiple levels: from high-level enterprise management to low-level direct process control of a device. The use of OPC UA for enterprise management involves dealings with customers and suppliers. It may be an attractive target for industrial espionage or sabotage and may also be exposed to threats through untargeted malware, such as worms, circulating on public networks. Disruption of communications at the process control end causes at least an economic cost to the enterprise and can have employee and public safety consequences or cause environmental damage. This may be an attractive target for those who seek to harm the enterprise or society.

OPC UA will be deployed in a diverse range of operational environments, with varying assumptions about threats and accessibility, and with a variety of security policies and enforcement regimes. OPC UA, therefore, provides a flexible set of security mechanisms. Figure 1 - OPC UA Network Model is a composite that shows a combination of such environments. Some OPC UA *Clients* and *Servers* are on the same host and can be more easily protected from external attack. Some *Clients* and *Servers* are on different hosts in the same operations network and might be protected by the security boundary protections that separate the operations network from external connections. Some OPC UA *Applications* run in relatively open environments where users and applications might be difficult to control. Other applications are embedded in control systems that have no direct electronic connection to external systems.
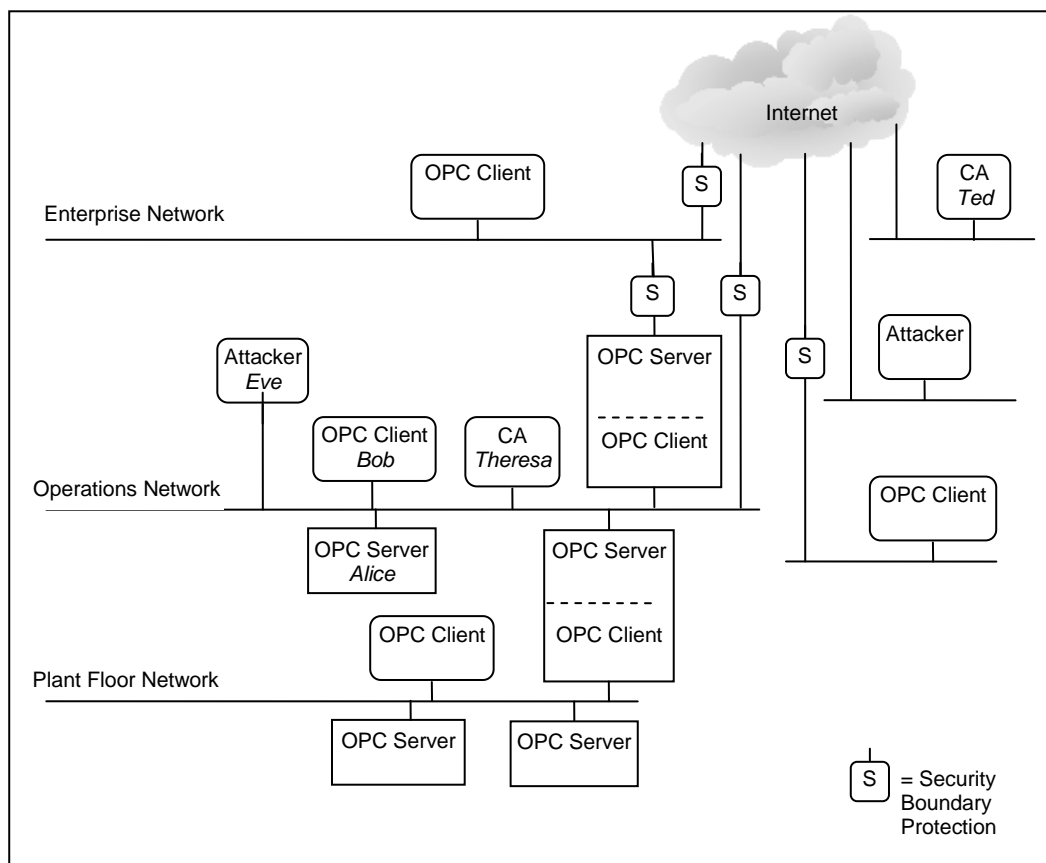

Figure 1 - OPC UA Network Model

T

OPC UA Applications may run at different places in this wide range of environments. *Clients* and *Servers* may communicate within the same host or between hosts within the highly protected control network. Alternatively, OPC UA *Clients* and *Servers* may communicate in the much more open environment over the Internet. The OPC UA activities are protected by whatever security controls the site provides to the parts of the system within which *OPC UA Applications* run.

Fundamentally, information system security reduces the risk of damage from attacks. It does this by identifying the threats to the system, identifying the system's vulnerabilities to these threats, and providing countermeasures. The countermeasures reduce vulnerabilities directly, counteract threats, or recover from successful attacks.

Industrial automation system security is achieved by meeting a set of objectives. These objectives have been refined through many years of experience in providing security for information systems in general and they remain quite constant despite the ever-changing set of threats to systems. They are described in**Error! R eference source not found.**. Following the sections that describe the OPC UA security architecture and functions,**Error! Reference source not found.** reconciles these objectives against the OPC UA functions.
 REF UAPart2 \h * MERGEFORMAT **Error! Reference source not found.** offers additional best practice guidelines to *Client* and *Server* developers or those that deploy *OPC UA Applications*.

4.2.1.2   Authentication

Entities such as clients, *Servers*, and users should prove their identities. *Authentication* can be based on something the entity is, has, or knows.

4.2.1.3   Authorization

The access to read, write, or execute resources should be authorized for only those entities that have a need for that access within the requirements of the system. *Authorization* can be as coarse-grained as allowing or disallowing a *Client* to access a *Server* or it could be much finer grained, such as allowing specific actions on specific information items by specific users.

4.2.1.4   Confidentiality

Data must be protected from passive attacks, such as eavesdropping, whether the data is being transmitted, in memory, or being stored. To provide *Confidentiality,* data encryption algorithms using special secrets for securing data are used along with *Authentication* and *Authorization* mechanisms for accessing that secret.

4.2.1.5   Integrity

Receivers must receive the same information that the sender sent, without the data being changed during transmission.

4.2.1.6   Auditability

Actions taken by a system have to be recorded in order to provide evidence to stakeholders that this system works as intended and to identify the initiator of certain actions and that attempts to compromise the system were denied.

4.2.1.7   Availability

Availability is impaired when the execution of software that needs to run is turned off or when software or the communication system is overwhelmed processing input. Impaired *Availability* in OPC UA can appear as slowing down of *Subscription* performance or inability to add sessions for example.

4.2.2 Security Architecture

The OPC UA security architecture is a generic solution that allows implementation of the required security features at various places in the *OPC UA Application* architecture. Depending on the different mappings described in **Error! Reference source not found.**, the security objectives are addressed at different levels. The OPC UA Security Architecture is structured in an Application Layer and a Communication Layer atop the Transport Layer as shown in Figure 2 – OPC UA Security Architecture.
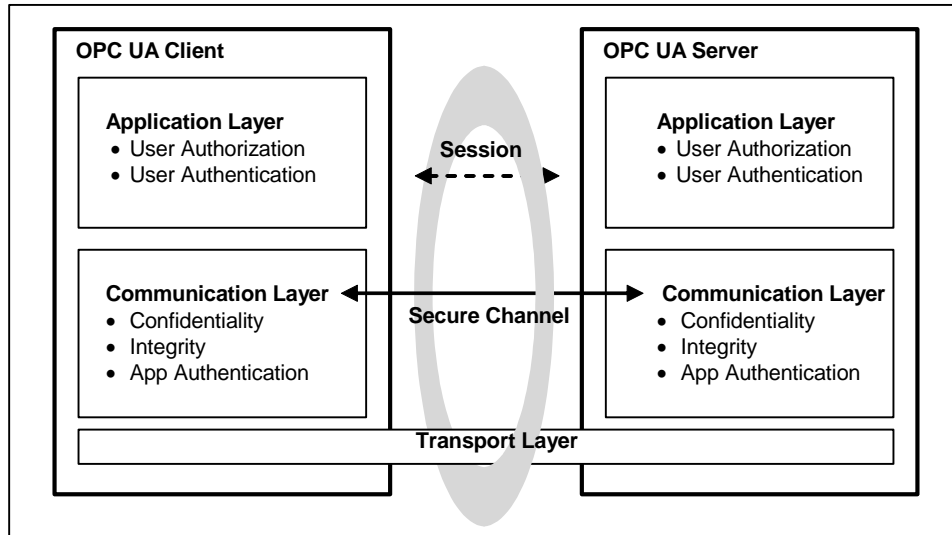

Figure 2 – OPC UA Security Architecture

The routine work of a *Client* application and a *Server* application to transmit information, settings, and commands is done in a session in the Application Layer. The Application Layer also manages the security objectives user *Authentication* and user *Authorization*. The security objectives that are managed by the Application Layer are addressed by the Session Services that are specified in **Error! Reference source not found.**. A session in the Application Layer communicates over a *Secure Channel* that is created in the Communication Layer and relies upon it for secure communication. All of the session data is passed to the Communication Layer for further processing.

Although a session communicates over a *Secure Channel* and has to be activated before it can be used, the binding of users, sessions, and *Secure Channels* is flexible.

Impersonation allows the user of the session to change. A session can have a different user than the user that activated the session for the first time, since user credentials are not validated before activating a session.

When a *Secure Channel* breaks, the session will remain valid and the *Client* will be able to re-establish the *Secure Channel,* otherwise the session closes after its lifetime expires.

The Communication Layer provides security mechanisms to meet *Confidentiality*, *Integrity* and application *Authentication* as security objectives.

One essential mechanism to meet the above mentioned security objectives is to establish a *Secure Channel* (see **Error! Reference source not found.**) that is used to secure the communication between a *Client* and a *Server*. The *Secure Channel* provides encryption to maintain *Confidentiality*, *Message Signatures* to maintain *Integrity* and *Digital Certificates* to provide application *Authentication* for data that comes from the Application Layer and passes the "secured" data to the Transport Layer. The security mechanisms that are managed by the Communication Layer are provided by the *Secure Channel* Services that are specified in **Error! Reference source not found.**.

T

The security mechanisms provided by the *Secure Channel* services are implemented by a protocol stack that is chosen for the implementation. Mappings of the services to some of the protocol stack options are specified in **Error! Reference source not found.**, which details how the functions of the protocol stack are used to meet the OPC UA security objectives.

The Communication Layer can represent an OPC UA protocol stack. OPC UA specifies alternative stack mappings that can be used as the Communication Layer. These mappings are described in **Error! Reference source not found.**.

If the OPC UA Native mapping is used, then functionalities for *Confidentiality*, *Integrity*, application *Authentication*, and the *Secure Channel* are similar to the **Error! Reference source not found.** specifications, as described in detail in **Error! Reference source not found.**.

If the Web Services mapping is used, then **Error! Reference source not found.**, **Error! Reference source not found.** and **Error! Reference source not found.** as well as **Error! Reference source not found.** are used to implement the mechanisms for *Confidentiality*, *Integrity*, application *Authentication* as well as for implementing a *Secure Channel*. For more specific information, see **Error! Reference source not found.**.

The Transport Layer handles the transmission, reception and the transport of data that is provided by the Communication Layer.

To survive the loss of the Transport Layer connections (e.g. TCP connections) and resume with a new connection, the Communication Layer is responsible for re-establishing the Transport Layer connection without interrupting the logical *Secure Channel*.

4.3 Address Space

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 3 – OPC UA Object ModelFigure 3 illustrates the model.
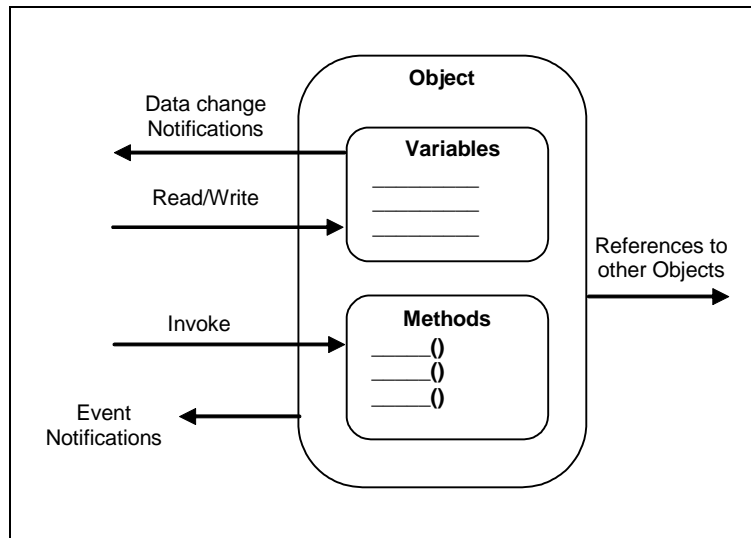
Figure 3 – OPC UA Object Model

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.

4.3.1 Node Model

4.3.1.1 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.

Objects and their components are represented in the *AddressSpace* as a set of *Nodes* described by *Attributes* and interconnected by *References*. Figure 4 – AddressSpace Node Model illustrates the model of a *Node* and the remainder of this subclause discusses the details of the Node Model.
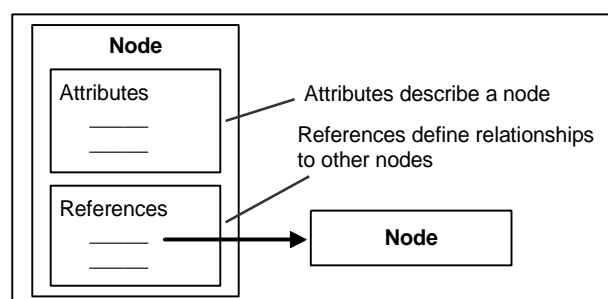

Figure 4 – AddressSpace Node Model

4.3.1.2   NodeClasses

NodeClasses are defined in terms of the *Attributes* and *References* that shall be instantiated (given values) when a *Node* is defined in the *AddressSpace*. *Attributes* are discussed in 4.3.1.3 and *References* in 4.3.1.4.

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

4.3.1.3   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**Error! Reference source not found.**.

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

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

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

4.3.1.4   References

References are used to relate *Nodes* to each other. They can be accessed using the browsing and querying *Services* defined in **Error! Reference source not found.**.

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.*

The *Node* that contains the *Reference* is referred to as the *SourceNode* and the *Node* that is referenced is referred to as the *TargetNode*. The combination of the *SourceNode*, the *ReferenceType* and the *TargetNode* are used in OPC UA *Services* to uniquely identify *References*. Thus, each *Node* can reference another *Node* with the same *ReferenceType* only once. Any subtypes of concrete *ReferenceTypes* are considered to be equal to the base concrete *ReferenceTypes* when identifying *References*. Figure 5 – Reference Model illustrates this model of a *Reference*.
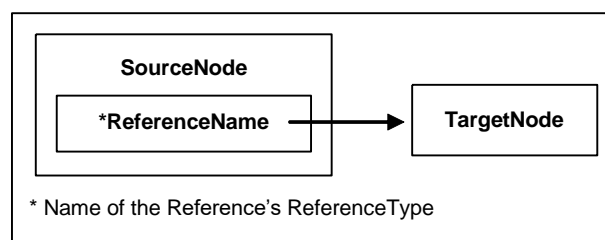
Figure 5 – Reference Model

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

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

4.3.2 Variables

4.3.2.1 General

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

4.3.2.2 Properties

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

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

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

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

4.3.2.3 DataVariables

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

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

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

DataVariables may have additional *DataVariables*, but only if they are complex. In this case, their *DataVariables* shall always be elements of their complex definitions. Following the example introduced by the description of *Properties* in 4.3.2.2, 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.

4.3.3 TypeDefinitionNodes

4.3.3.1 General

OPC UA *Servers* shall provide type definitions for *Objects* and *Variables*. The *HasTypeDefinition Reference* shall be used to link an instance with its type definition represented by a *TypeDefinitionNode*. Type definitions are required, however, **Error! Reference source not found.** defines a *BaseObjectType*, a *PropertyType,* and a *BaseDataVariableType* so a *Server* can use such a base type if no more specialised type information is available. *Objects* and *Variables* inherit the *Attributes* specified by their *TypeDefinitionNode*.

In some cases, the *NodeId* used by the *HasTypeDefinition Reference* will be well-known to *Clients* and *Servers*. Organizations may define *TypeDefinitionNodes* that are well-known in the industry. Well-known *NodeIds* of *TypeDefinitionNodes* provide for commonality across OPC UA *Servers* and allow *Clients* to interpret the *TypeDefinitionNode* without having to read it from the *Server*. Therefore, *Servers* may use well-known *NodeIds*

without representing the corresponding *TypeDefinitionNodes* in their *AddressSpace*. However, the *TypeDefinitionNodes* shall be provided for generic *Clients*. These *TypeDefinitionNodes* may exist in another *Server*.

The following example, illustrated in Figure 6 – Example of a Variable Defined by a VariableType, 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*.
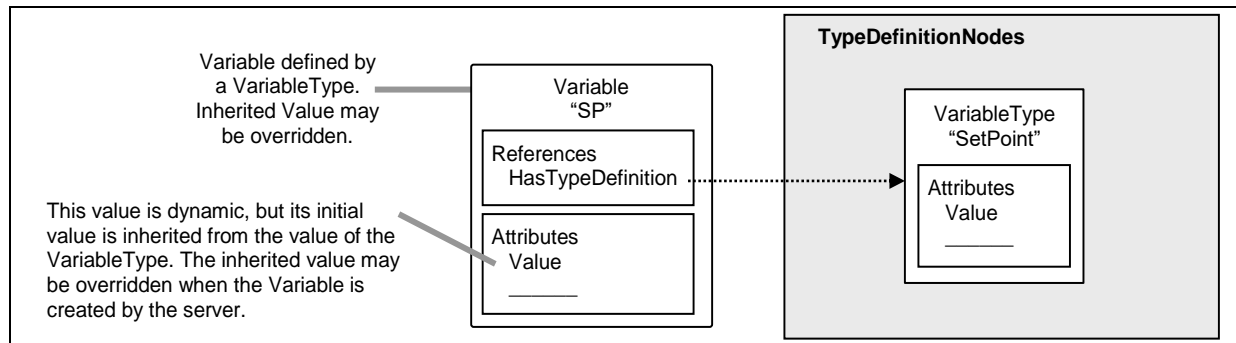

Figure 6 – Example of a Variable Defined by a VariableType

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

A *TypeDefinitionNode* references instances instead of other *TypeDefinitionNodes* to allow unique names for several instances of the same type, to define default values and to add *References* for those instances that are specific to this complex *TypeDefinitionNode* and not to the *TypeDefinitionNode* of the instance. For example, in Figure 7 – Example of a Complex TypeDefinition 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.
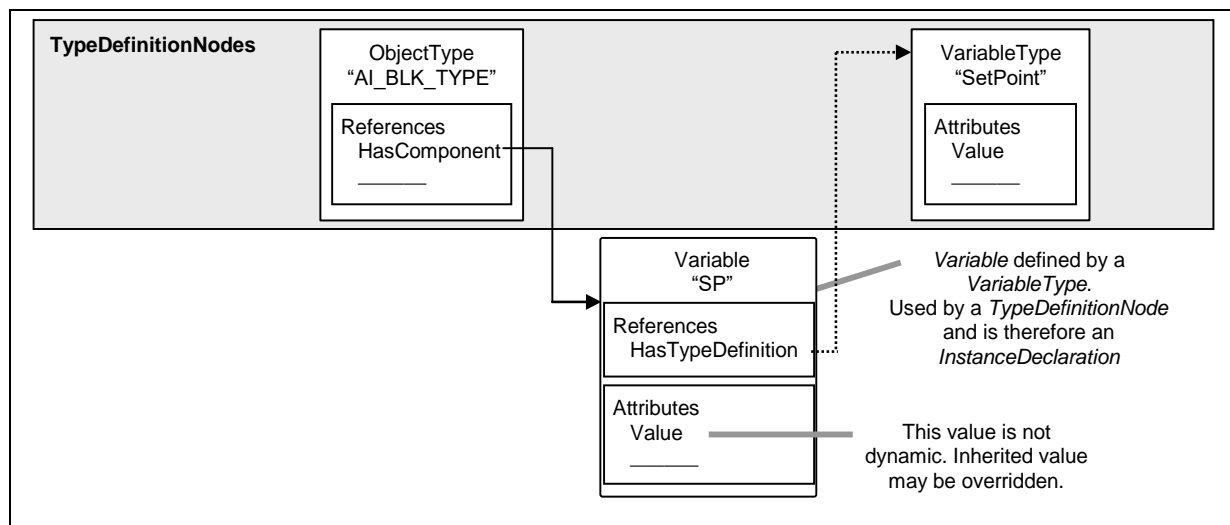

Figure 7 – Example of a Complex TypeDefinition

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.

4.3.3.3 Subtyping

This standard allows subtyping of type definitions. The subtyping rules are defined in **Error! Reference s
ource not found.**. Subtyping of *ObjectTypes* and *VariableTypes* allows:

- *Clients* that only know the supertype are able to handle an instance of the subtype as if it were an instance of the supertype;

- instances of the supertype can be replaced by instances of the subtype;

- specialised types that inherit common characteristics of the base type.

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

4.3.3.4 Instantiation of complex TypeDefinitionNodes

The instantiation of complex *TypeDefinitionNodes* depends on the *ModellingRules*. However, the intention is that instances of a type definition will reflect the structure defined by the *TypeDefinitionNode*. Figure 8 – Object and its Components defined by an ObjectType shows an instance of the *TypeDefinitionNode* "AI_BLK_TYPE", where the *ModellingRule Mandatory* 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*.

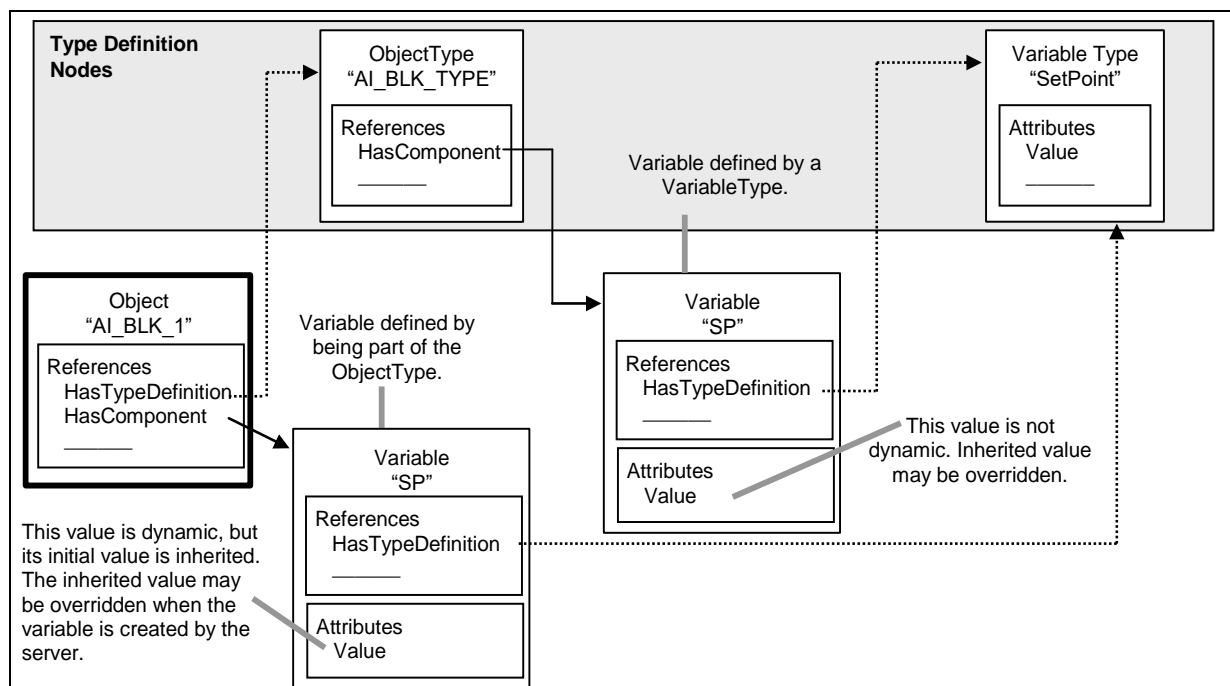**Figure 8 – Object and its Components defined by an ObjectType**

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".

T

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

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

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

4.3.4 Event Model

4.3.4.1 General

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

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

Event Subscriptions use the Monitoring and Subscription *Services* defined in **Error! Reference source not f ound.** to subscribe to *Event Notifications* of a *Node*.

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

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

T

4.3.4.2 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. **Error! Reference source not found.** defines how a *Server* exposes the *EventTypes* in detail.

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

Standard *EventTypes* are described in **Error! Reference source not found.**. Their representation in the *A ddressSpace* is specified in **Error! Reference source not found.**.

4.3.4.3 Event Categorization

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

Event sources can also be organised into groups by using the *Event ReferenceTypes*. 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.

T

4.3.5 Methods

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

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

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

Methods are invoked by using the Call *Service* defined in**Error! Reference source not found.**. Each *Method* is i nvoked 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*.

4.4 Services

4.4.1 Service Set Model

The OPC UA *Service* definitions are abstract descriptions and do not represent a specification for implementation. The mapping between the abstract descriptions and the *Communication Stack* derived from these *Services* are defined in **Error! Reference source not found.**. In the case of an implementation as web s ervices, the OPC UA *Services* correspond to the web service and an OPC UA *Service* corresponds to an operation of the web service.

These *Services* are organised into *Service Sets*. Each *Service Set* defines a set of related *Services*. The organisation in *Service Sets* is a logical grouping used in this standard and is not used in the implementation.

The *Discovery Service Set*, illustrated in Figure 9 – Discovery Service Set, defines *Services* that allow a *Client* to discover the *Endpoints* implemented by a *Server* and to read the security configuration for each of those *Endpoints*.
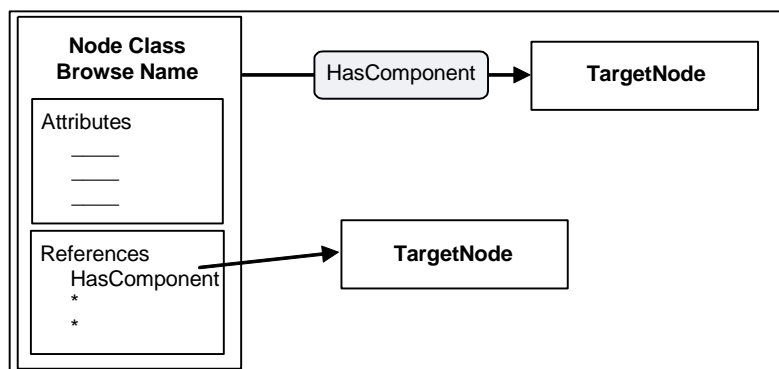

Figure 9 – Discovery Service Set

The *SecureChannel Service Set*, illustrated in Figure 10 – SecureChannel Service Set , defines *Services* that allow a *Client* to establish a communication channel to ensure the *Confidentiality* and *Integrity* of *Messages* exchanged with the *Server*.
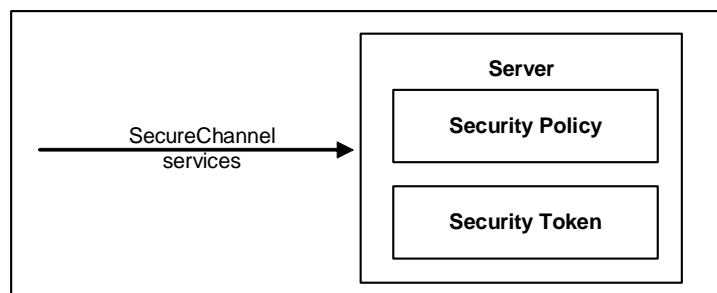
Figure 10 – SecureChannel Service Set

The *Session Service Set*, illustrated in Figure 11 – Session Service Set, defines *Services* that allow the *Client* to authenticate the user on whose behalf it is acting and to manage *Sessions*.
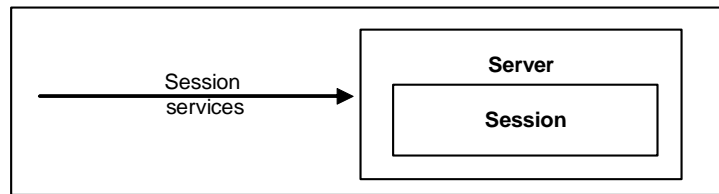
Figure 11 – Session Service Set

The *NodeManagement Service Set*, illustrated in Figure 12 – NodeManagement Service Set, defines *Services* that allow the *Client* to add, modify and delete *Nodes* in the *AddressSpace*.
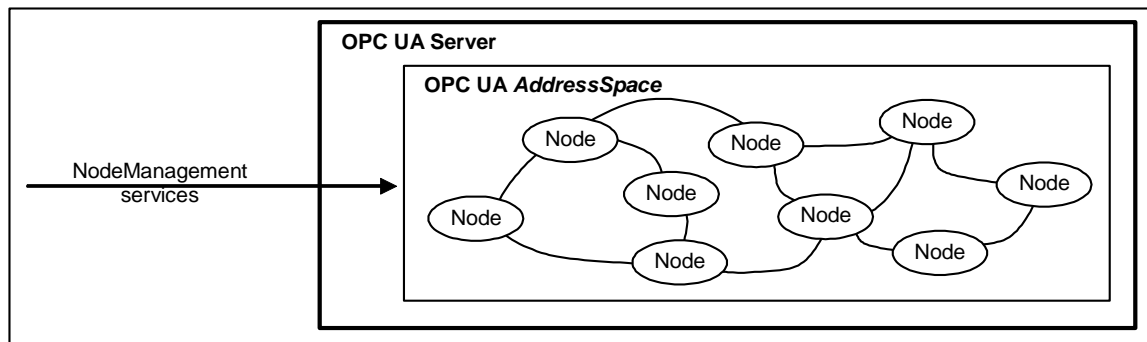
Figure 12 – NodeManagement Service Set

The *View Service Set*, illustrated in Figure 13 – View Service Set, defines *Services* that allow *Clients* to browse through the *AddressSpace* or subsets of the *AddressSpace* called *Views*. The *Query Service Set* allows *Clients* to get a subset of data from the *AddressSpace* or the *View*.
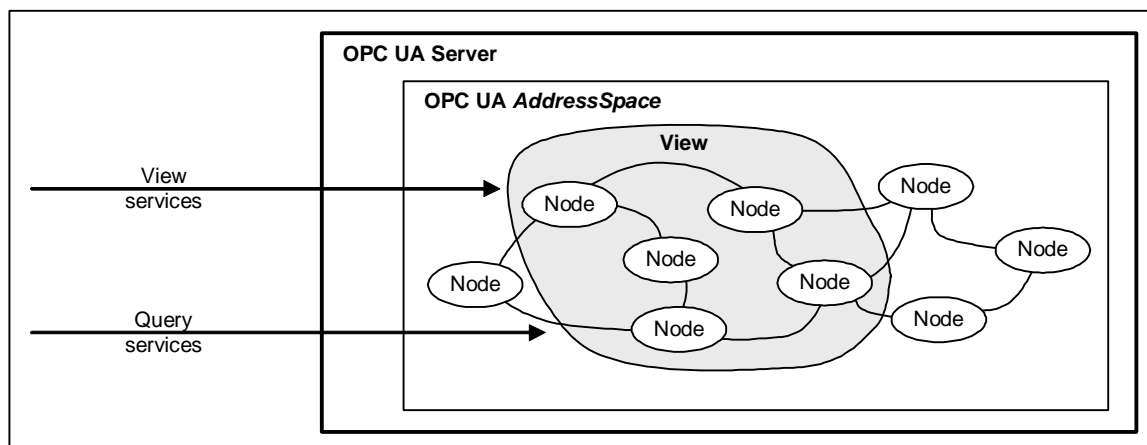
Figure 13 – View Service Set

The *Attribute Service Set* is illustrated in Figure 14 – Attribute Service Set . It defines *Services* that allow *Clients* to read and write *Attributes* of *Nodes*, including their historical values. Since the value of a *Variable* is modelled as an *Attribute*, these *Services* allow *Clients* to read and write the values of *Variables*.

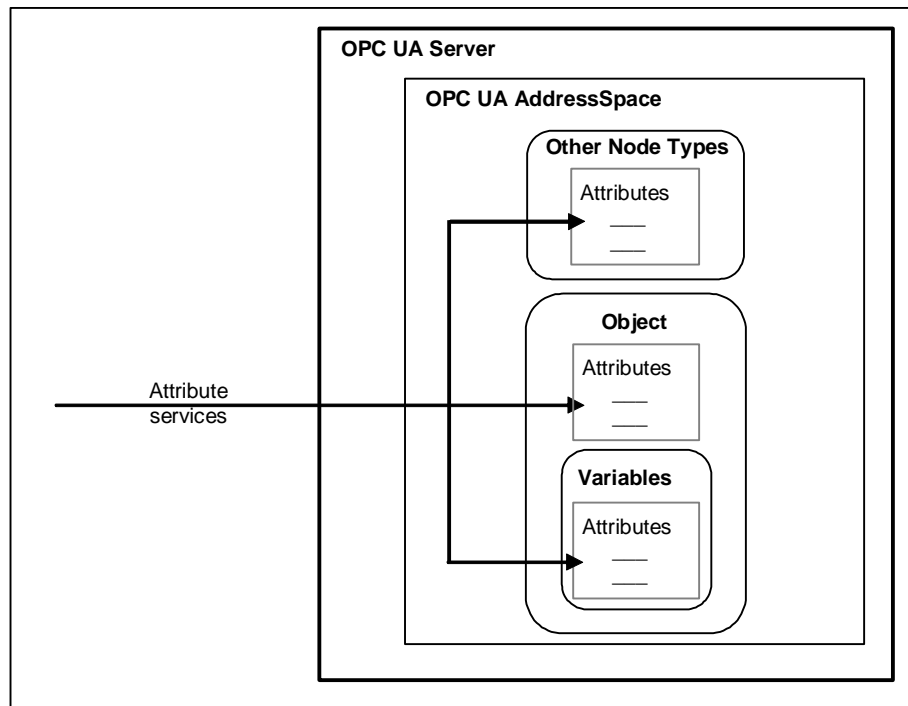**Figure 14 – Attribute Service Set**

The *Method Service Set* is illustrated in Figure 15 – Method Service Set. It defines *Services* that allow *Clients* to call methods. Methods run to completion when called. They may be called with method-specific input parameters and may return method-specific output parameters.
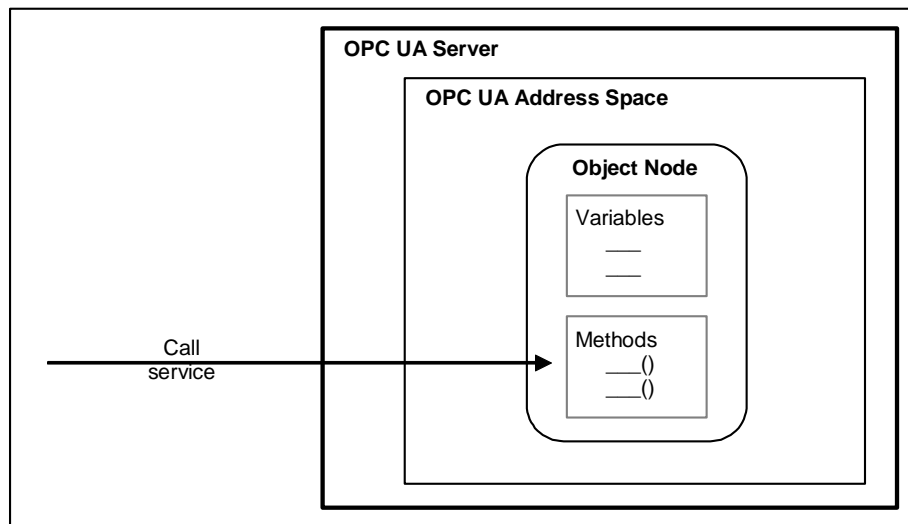

Figure 15 – Method Service Set

The *MonitoredItem Service Set* and the *Subscription Service Set*, illustrated in Figure 16 – MonitoredItem and Subscription Service Sets, are used together to subscribe to *Nodes* in the OPC UA *AddressSpace*.

The *MonitoredItem Service Set* defines *Services* that allow *Clients* to create, modify, and delete *MonitoredItems* used to monitor *Attribute*s for value changes and *Objects* for *Events*.

These *Notifications* are queued for transfer to the *Client* by *Subscriptions*.

The *Subscription Service Set* defines *Services* that allow *Clients* to create, modify and delete *Subscriptions*. *Subscriptions* send *Notifications* generated by *MonitoredItems* to the *Client*. *Subscription Services* also provide for *Client* recovery from missed *Messages* and communication failures.
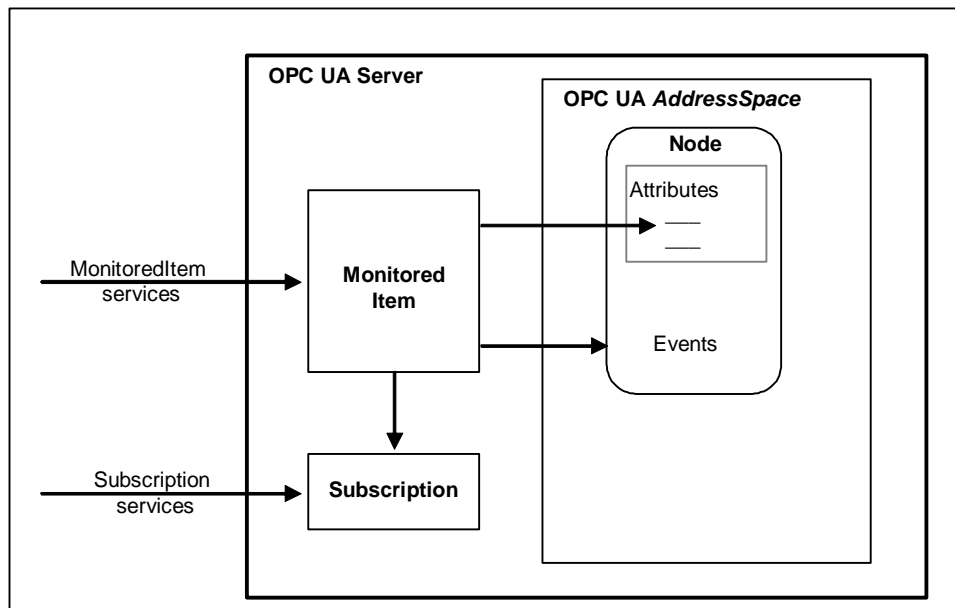
Figure 16 – MonitoredItem and Subscription Service Sets

4.4.2 Request/response Service procedures

Request/response *Service* procedures describe the processing of requests received by the *Server*, and the subsequent return of responses. The procedures begin with the requesting *Client* submitting a *Service* request *Message* to the *Server*.

Upon receipt of the request, the *Server* processes the *Message* in two steps. In the first step, it attempts to decode and locate the *Service* to execute. The error handling for this step is specific to the communication technology used and is described in **Error! Reference source not found.**.

If it succeeds, then it attempts to access each operation identified in the request and perform the requested operation. For each operation in the request, it generates a separate success/failure code that it includes in a positive response *Message* along with any data that is to be returned.

To perform these operations, both the *Client* and the *Server* may make use of the API of a *Communication Stack* to construct and interpret *Messages* and to access the requested operation.

The implementation of each service request or response handling shall check that each service parameter lies within the specified range for that parameter.

T

4.4.3 MonitoredItem Service Set

4.4.3.1 MonitoredItem model

4.4.3.1.1 Overview

Clients define *MonitoredItems* to subscribe to data and *Events*. Each *MonitoredItem* identifies the item to be monitored and the *Subscription* to use to send *Notifications*. The item to be monitored may be any *Node Attribute*.

Notifications are data structures that describe the occurrence of data changes and *Events*. They are packaged into *NotificationMessages* for transfer to the *Client*. The *Subscription* periodically sends *NotificationMessages* at a user-specified publishing interval, and the cycle during which these messages are sent is called a publishing cycle.

Four primary parameters are defined for *MonitoredItems* that tell the *Server* how the item is to be sampled, evaluated and reported. These parameters are the sampling interval, the monitoring mode, the filter and the queue parameter. Figure 17 – MonitoredItem Model illustrates these concepts.
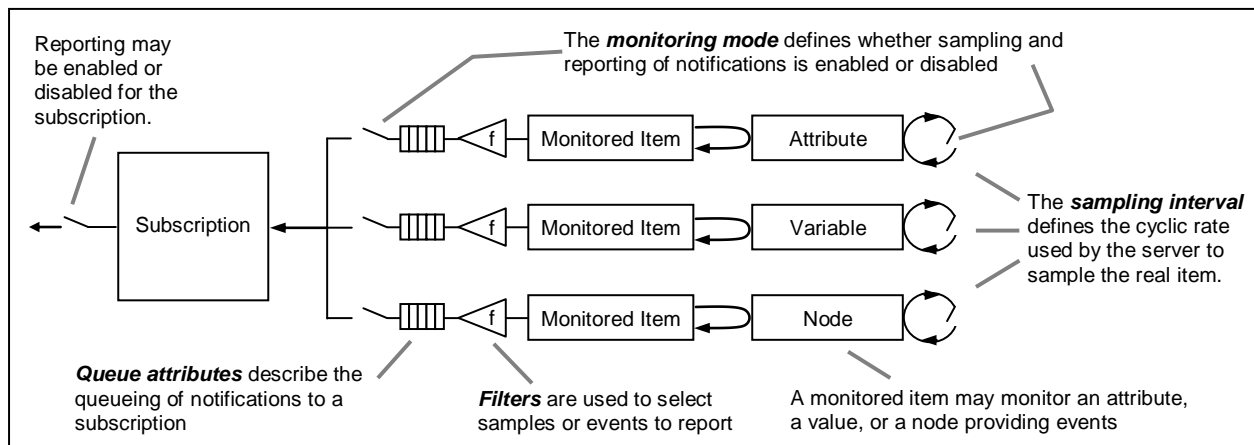
Figure 17 – MonitoredItem Model

Attributes, other than the *Value Attribute*, are only monitored for a change in value. The filter is not used for these *Attributes*. Any change in value for these *Attributes* causes a *Notification* to be generated.

The *Value Attribute* is used when monitoring *Variables*. *Variable* values are monitored for a change in value or a change in their status. The filters defined in the OPC UA Specification **Error! Reference source not found.** and in **Error! Reference source not found.** are used to determine if the value change is large enough to cause a *Notification* to be generated for the *Variable*.

Objects and views can be used to monitor *Events*. *Events* are only available from *Nodes* where the *SubscribeToEvents* bit of the *EventNotifier Attribute* is set. The filter defined in in the OPC UA Specification **Error! Reference source not found.** is used to determine if an *Event* received from the *Node* is sent to the *Client*. The filter also allows selecting fields of the *EventType* that will be contained in the *Event* such as *EventId*, *EventType*, *SourceNode*, *Time* and *Description*.

OPC UA Specification **Error! Reference source not found.** describes the *Event* model and the base *EventTypes*.

T

The *Properties* of the base *EventTypes* and the representation of the base *EventTypes* in the *AddressSpace* are specified in OPC UA Specification **Error! Reference source not found.**.

4.4.3.1.2 Sampling interval

Each *MonitoredItem* created by the *Client* is assigned a sampling interval that is either inherited from the publishing interval of the *Subscription* or that is defined specifically to override that rate. A negative number indicates that the default sampling interval defined by the publishing interval of the *Subscription* is requested. The sampling interval indicates the fastest rate at which the *Server* should sample its underlying source for data changes.

A *Client* shall define a sampling interval of 0 if it subscribes for *Events*.

The assigned sampling interval defines a "best effort" cyclic rate that the *Server* uses to sample the item from its source. "Best effort" in this context means that the *Server* does its best to sample at this rate. Sampling at rates faster than this rate is acceptable, but not necessary to meet the needs of the *Client*. How the *Server* deals with the sampling rate and how often it actually polls its data source internally is a *Server* implementation detail. However, the time between values returned to the *Client* shall be greater or equal to the sampling interval.

The *Client* may also specify 0 for the sampling interval, which indicates that the *Server* should use the fastest practical rate. It is expected that *Servers* will support only a limited set of sampling intervals to optimize their operation. If the exact interval requested by the *Client* is not supported by the *Server*, then the *Server* assigns to the *MonitoredItem* the most appropriate interval as determined by the *Server*. It returns this assigned interval to the *Client*. The *Server* Capabilities *Object* defined in **Error! R eference source not found.** identifies the sampling intervals supported by the *Server*.

The *Server* may support data that is collected based on a sampling model or generated based on an exception-based model. The fastest supported sampling interval may be equal to 0, which indicates that the data item is exception-based rather than being sampled at some period. An exception-based model means that the underlying system does not require sampling and reports data changes.

The *Client* may use the revised sampling interval values as a hint for setting the publishing interval as well as the keep-alive count of a *Subscription*. If, for example, the smallest revised sampling interval of the *MonitoredItems* is 5 seconds, then the time before a keep-alive is sent should be longer than 5 seconds.

Note that, in many cases, the OPC UA *Server* provides access to a decoupled system and therefore has no knowledge of the data update logic. In this case, even though the OPC UA *Server* samples at the negotiated rate, the data might be updated by the underlying system at a much slower rate. In this case, changes can only be detected at this slower rate.

If the behaviour by which the underlying system updates the item is known, it will be available via the *MinimumSamplingInterval Attribute* defined in **Error! Reference source not found.**. If the *Server* s pecifies a value for the *MinimumSamplingInterval Attribute* it shall always return a *revisedSamplingInterval* that is equal or higher than the *MinimumSamplingInterval* if the *Client* subscribes to the *Value Attribute*.

Clients should also be aware that the sampling by the OPC UA *Server* and the update cycle of the underlying system are usually not synchronized. This can cause additional delays in change detection, as illustrated in Figure 18 – Typical delay in change detection.
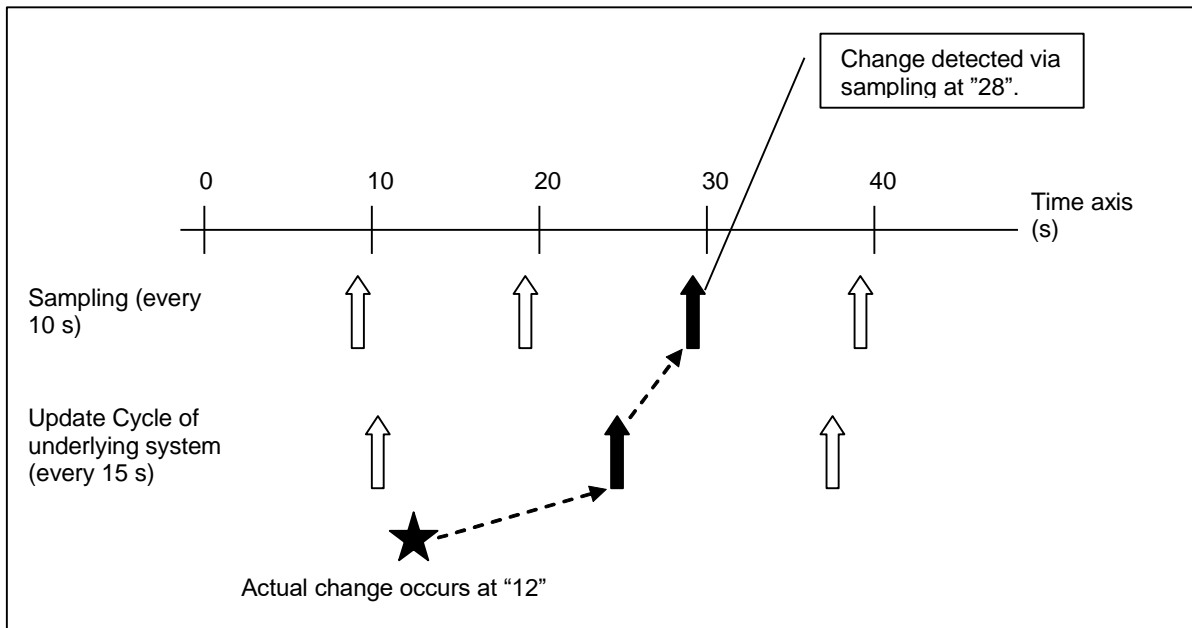
Figure 18 – Typical delay in change detection

4.4.3.1.3 *Monitoring mode*

The monitoring mode parameter is used to enable and disable the sampling of a *MonitoredItem*, and also to provide for independently enabling and disabling the reporting of *Notifications*. This capability allows a *MonitoredItem* to be configured to sample, sample and report, or neither. Disabling sampling does not change the values of any of the other *MonitoredItem* parameter, such as its sampling interval.

When a *MonitoredItem* is enabled (i.e. when the *MonitoringMode* is changed from *DISABLED* to *SAMPLING* or *REPORTING*) or it is created in the enabled state, the *Server* shall report the first sample as soon as possible and the time of this sample becomes the starting point for the next sampling interval.

4.4.3.1.4 *Filter*

Each time a *MonitoredItem* is sampled, the *Server* evaluates the sample using the filter defined for the *MonitoredItem*. The filter parameter defines the criteria that the *Server* uses to determine if a *Notification* should be generated for the sample. The type of filter is dependent on the type of the item that is being monitored. For example, the *DataChangeFilter* and the *AggregateFilter* are used when monitoring *Variable Values* and the *EventFilter* is used when monitoring *Events*. Sampling and evaluation, including the use of filters, are described in this standard. Additional filters may be defined in other parts of this series of standards.

4.4.3.1.5   Queue parameters

If the sample passes the filter criteria, a *Notification* is generated and queued for transfer by the *Subscription*. The size of the queue is defined when the *MonitoredItem* is created. When the queue is full and a new *Notification* is received, the *Server* either discards the oldest *Notification* and queues the new one, or it replaces the last value added to the queue with the new one. The *MonitoredItem* is configured for one of these discard policies when the *MonitoredItem* is created. If a Notification is discarded for a *DataValue* and the size of the queue is larger than one, then the *Overflow* bit (flag) in the *InfoBits* portion of the *DataValue statusCode* is set. If *discardOldest* is TRUE, the oldest value gets deleted from the queue and the next value in the queue gets the flag set. If *discardOldest* is FALSE, the last value added to the queue gets replaced with the new value. The new value gets the flag set to indicate the lost values in the next *NotificationMessage*. Figure 19 – Queue overflow handling illustrates the queue overflow handling.
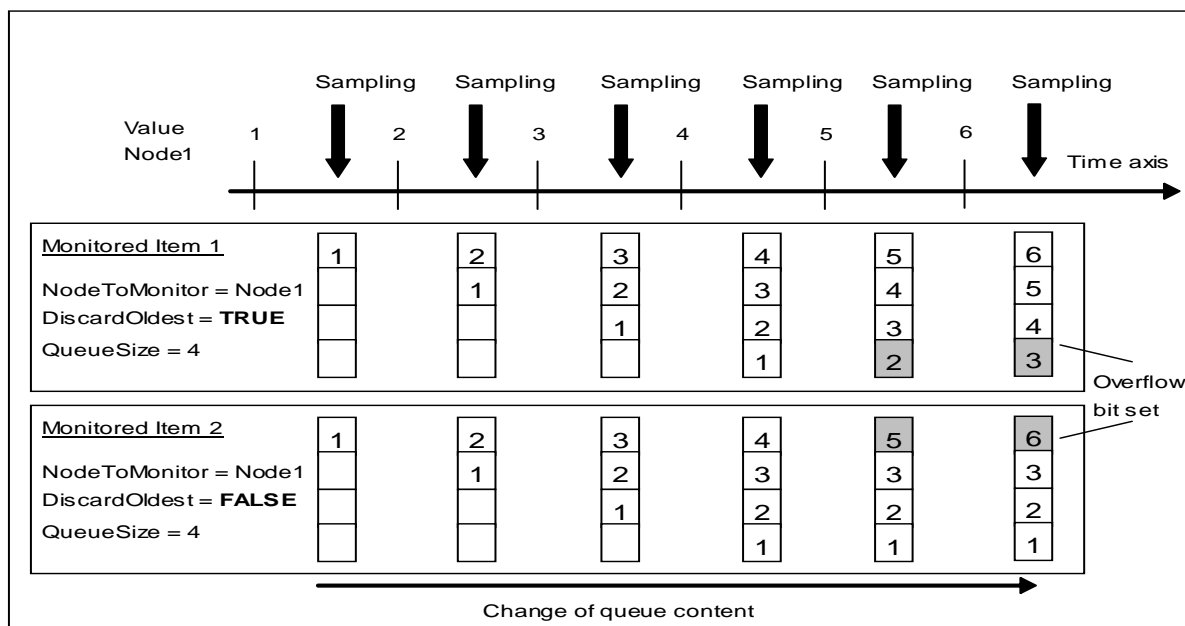
Figure 19 – Queue overflow handling

If the queue size is one, the queue becomes a buffer that always contains the newest *Notification*. In this case, if the sampling interval of the *MonitoredItem* is faster than the publishing interval of the *Subscription*, the *MonitoredItem* will be over sampling and the *Client* will always receive the most up-to-date value. The discard policy is ignored if the queue size is one.

On the other hand, the *Client* may want to subscribe to a continuous stream of *Notifications* without any gaps, but does not want them reported at the sampling interval. In this case, the *MonitoredItem* would be created with a queue size large enough to hold all *Notifications* generated between two consecutive publishing cycles. Then, at each publishing cycle, the *Subscription* would send all *Notifications* queued for the *MonitoredItem* to the *Client*. The *Server* shall return *Notifications* for any particular item in the same order they are in the queue.

The *Server* may be sampling at a faster rate than the sampling interval to support other *Clients*; the *Client* should only expect values at the negotiated sampling interval. The *Server* may deliver fewer values than dictated by the sampling interval, based on the filter and implementation constraints. If a *DataChangeFilter* is configured for a *MonitoredItem*, it is always applied to the newest value in the queue compared to the current sample.

If, for example, the *AbsoluteDeadband* in the *DataChangeFilter* is "10", the queue could consist of values in the following order:

- 100
- 111
- 100
- 89
- 100

Queuing of data may result in unexpected behaviour when using a *Deadband* filter and the number of encountered changes is larger than the number of values that can be maintained. The new first value in the queue may not exceed the *Deadband* limit of the previous value sent to the *Client*.

The queue size is the maximum value supported by the *Server* when monitoring *Events*. In this case, the *Server* is responsible for the *Event* buffer. If *Events* are lost, an *Event* of the type *EventQueueOverflowEventType* is placed in the queue. This Event is generated when the first Event has to be discarded on a MonitoredItem subscribing for Events. It is put into the Queue of the MonitoredItem in addition to the size of the Queue defined for this MonitoredItem without discarding any other Event. If *discardOldest* is set to TRUE it is put at the beginning of the queue and is never discarded, otherwise at the end. An aggregating *Server* shall not pass on such an *Event*. It shall be handled like other connection error scenarios.

4.4.3.1.6   Triggering model

The *MonitoredItems Service* allows adding items that are reported only when some other item (the triggering item) triggers. This is done by creating links between the triggered items and the items to report. The monitoring mode of the items to report is set to sampling-only so that it will sample and queue *Notifications* without reporting them. Figure 20 – Triggering Model illustrates this concept.
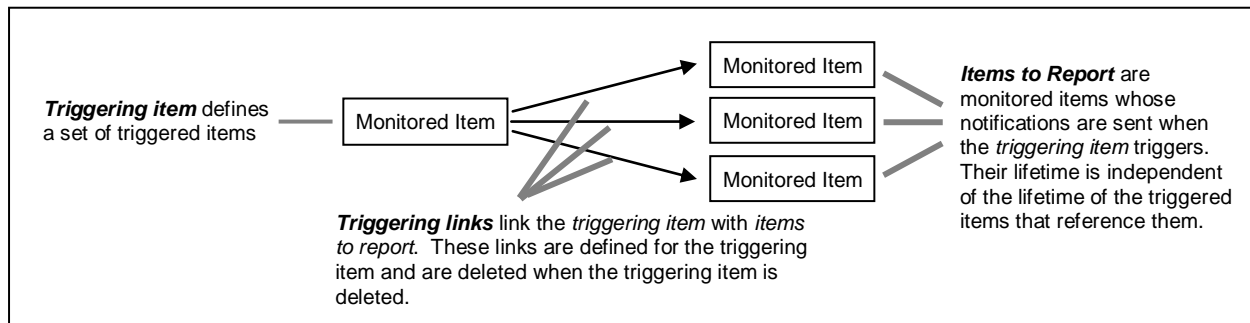

Figure 20 – Triggering Model

The triggering mechanism is a useful feature that allows *Clients* to reduce the data volume on the wire by configuring some items to sample frequently but only report when some other *Event* happens.

T

The following triggering behaviours are specified.

a) If the monitoring mode of the triggering item is *SAMPLING, then* it is not reported when the triggering item triggers the items to report.

b) If the monitoring mode of the triggering item is *REPORTING, then it* is reported when the triggering item triggers the items to report.

c) If the monitoring mode of the triggering item is *DISABLED*, then the triggering item does not trigger the items to report.

d) If the monitoring mode of the item to report is *SAMPLING, then it* is reported when the triggering item triggers the items to report.

e) If the monitoring mode of the item to report is *REPORTING*, this effectively causes the triggering item to be ignored. All notifications of the items to report are sent after the publishing interval expires.

f) If the monitoring mode of the item to report is *DISABLED*, then there will be no sampling of the item to report and therefore no notifications to report.

g) The first trigger shall occur when the first notification is queued for the triggering item after the creation of the link.

Clients create and delete triggering links between a triggering item and a set of items to report. If the *MonitoredItem* that represents an item to report is deleted before its associated triggering link is deleted, the triggering link is also deleted, but the triggering item is otherwise unaffected.

Deletion of a *MonitoredItem* should not be confused with the removal of the *Attribute* that it monitors. If the *Node* that contains the *Attribute* being monitored is deleted, the *MonitoredItem* generates a *Notification* with a *StatusCode Bad_NodeIdUnknown* that indicates the deletion, but the *MonitoredItem* is not deleted.

4.4.4 Subscription Service Set

4.4.4.1 Subscription model

4.4.4.1.1 Overview

Subscriptions are used to report *Notifications* to the *Client*. Their general behaviour is summarized below. Their precise behaviour is described in the OPC UA Specification **Error! Reference source not found.**.

h) *Subscriptions* have a set of *MonitoredItems* assigned to them by the *Client*. *MonitoredItems* generate *Notifications* that are to be reported to the *Client* by the *Subscription* (see **Error! Reference source not found.** for a description of *MonitoredItems*).

i) *Subscriptions* have a publishing interval. The publishing interval of a *Subscription* defines the cyclic rate at which the *Subscription* executes. Each time it executes, it attempts to send a *NotificationMessage* to the *Client*. *NotificationMessages* contain *Notifications* that have not yet been reported to *Client*.

j) *NotificationMessages* are sent to the *Client* in response to *Publish* requests. *Publish* requests are normally queued to the *Session* as they are received, and one is de-queued and processed by a subscription related to this *Session* for each publishing cycle, if there are *Notifications* to report. When there are not, the *Publish* request is not de-queued from the *Session*, and the *Server* waits until the next cycle and checks again for *Notifications*.

k) At the beginning of a cycle, if there are *Notifications* to send but there are no *Publish* requests queued, the *Server* enters a wait state for a *Publish* request to be received. When one is received, it is processed immediately without waiting for the next publishing cycle.

l) *NotificationMessages* are uniquely identified by sequence numbers that enable *Clients* to detect missed *Messages*. The publishing interval also defines the default sampling interval for its *MonitoredItems*.

m) *Subscriptions* have a keep-alive counter that counts the number of consecutive publishing cycles in which there have been no *Notifications* to report to the *Client*. When the maximum keep-alive count is reached, a *Publish* request is de-queued and used to return a keep-alive *Message*. This keep-alive *Message* informs the *Client* that the *Subscription* is still active. Each keep-alive *Message* is a response to a Publish request in which the *notificationMessage* parameter does not contain any Notifications and that contains the sequence number of the next *NotificationMessage* that is to be sent. In the clauses that follow, the term *NotificationMessage* refers to a response to a Publish request in which the *notificationMessage* parameter actually contains one or more *Notifications*, as opposed to a *keep-alive Message* in which this parameter contains no *Notifications*. The maximum keep-alive count is set by the *Client* during *Subscription* creation and may be subsequently modified using the *ModifySubscription Service*. Similar to *Notification* processing described in (c) above, if there are no *Publish* requests queued, the *Server* waits for the next one to be received and sends the keep-alive immediately without waiting for the next publishing cycle.

n) Publishing by a *Subscription* may be enabled or disabled by the *Client* when created, or subsequently using the *SetPublishingMode Service*. Disabling causes the *Subscription* to cease sending *NotificationMessages* to the *Client*. However, the *Subscription* continues to execute cyclically and continues to send keep-alive *Messages* to the *Client*.

o) *Subscriptions* have a lifetime counter that counts the number of consecutive publishing cycles in which there have been no *Publish* requests available to send a *Publish* response for the *Subscription*. Any *Service* call that uses the *SubscriptionId* or the processing of a *Publish* response resets the lifetime counter of this *Subscription*. When this counter reaches the value calculated for the lifetime of a *Subscription* based on the MaxKeepAliveCount parameter in the *CreateSubscription Service* (**Error! Reference source not found.**), the *Subscription* is closed. Closing the *Subscription* causes its *MonitoredItems* to be deleted. In addition the

T

Server shall issue a *StatusChangeNotification notificationMessage* with the status code Bad_Timeout. The *StatusChangeNotification notificationMessage* type is defined in **Error! Reference source not found.**.

p) *Sessions* maintain a retransmission queue of sent *NotificationMessages*. *NotificationMessages* are retained in this queue until they are acknowledged. The *Session* shall maintain a retransmission queue size of at least two times the number of *Publish* requests per *Session* the *Server* supports. The minimum size of the retransmission queue may be changed by a *Profile* in **Error! Reference source not found.**. The minimum n umber of *Publish* requests per *Session* the *Server* shall support is defined in **Error! Reference source not found.**. *Clients* are required to acknowledge *NotificationMessages* as they are received. In the case of a retransmission queue overflow, the oldest sent NotificationMessage gets deleted. If a *Subscription* is transferred to another *Session*, the queued *NotificationMessages* for this *Subscription* are moved from the old to the new *Session*.

The sequence number is an unsigned 32-bit integer that is incremented by one for each *NotificationMessage* sent. The value 0 is never used for the sequence number. The first *NotificationMessage* sent on a *Subscription* has a sequence number of 1. If the sequence number rolls over, it rolls over to 1.

When a *Subscription* is created, the first *Message* is sent at the end of the first publishing cycle to inform the *Client* that the *Subscription* is operational. A *NotificationMessage* is sent if there are *Notifications* ready to be reported. If there are none, a keep-alive *Message* is sent instead that contains a sequence number of 1, indicating that the first *NotificationMessage* has not yet been sent. This is the only time a keep-alive *Message* is sent without waiting for the maximum keep-alive count to be reached, as specified in (f) above.

The value of the sequence number is never reset during the lifetime of a *Subscription*. Therefore, the same sequence number shall not be reused on a *Subscription* until over four billion *NotificationMessages* have been sent. At a continuous rate of one thousand *NotificationMessages* per second on a given *Subscription*, it would take roughly fifty days for the same sequence number to be reused. This allows *Clients* to safely treat sequence numbers as unique.

Sequence numbers are also used by *Clients* to acknowledge the receipt of *NotificationMessages*. *Publish* requests allow the *Client* to acknowledge all *Notifications* up to a specific sequence number and to acknowledge the sequence number of the last *NotificationMessage* received. One or more gaps may exist in between. Acknowledgements allow the *Server* to delete *NotificationMessages* from its retransmission queue.

Clients may ask for retransmission of selected *NotificationMessages* using the Republish *Service*. This *Service* returns the requested *Message*.

4.5 Information Model

The base OPC UA specifications provide only the infrastructure to model information. The information can be modeled by vendors in different ways. To avoid that situation, the OPC UA specification provides possibilities to define Information Model specifications based on OPC UA. The base principles of the OPC UA information modeling are:

- Using object-oriented techniques including type hierarchies and inheritance.

- Type information is exposed and can be accessed the same way as instances.

- Full meshed network of nodes allowing information to be connected in various ways.

- Extensibility regarding the type hierarchies as well as the types of references.

- No limitation on how to model information in order to allow an appropriate model for the provided data.

- OPC UA information modeling is always done on server-side.

4.5.1 Nodes

The base modeling concepts of OPC UA are *Nodes* and *References* between *Nodes*.

Nodes can include *Variables*, *Events*, *Methods*, *History* and the functionality of *Nodes* can be extended.
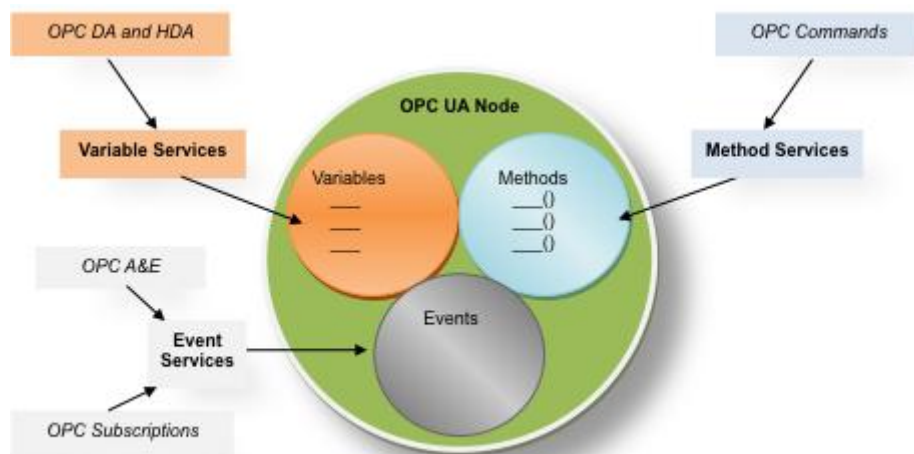
4.5.2 References

Nodes are hierarchical organized and can reference to other nodes:
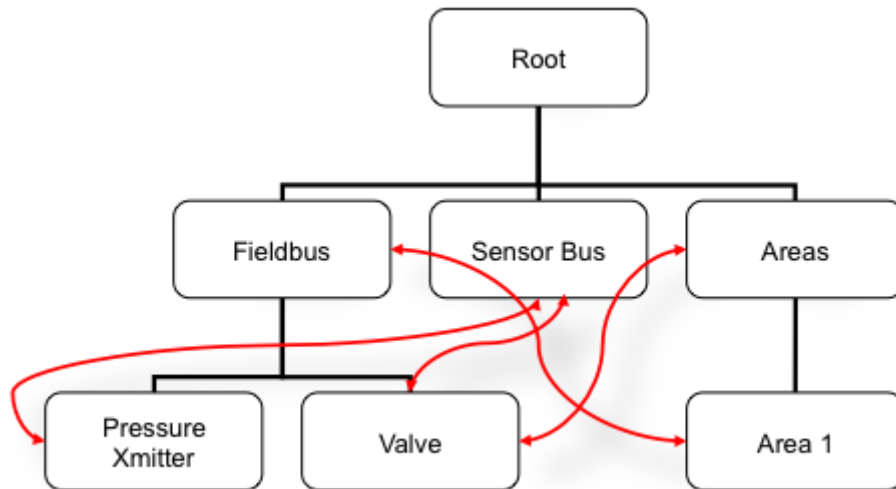
4.5.3 Standard AddressSpace

OPC UA defines a standard *AddressSpace* including

- *Views* (like database views^)
- *Objects* (the Nodes worked with)
- *Types* (Data Types supported by the server)

T
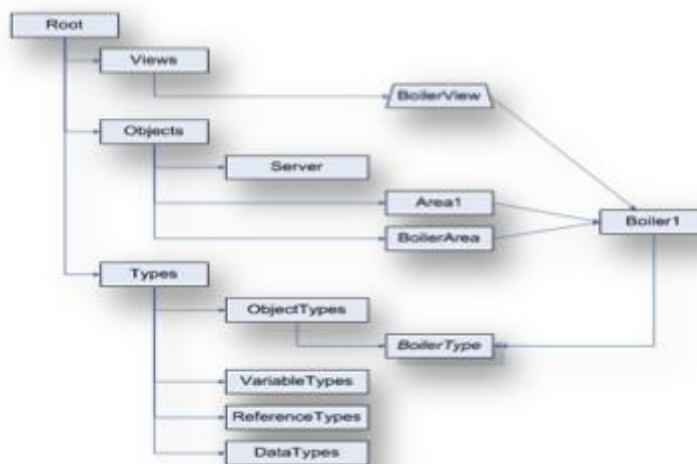
4.5.4 Data Types

OPC UA supports:

- Standard "scalar" data types
- Complex data types possible
- Definition of object types and instances
- Definition of custom specific data types

T
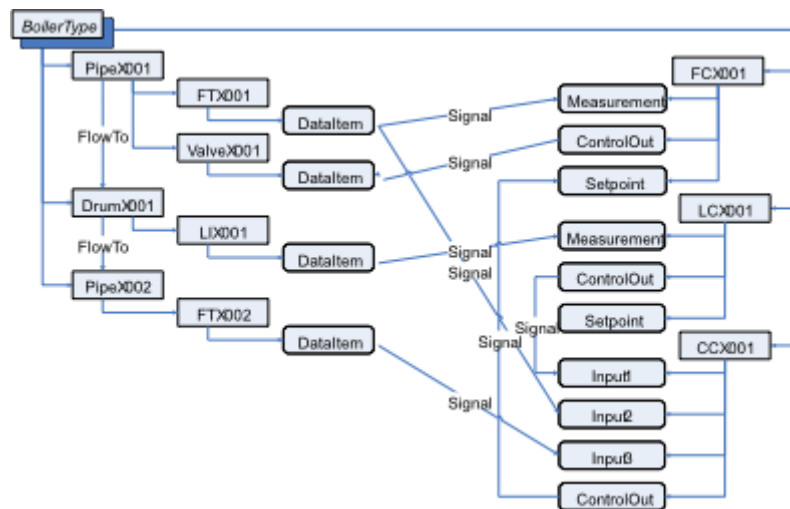
4.6 Mappings

The first 5 parts of the OPC UA specification are written to be independent of the technology used for implementation. This approach means OPC UA is a flexible specification that will continue to be applicable as technology evolves. On the other hand, this approach means that it is not possible to build an OPC UA *Application* with the information contained in **Error! Reference source not found.** through to **Error! Reference source not found.** because important implementation details have been left out.

This standard defines *Mappings* between the abstract specifications and technologies that can be used to implement them. The *Mappings* are organized into three groups: *DataEncodings, SecurityProtocols* and *TransportProtocols*. Different *Mappings* are combined together to create *StackProfiles*. All OPC UA *Applications* shall implement at least one *StackProfile* and can only communicate with other OPC UA *Applications* that implement the same *StackProfile.*

This standard defines the *DataEncodings*, the *SecurityProtocols* and the *TransportProtocols*. The *StackProfiles* are defined in **Error! Reference source not found.**.

All communication between OPC UA *Applications* is based on the exchange of *Messages*. The parameters contained in the *Messages* are defined in **Error! Reference source not found.**; however, their format is specified by the *DataEncoding* and *TransportProtocol*. For this reason, each *Message* defined in **Error! Reference source not found.** shall have a normative description which specifies exactly what shall be put on the wire. The normative descriptions are defined in the appendices.

A *Stack* is a collection of software libraries that implement one or more *StackProfiles*. The interface between an OPC UA *Application* and the *Stack* is a non-normative API which hides the details of the *Stack* implementation. An API depends on a specific *DevelopmentPlatform*. Note that the datatypes exposed in the API for a *DevelopmentPlatform* may not match the datatypes defined by the specification because of limitations of the *DevelopmentPlatform.* For example, Java does not support unsigned integers which means that any Java API will need to map unsigned integers onto a signed integer type.

Figure 21 – The OPC UA Stack Overview illustrates the relationships between the different concepts defined in this standard.

T
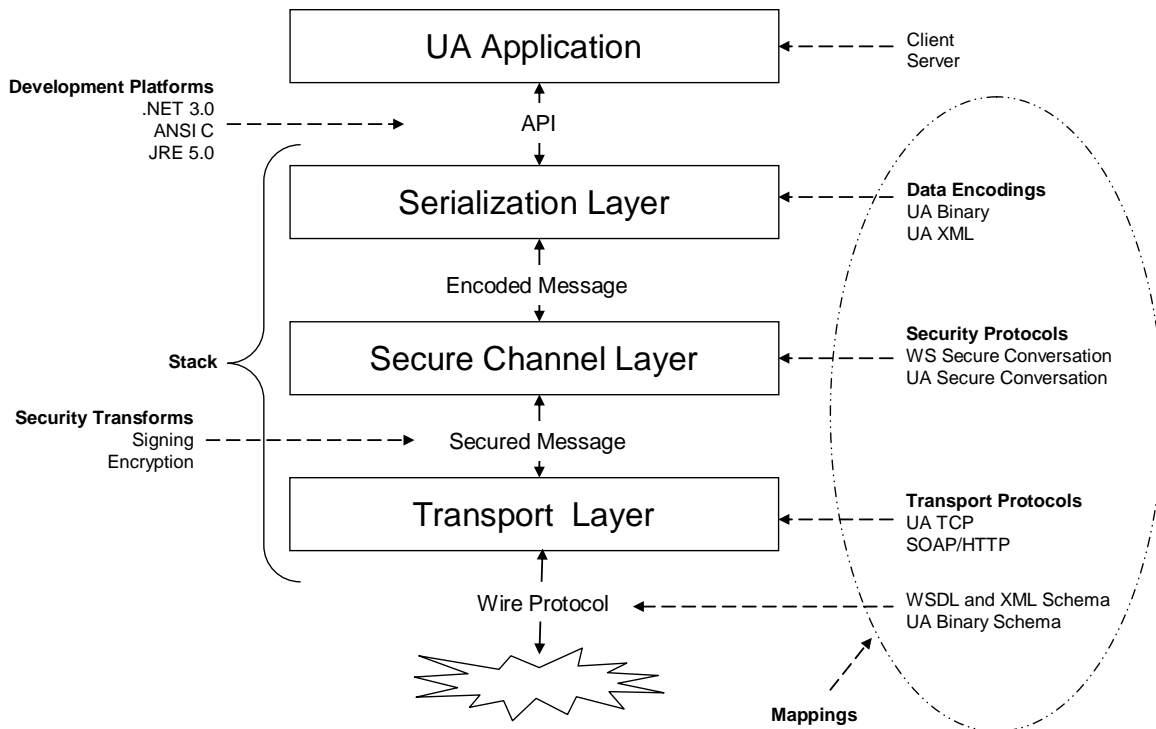
Figure 21 – The OPC UA Stack Overview

The layers described in this specification do not correspond to layers in the OSI 7 layer model [**Error! R eference source not found.**]. Each OPC UA *StackProfile* should be treated as a single Layer 7 (Application) protocol that is built on an existing Layer 5, 6 or 7 protocol such as TCP/IP, TLS or HTTP.The *SecureChannel* layer is always present even if the *SecurityMode* is *None*. In this situation, no security is applied but the *SecurityProtocol* implementation shall maintain a logical channel with a unique identifier. Users and administrators are expected to understand that a *SecureChannel* with *SecurityMode* set to *None* cannot be trusted unless the *Application* is operating on a physically secure network or a low level protocol such as IPSec is being used.

4.7 Profiles

The OPC Unified Architecture multipart specification describes a number of *Services* and a variety of information models. These *Services* and information models can be referred to as features of a *Server* or *Client*. *Servers* and *Clients* need to be able to describe which features they support. This part provides a grouping of these features. The individual features are grouped into *ConformanceUnits* which are further grouped into *Profiles*. Figure 22 - Profile - ConformanceUnit - TestCases provides an overview of the interactions between *Profiles*, *ConformanceUnits* and *TestCases*. The large arrows indicate the components that are used to construct the parent. For example a *Profile* is constructed from *Profiles* and *ConformanceUnits*. The figure also illustrates a feature of the OPC UA compliance test tool, in that it will test if a requested *Profile* passes all *ConformanceUnits*. It will also test all other *ConformanceUnits* and report any other *Profiles* that pass conformance testing. The individual *TestCases* are defined in separate. The *TestCases* are related back to the appropriate *ConformanceUnits* defined in this specification. This relationship is also displayed by the OPC UA Compliance Test Tool.
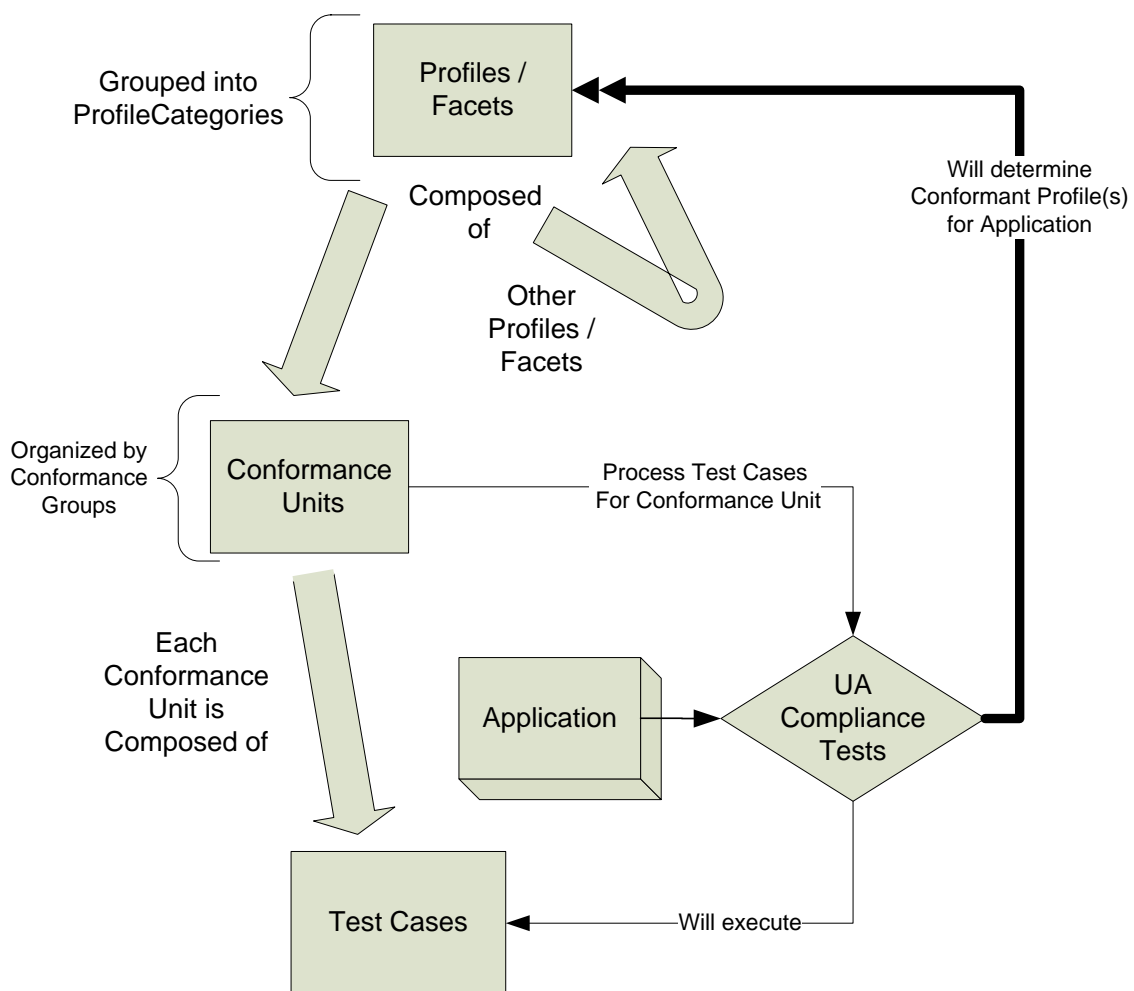

Figure 22 - Profile - ConformanceUnit - TestCases

T

4.7.1 ConformanceUnit

Each *ConformanceUnit* represents a specific set of features (e.g. a group of services, portions of services or information models) that can be tested as a single entity. *ConformanceUnits* are the building blocks of a *Profile*. Each *ConformanceUnit* can also be used as a test category. For each *ConformanceUnit,* there would be a number of TestCases that test the functionality described by the *ConformanceUnit.* The description of a *ConformanceUnit* is intended to provide enough information to illustrate the required functionality, but in many cases to obtain a complete understanding of the *ConformanceUnit* the reader may be required to also examine the appropriate part of the OPC UA specification. Additional Information regarding testing of a *ConformanceUnit* are provided in the test specifications.

The same features do not appear in more than one *ConformanceUnit*.

4.7.2 Profiles

Profiles are named groupings of *ConformanceUnits*. The *Servers* and *Clients* in an OPC UA application will provide the names of *Profiles* that they support. The definition of *Profiles* is a dynamic activity, in that it is expected that new *Profiles* will be added in the future. A *Profile* can be defined to inherit from an existing *Profile*. The new *Profile* may add additional *ConformanceUnits.* These additional *ConformanceUnits* may add additional features that are to be tested. The additional *ConformanceUnits* may also further restrict inherited *ConformanceUnits*.

An OPC UA Application will typically support multiple *Profiles*.

Multiple *Profiles* may include the same *ConformanceUnit*.

Testing of a *Profile* consists of testing the individual *ConformanceUnits* that comprise the *Profile*.

Profiles are named based on naming conventions

4.7.3 Profile Categories

Profiles are grouped into categories to help vendors and end users understand the applicability of a *Profile*. Categories include: *Server* and *Client*, but other example could also include Power Generation or Chemical Plant. A *Profile* shall be assigned to one or more categories.

Table 2 – Profile Categories contains the list of currently defined *ProfileCategories*.

Table 2 – Profile Categories

Category	Description
Client	*Profiles* of this category specify a complete functional set for an OPC UA *Client*. The URI of such profiles has to be part of a Software *Certificate* passed in the CreateSession request.
Discovery Server	*Profiles* of this category are for *Discovery Servers*
External	*Profiles* that are defined outside
Security	*Profiles* of this category specify a security policy. The URI of such profiles has to be part of an Endpoint Description returned from the GetEndpoint service.
Server	*Profiles* of this category specify a complete functional set for an OPC UA *Server*. The URI of such profiles has to be part of a Software *Certificate* returned with the CreateSession service response.
Transport	*Profiles* of this category specify a specific protocol mapping. The URI of such profiles has to be part of an Endpoint Description.

T

4.8 Data Access

Data Access deals with the representation and use of automation data in OPC UA Servers.

Automation data can be located inside the OPC UA Server or on I/O cards directly connected to the OPC UA Server. It can also be located in sub-servers or on other devices such as controllers and input/output modules, connected by serial links via field buses or other communication links. OPC UA Data Access Servers provide one or more OPC UA Data Access Clients with transparent access to their automation data.

The links to automation data instances are called *DataItems*. Which categories of automation data are provided is completely vendor-specific. Figure 23 – OPC *DataItems* are linked to automation data illustrates how the *AddressSpace* of an OPC UA server might consist of a broad range of different *DataItems*.
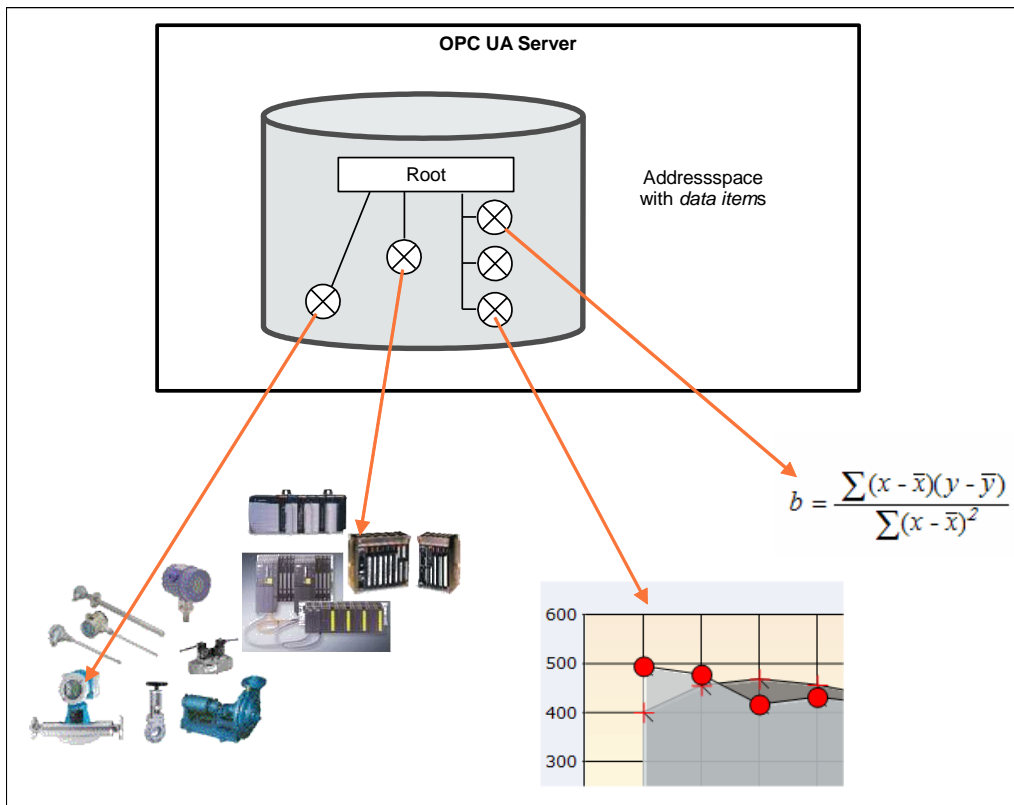
Figure 23 – OPC *DataItems* are linked to automation data

Clients may read or write *DataItems*, or monitor them for value changes. The services needed for these operations are specified in **Error! Reference source not found.**. Changes are defined as a change in status (quality) or a change in value that exceeds a client-defined range called a *Deadband*. To detect the value change, the difference between the current value and the last reported value is compared to the *Deadband*.

T

4.8.1 Model

The DataAccess model extends the variable model by defining *VariableTypes*. The *DataItemType* is the base type. *ArrayItemType*, *AnalogItemType* and *DiscreteItemType* (and its *TwoState* and *MultiState* subtypes) are specializations. See Figure 24 – *DataItem VariableType* Hierarchy. Each of these *VariableTypes* can be further extended to form domain or server specific *DataItems*.

Figure 24 – *DataItem VariableType* Hierarchy

T
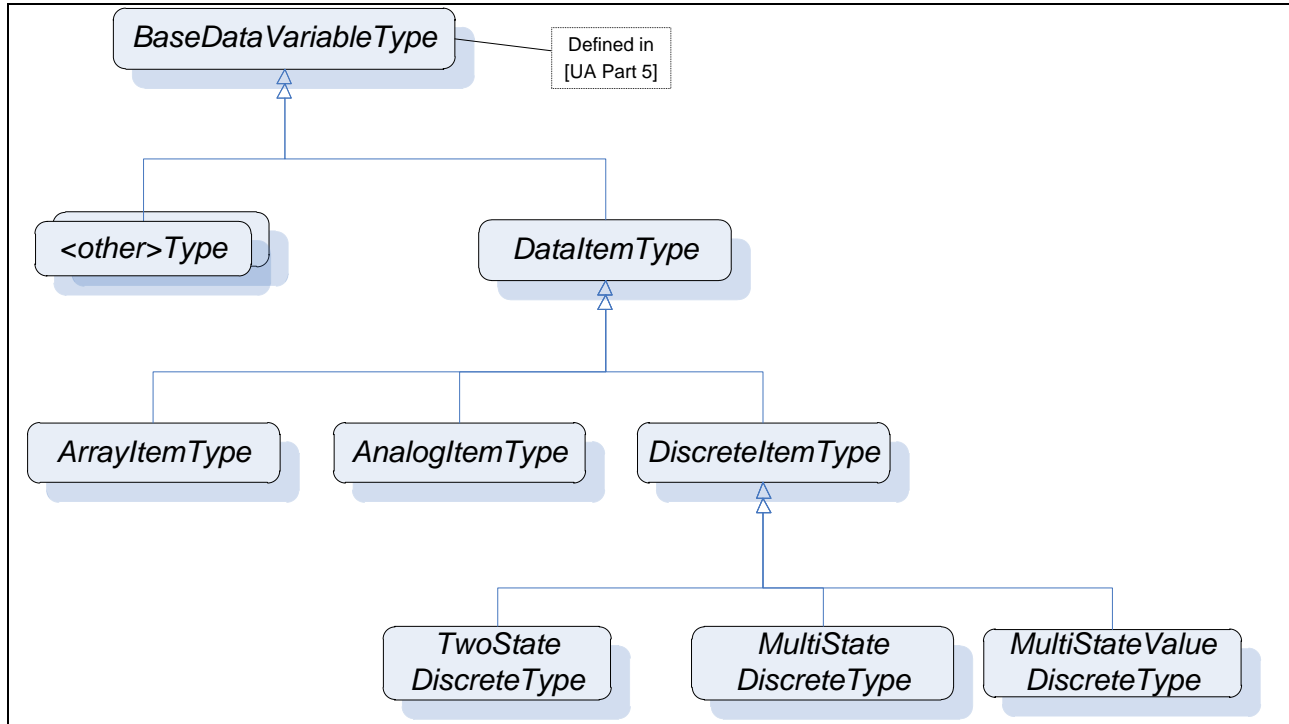
4.9 Alarms and Conditions

This standard defines an *Information Model* for *Conditions,* Dialog *Conditions,* and *Alarms* including acknowledgement capabilities. It is built upon and extends base Event handling which is defined in **Error! R eference source not found.**, **Error! Reference source not found.** and **Error! Reference source not found.**. This *Information Model* can also be extended to support the additional needs of specific domains. The details of what aspects of the Information Model are supported are defined via Profiles (see **Error! Reference source not found.** for Profile definitions). Some systems may expose historical Events and Conditions via the standard Historical Access framework (see **Error! Reference source not found.** for Historical Event d efinitions).

4.9.1 Conditions

Conditions are used to represent the state of a system or one of its components. Some common examples are:

- a temperature exceeding a configured limit
- a device needing maintenance
- a batch process that requires a user to confirm some step in the process before proceeding

Each *Condition* instance is of a specific *ConditionType.* The *ConditionType* and derived types are sub-types of the *BaseEventType* (see **Error! Reference source not found.** and **Error! Reference source not found.**). Th is part defines types that are common across many industries. It is expected that vendors or other standardisation groups will define additional *ConditionTypes* deriving from the common base types defined in this part. The *ConditionTypes* supported by a *Server* are exposed in the *AddressSpace* of the *Server.*

Condition instances are specific implementations of a *ConditionType.* It is up to the *Server* whether such instances are also exposed in the *Server's AddressSpace. Condition* instances shall have a unique identifier to differentiate them from other instances. This is independent of whether they are exposed in the *AddressSpace.*

As mentioned above, *Conditions* represent the state of a system or one of its components. In certain cases, however, previous states that still need attention also have to be maintained. *ConditionBranches* are introduced to deal with this requirement and distinguish current state and previous states. Each *ConditionBranch* has a BranchId that differentiates it from other branches of the same *Condition* instance. The *ConditionBranch* which represents the current state of the *Condition* (the trunk) has a Null BranchId. *Server*s can generate separate *Event Notifications* for each branch. When the state represented by a *ConditionBranch* does not need further attention, a final *Event Notification* for this branch will have the *Retain Property* set to False. Maintaining previous states and therefore also the support of multiple branches is optional for *Servers.*

Conceptually, the lifetime of the *Condition* instance is independent of its state. However, *Servers* may provide access to *Condition* instances only while *ConditionBranches* exist.

The base *Condition* state model is illustrated in Figure 25 – Base Condition State Model. It is extended by the various *Condition* subtypes defined in this standard and may be further extended by vendors or other standardisation groups. The primary states of a *Condition* are disabled and enabled. The *Disabled* state is intended to allow *Conditions* to be turned off at the *Server* or below the *Server* (in a device or some underlying system). The *Enabled* state is normally extended with the addition of sub-states.

This standard defines an *Information Model* for *Conditions*, Dialog *Conditions*, and *Alarms* including acknowledgement capabilities. It is built upon and extends base Event handling which is defined in **Error! R eference source not found.**, **Error! Reference source not found.** and **Error! Reference source not found.**. This *Information Model* can also be extended to support the additional needs of specific domains. The details of what aspects of the Information Model are support are provided via Profiles (see **Error! Reference source not found.**). It is expected that systems will provide historical Events and Conditions via the standard Historical Access framework (see **Error! Reference source not found.**).
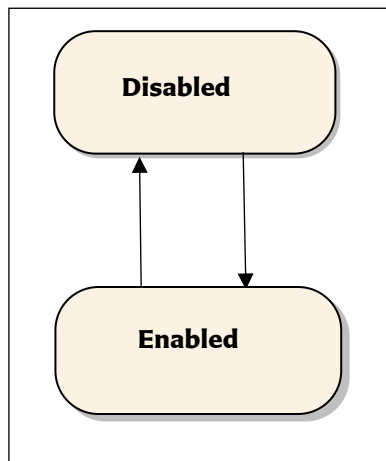
Figure 25 – Base Condition State Model

A transition into the *Disable*d state results in a *Condition Event* however no subsequent *Event Notifications* are generated until the *Condition* returns to the Enabled state.

When a *Condition* enters the Enabled state, that transition and all subsequent transitions result in *Condition Events* being generated by the *Server*.

If *Auditing* is supported by a *Server,* the following *Auditing* related action shall be performed. The *Server* will generate *AuditEvents* for *Enable* and *Disable* operations (either through a *Method* call or some *Server* / vendor – specific means), rather than generating an *AuditEvent Notification* for each *Condition* instance being enabled or disabled. *AuditEvents* are also generated for any other *Operator* action that results in changes to the *Conditions*.

4.9.2 Acknowledgeable Conditions

AcknowledgeableConditions are sub-types of the base *ConditionType*. *AcknowledgeableConditions* expose states to indicate whether a *Condition* has to be acknowledged or confirmed.

An *AckedState* and a *ConfirmedState* extend the *EnabledState* defined by the *Condition*. The state model is illustrated in Figure 26 – AcknowledgeableConditions State Model. The enabled state is extended by adding the *AckedState* and (optionally) the *ConfirmedState*.
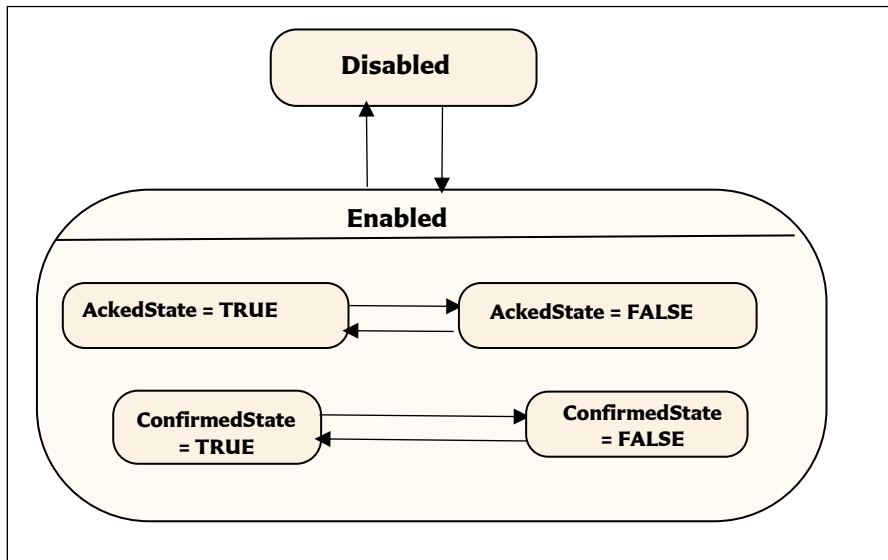
Figure 26 – AcknowledgeableConditions State Model

Acknowledgment of the transition may come from the *Client* or may be due to some logic internal to the *Server*. For example, acknowledgment of a related *Condition* may result in this *Condition* becoming acknowledged, or the *Condition* may be set up to automatically acknowledge itself when the acknowledgeable situation disappears.

Two *Acknowledge* state models are supported by this standard. Either of these state models can be extended to support more complex acknowledgement situations.

The basic *Acknowledge* state model is illustrated in Figure 27 – Acknowledge State Model. This model defines an AckedState. The specific state changes that result in a change to the state depend on a Server's implementation. For example, in typical *Alarm* models the change is limited to a transition to the *Active* state or transitions within the *Active* state. More complex models however can also allow for changes to the AckedState when the *Condition* transitions to an inactive state.

T
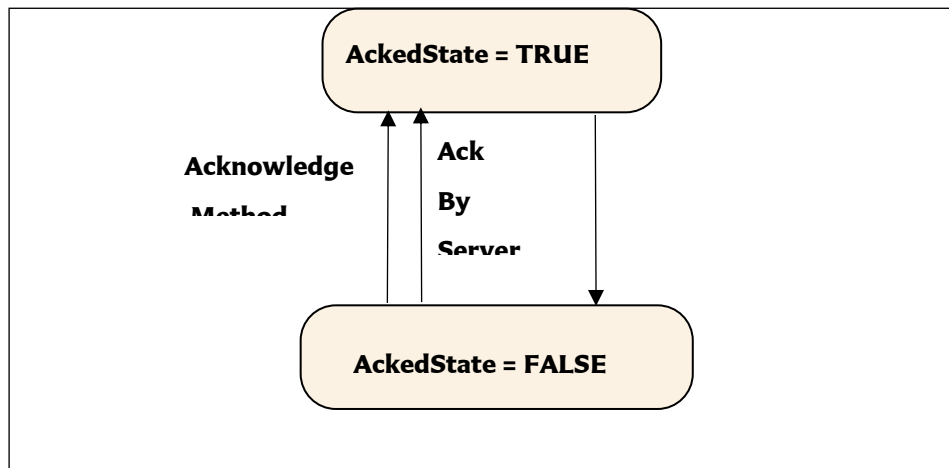
Figure 27 – Acknowledge State Model

A more complex state model which adds a confirmation to the basic *Acknowledge* is illustrated in Figure 28 – Confirmed Acknowledge State Model. The *Confirmed Acknowledge* model is typically used to differentiate between acknowledging the presence of a *Condition* and having done something to address the *Condition*. For example an *Operator* receiving a motor high temperature *Notification* calls the *Acknowledge Method* to inform the *Server* that the high temperature has been observed. The *Operator* then takes some action such as lowering the load on the motor in order to reduce the temperature. The *Operator* then calls the *Confirm Method* to inform the *Server* that a corrective action has been taken.

``
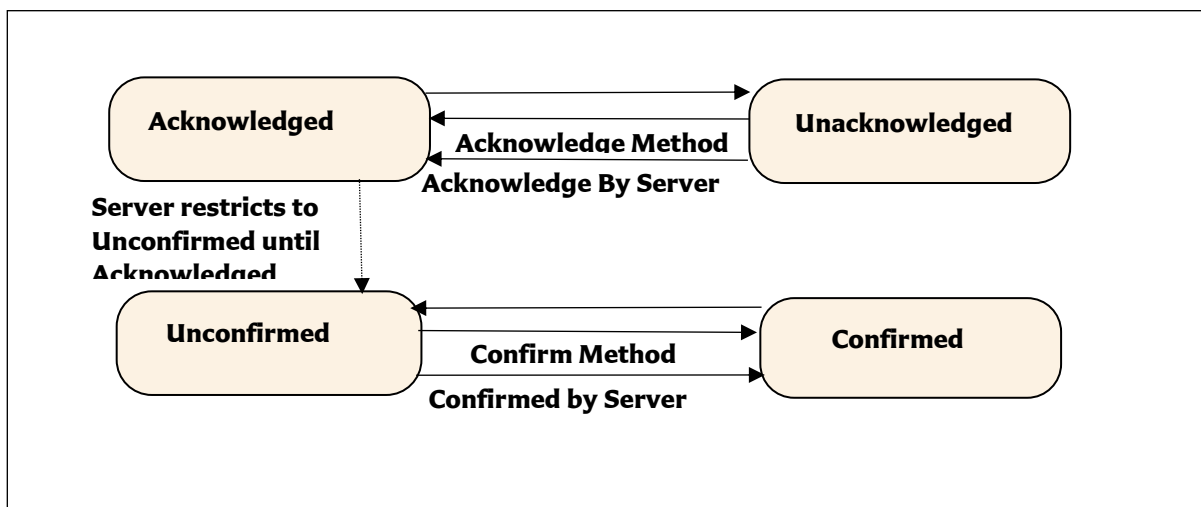
Figure 28 – Confirmed Acknowledge State Model

4.9.3 Previous States of Conditions

Some systems require that previous states of a *Condition* are preserved for some time. A common use case is the acknowledgement process. In certain environments it is required to acknowledge both the transition into *Active* state and the transition into an inactive state. Systems with strict safety rules sometimes require that every transition into *Active* state has to be acknowledged. In situations where state changes occur in short succession there can be multiple unacknowledged states and the *Server* has to maintain *ConditionBranches* for all previous unacknowledged states. These branches will be deleted after they have been acknowledged or if they reached their final state.

Multiple ConditionBranches can also be used for other use cases where snapshots of previous states of a *Condition* require additional actions.

4.9.4 Condition State Synchronization

When a *Client* subscribes for *Events*, the *Notification* of transitions will begin at the time of the *Subscription*. The currently existing state will not be reported. This means for example that *Clients* are not informed of currently *Active Alarms* until a new state change occurs.

Clients can obtain the current state of all *Condition* instances that are in an interesting state, by requesting a *Refresh* for a *Subscription*. It should be noted that *Refresh* is not a general replay capability since the *Server* is not required to maintain an *Event* history.

Clients request a *Refresh* by calling the ConditionRefresh *Method*. The *Server* will respond with a *RefreshStartEvent*. This *Event* is followed by the *Retained Conditions*. The *Server* may also send new *Event Notifications* interspersed with the *Refresh* related *Event Notifications*. After the *Server* is done with the *Refresh,* a *RefreshEndEvent* is issued marking the completion of the Refresh. *Clients* shall check for multiple *Event Notifications* for a *ConditionBranch* to avoid overwriting a new state delivered together with an older state from the *Refresh* process. If a *ConditionBranch* exists, then the current *Condition* shall be reported. This is true even if the only interesting item regarding the *Condition* is that *ConditionBranches* exist. This allows a *Client* to accurately represent the current *Condition* state.

A *Client* that wishes to display the current status of *Alarms* and *Conditions* (known as a "current *Alarm* display") would use the following logic to process *Refresh Event Notifications*. The *Client* flags all *Retained Conditions* as suspect on reception of the *Event* of the *RefreshStartEvent*. The *Client* adds any new *Events* that are received during the Refresh without flagging them as suspect. The *Client* also removes the suspect flag from any *Retained Conditions* that are returned as part of the Refresh. When the *Client* receives a *RefreshEndEvent,* the *Client* removes any remaining suspect *Events,* since they no longer apply.

The following items should be noted with regard to ConditionRefresh:

- Some systems require that previous states of a *Condition* are preserved for some time. Some *Servers* – in particular if they require acknowledgement of previous states – will maintain separate *ConditionBranches* for prior states that still need attention.

  ConditionRefresh shall issue *Event Notifications* for all interesting states (current and previous) of a *Condition* instance and *Clients* can therefore receive more than one *Event* for a *Condition* instance with different BranchIds.

- Under some circumstances a *Server* may not be capable of ensuring the *Client* is fully in sync with the current state of *Condition* instances. For example if the underlying system represented by the *Server* is reset or communications are lost for some period of time the *Server* may need to resynchronize itself with the underlying system. In these cases the *Server* shall send an *Event* of the *RefreshRequiredEventType* to advise the *Client* that a *Refresh* may be necessary. A *Client* receiving this special *Event* should initiate a *ConditionRefresh* as noted in this clause.

- To ensure a *Client* is always informed, the three special *EventTypes* (*RefreshEndEventType, RefreshStartEventType* and *RefreshRequiredEventType*) ignore the *Event* content filtering associated with a *Subscription* and will always be delivered to the *Client*.

- *ConditionRefresh* applies to a *Subscription*. If multiple *Event Notifiers* are included in the same *Subscription*, all *Event Notifiers* are refreshed.

4.9.5 Severity, Quality, and Comment

Comment, Severity and Quality are important elements of *Conditions* and any change to them will cause *Event Notifications*.

The Severity of a *Condition* is inherited from the base *Event* model defined in [UA Part 5]. It indicates the urgency of the *Condition* and is also commonly called 'priority', especially in relation to *Alarms* in the *ProcessConditionClass*.

A Comment is a user generated string that is to be associated with a certain state of a *Condition*.

Quality refers to the quality of the data value(s) upon which this *Condition* is based. Since a *Condition* is usually based on one or more *Variables*, the *Condition* inherits the quality of these *Variables*. E.g., if the process value is "Uncertain", the "LevelAlarm" *Condition* is also questionable.

4.9.6 Dialogs

Dialogs are *ConditionTypes* used by a *Server* to request user input. They are typically used when a *Server* has entered some state that requires intervention by a *Client*. For example a *Server* monitoring a paper machine indicates that a roll of paper has been wound and is ready for inspection. The *Server* would activate a Dialog *Condition* indicating to the user that an inspection is required. Once the inspection has taken place the user responds by informing the *Server* of an accepted or unaccepted inspection allowing the process to continue.

4.9.7 Alarms

Alarms are specializations of *AcknowledgeableConditions* that add the concepts of an *Active* state, a *Shelving* state and a *Suppressed* state to a *Condition*. The state model is illustrated in Figure 29 – Alarm State Machine Model.
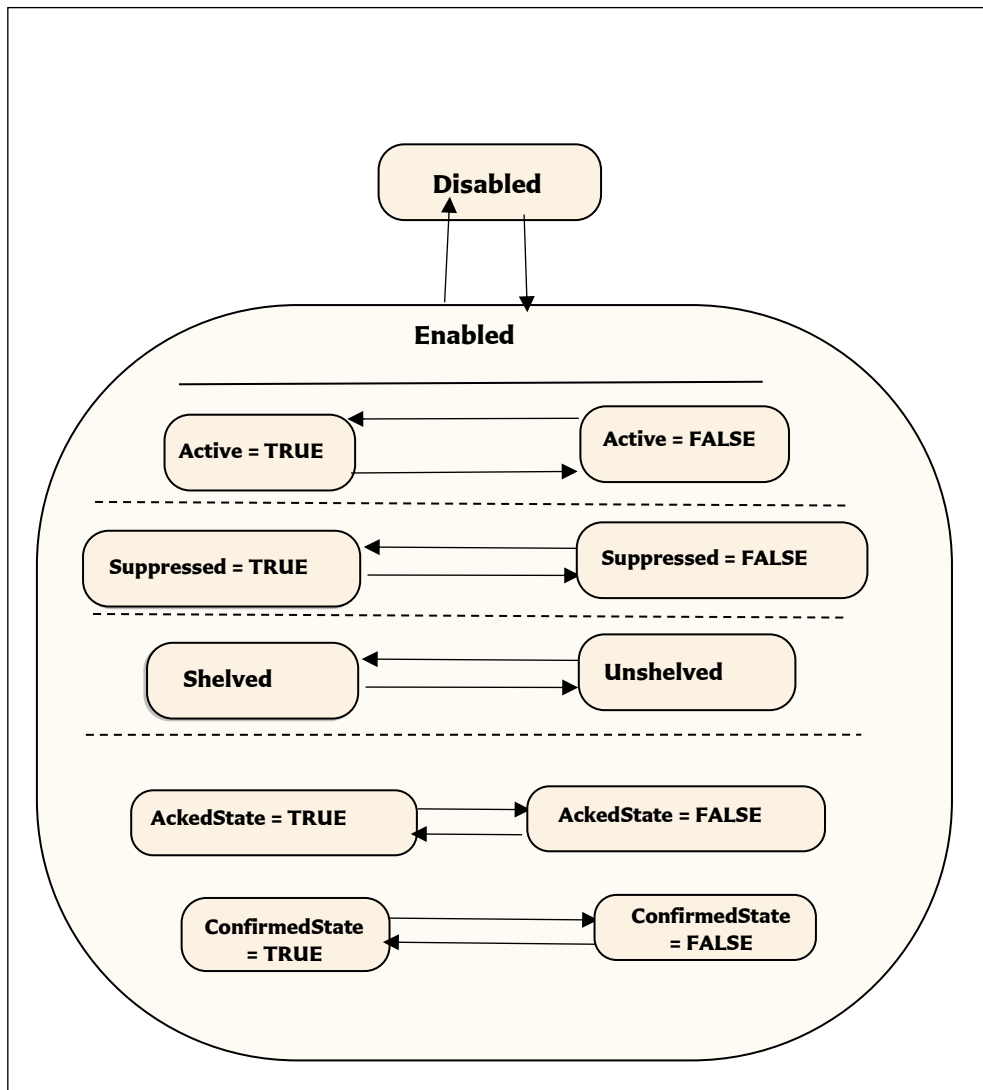
Figure 29 – Alarm State Machine Model

An *Alarm* in the *Active* state indicates that the situation the *Condition* is representing currently exists. When an *Alarm* is an inactive state it is representing a situation that has returned to a normal state.

Some *Alarm* subtypes introduce sub-states of the *Active* state. For example an *Alarm* representing a temperature may provide a high level state as well as a critically high state.

The *Shelving* state can be set by an *Operator* via OPC UA *Methods*. The *Suppressed* state is set internally by the *Server* due to system specific reasons. *Alarm* systems typically implement the *Suppress* and Shelve features to help keep *Operators* from being overwhelmed during *Alarm* "storms" by limiting the number of *Alarms* an *Operator* sees on a current *Alarm* display. This is accomplished by setting the *SuppressedOrShelved* flag on second order dependent *Alarms* and/or *Alarms* of less severity, leading the *Operator* to concentrate on the most critical issues.

The *Shelved* and *Suppressed* states differ from the *Disable*d state in that *Alarms* are still fully functional and can be included in *Subscription Notifications* to a *Client*.

4.9.8 Multiple Active States

In some cases it is desirable to further define the *Active* state of an *Alarm* by providing a sub-state machine for the *Active* State. For example a multi-state level *Alarm* when in the *Active* state may be in one of the following sub-states: LowLow, Low, High or HighHigh. The state model is illustrated in Figure 30.
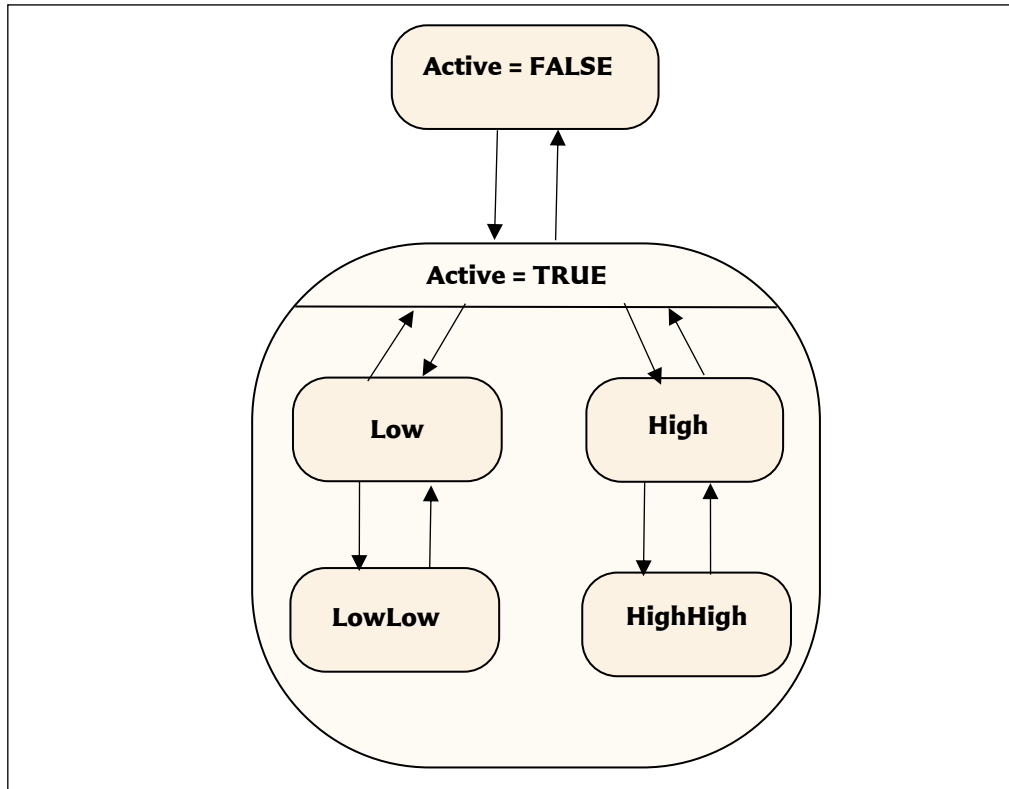
Figure 30 – Multiple Active States Example

With the multi-state *Alarm* model, state transitions among the sub-states of *Active* are allowed without causing a transition out of the *Active* state.

To accommodate different use cases both a (mutually) exclusive and a non-exclusive model are supported.

Exclusive means that the *Alarm* can only be in one sub-state at a time. If for example a temperature exceeds the HighHigh limit the associated exclusive LevelAlarm will be in the HighHigh sub-state and not in the High sub-state.

Some *Alarm* systems, however, allow multiple sub-states to exist in parallel. This is called non-exclusive. In the previous example where the temperature exceeds the HighHigh limit a non-exclusive LevelAlarm will be both in the High and the HighHigh sub-state.

4.9.9 Condition Instances in the Address Space

Because *Conditions* always have a state (*Enabled* or *Disabled*) and possibly many sub-states it makes sense to have instances of *Conditions* present in the *AddressSpace*. If the *Server* exposes *Condition* instances they usually will appear in the *AddressSpace* as components of the *Objects* that "own" them. For example a temperature transmitter that has a built-in high temperature *Alarm* would appear in the *AddressSpace* as an instance of some temperature transmitter *Object* with a *HasComponent Reference* to an instance of a *LevelAlarmType*.

The availability of instances allows Data Access *Clients* to monitor the current *Condition* state by subscribing to the *Attribute* values of *Variable Nodes*.

While exposing *Condition* instances in the *AddressSpace* is not always possible, doing so allows for direct interaction (read, write and *Method* invocation) with a specific *Condition* instance. For example, if a *Condition* instance is not exposed, there is no way to invoke the *Enable* or *Disable Method* for the specific *Condition* instance.

4.9.10 Alarm and Condition Auditing

The OPC UA Standards include provisions for auditing. Auditing is an important security and tracking concept. Audit records provide the "Who", "When" and "What" information regarding user interactions with a system. These audit records are especially important when *Alarm* management is considered. *Alarms* are the typical instrument for providing information to a user that something needs the user's attention. A record of how the user reacts to this information is required in many cases. Audit records are generated for all *Method* calls that affect the state of the system, for example an *Acknowledge Method* call would generate an *AuditConditionAck Event*.

The standard *AuditEventTypes* defined in **Error! Reference source not found.** already includes the fields r equired for *Condition* related audit records. To allow for filtering and grouping, this standard defines a number of sub-types of the *AuditEventTypes* but without adding new fields to them.

This standard describes the *AuditEventType* that each *Method* is required to generate. For example, the *Disable Method* has an *AlwaysGeneratesEvent Reference* to an *AuditConditionEnableEventType*. An *Event* of this type shall be generated for every invocation of the *Method*. The audit *Event* describes the user interaction with the system, in some cases this interaction may affect more than one *Condition* or be related to more than one state.

4.9.11 Model

The *Alarm* and *Condition* model extends the OPC UA base *Event* model by defining various *Event Types* based on the *BaseEventType*. All of the *Event Types* defined in this standard can be further extended to form domain or *Server* specific *Alarm* and *Condition Types*.

Instances of *Alarm* and *Condition Types* may be optionally exposed in the *AddressSpace* in order to allow direct access to the state of an *Alarm* or *Condition*.
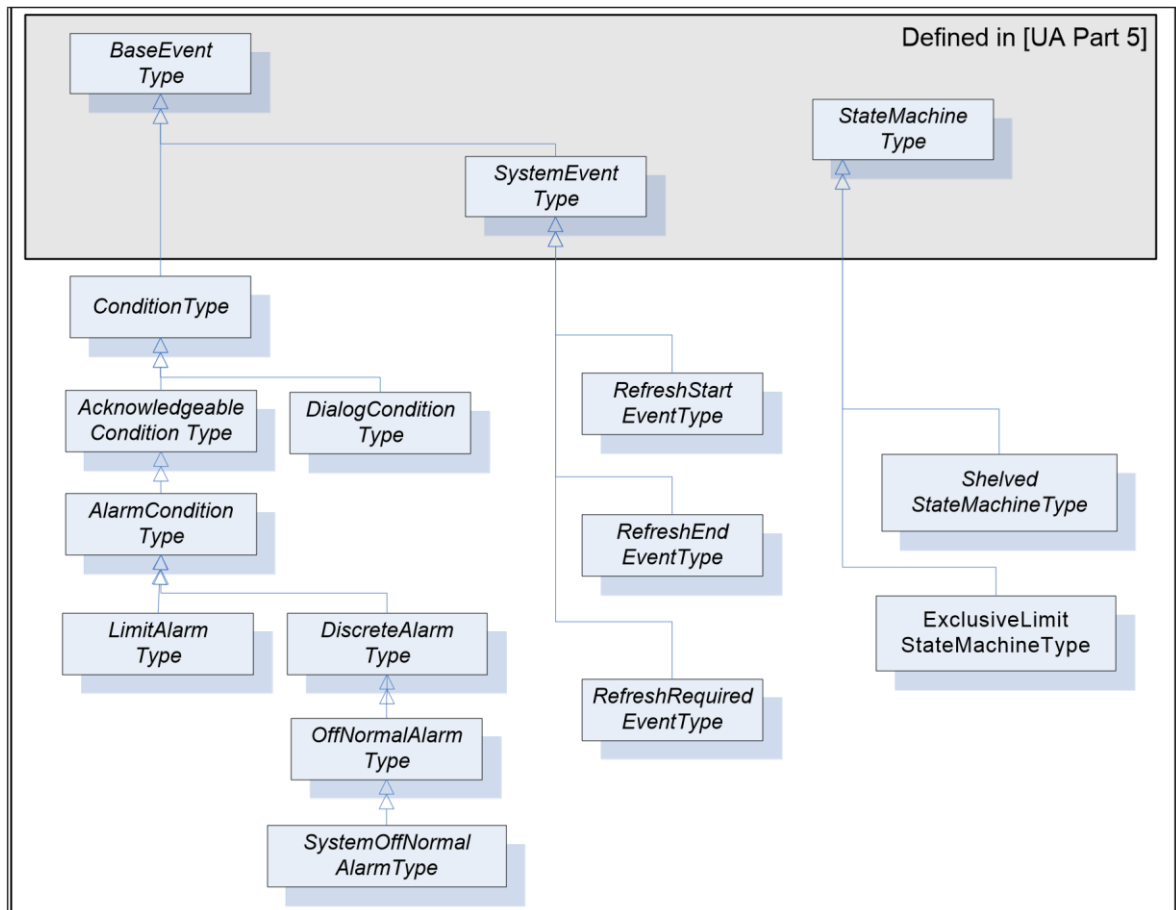
Figure 31 – ConditionType hierarchy

4.9.11.1   Condition Model

The *Condition* model extends the *Event* model by defining the *ConditionType*. The *ConditionType* introduces the concept of states differentiating it from the base *Event* model. Unlike the *BaseEventType, Conditions* are not transient. The *ConditionType* is further extended into Dialog and *AcknowledgeableConditionTypes,* each of which have their own subtypes.

The *Condition* model is illustrated in Figure 32 – Condition model and formally defined in the subsequent tables. It is worth noting that this figure, like all figures in this document, is not intended to be complete. Rather, the figures only illustrate information provided by the formal definitions.
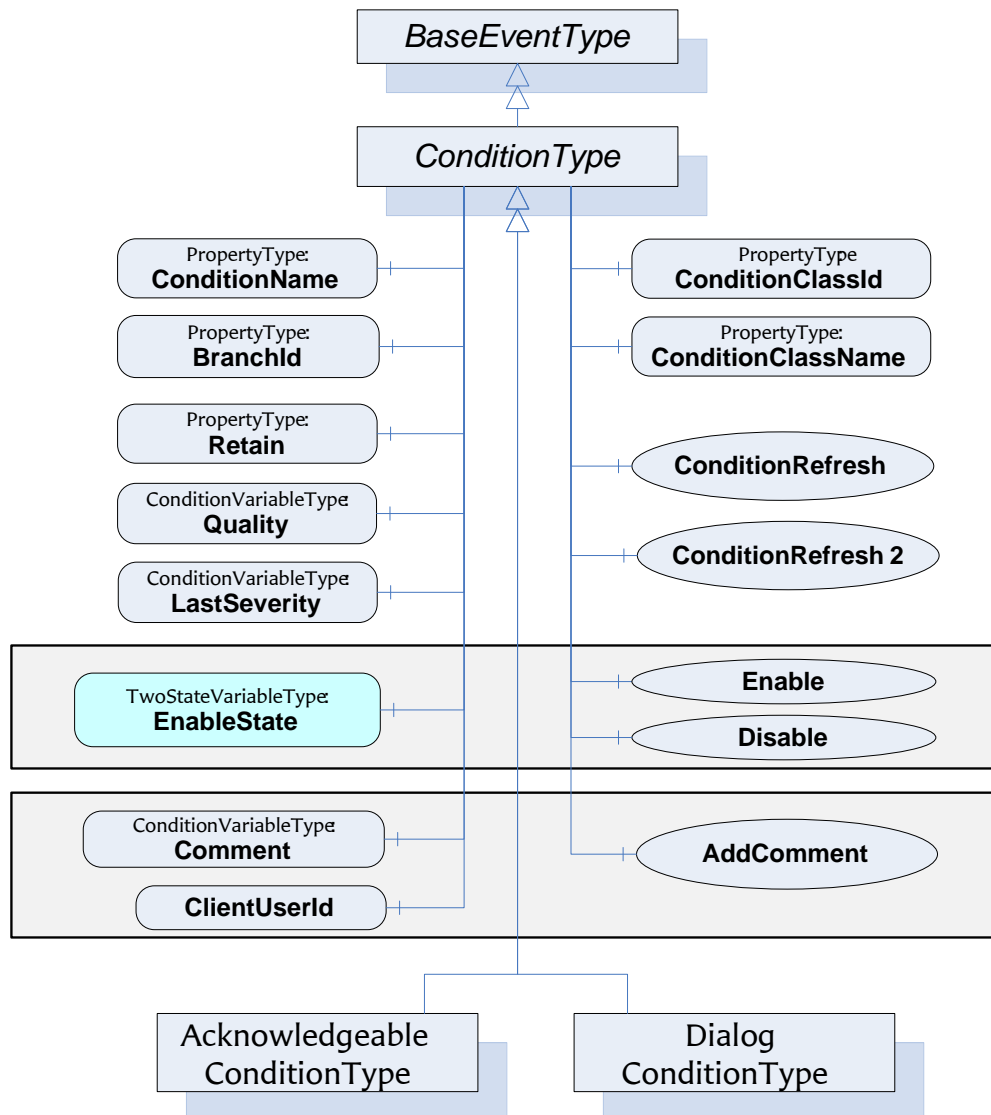
Figure 32 – Condition model

4.9.11.2 Dialog Model

The Dialog Model is an extension of the *Condition* model used by a *Server* to request user input. It provides functionality similar to the standard *Message* dialogs found in most operating systems. The model can easily be customized by providing *Server* specific response options in the *ResponseOptionSet* and by adding additional functionality to derived *Condition Types*.

The *DialogConditionType* is used to represent *Conditions* as dialogs. It is illustrated in Figure 33 - DialogConditionType Overview
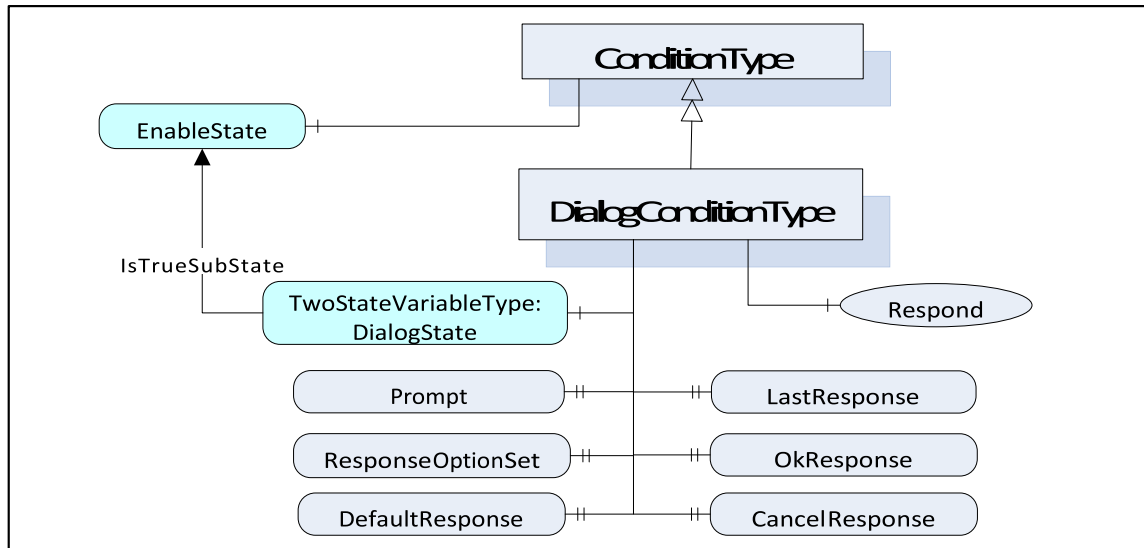
Figure 33 - DialogConditionType Overview

4.9.11.3 Acknowledgeable Condition Model

The Acknowledgeable *Condition* Model extends the *Condition* model. States for acknowledgement and confirmation are added to the *Condition* model.

AcknowledgeableConditions are represented by the *AcknowledgeableConditionType* which is a subtype of the *ConditionType*. The model is formally defined in the following sub clauses.

The *AcknowledgeableConditionType* extends the *ConditionType* by defining acknowledgement characteristics. It is an abstract type. The *AcknowledgeableConditionType* is illustrated in Figure 34 - AcknowledgeableConditionType Overview.
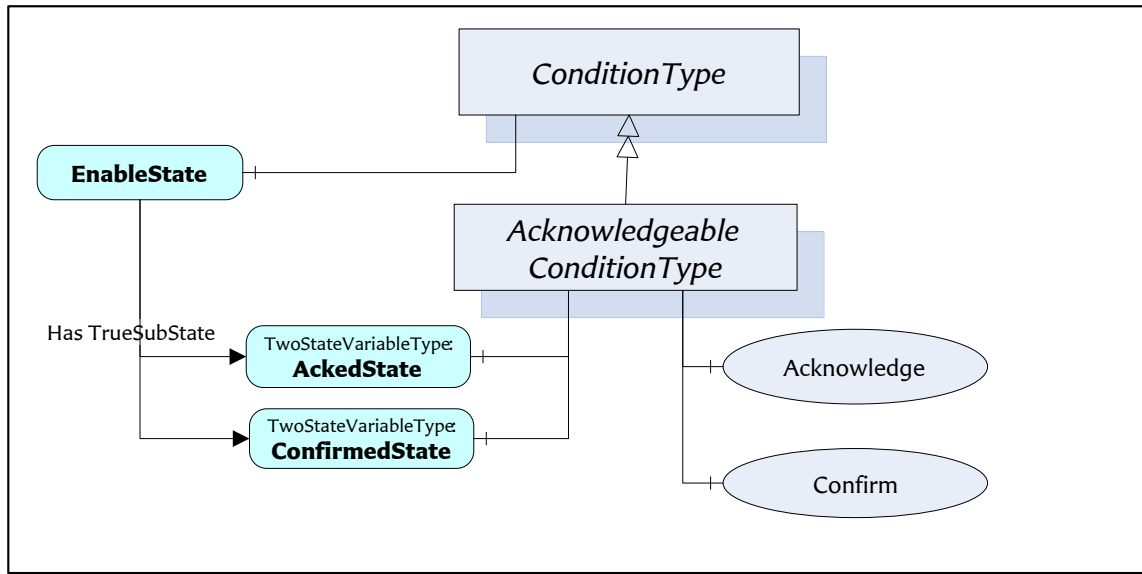
Figure 34 - AcknowledgeableConditionType Overview

4.9.11.4 Alarm model

Figure 35 - AlarmConditionType hierarchy model informally describes the AlarmConditionType, its sub-types and where it is in the hierarchy of *Event Types*.
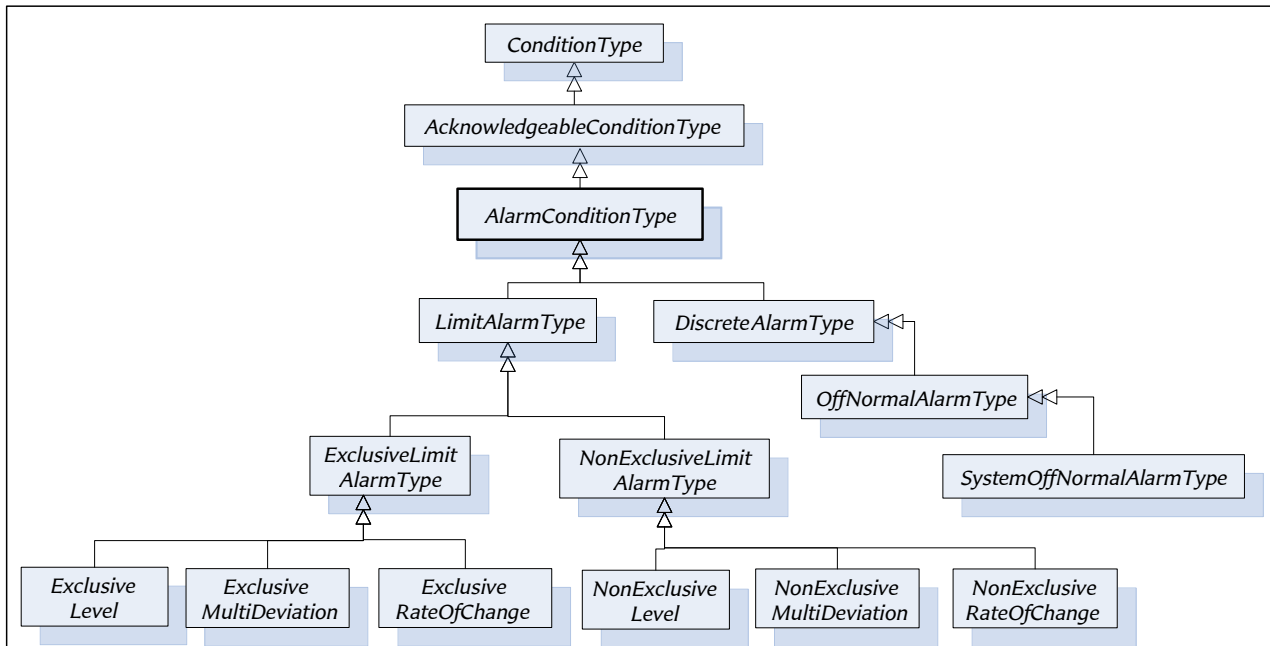
Figure 35 - AlarmConditionType hierarchy model

The *AlarmConditionType* is an abstract type that extends the *AcknowledgeableConditionType* by introducing an *ActiveState*, *SuppressedState* and *ShelvingState*. The *Alarm* model is illustrated in Figure 36 - AlarmConditionType definition. This illustration is not intended to be a complete definition.
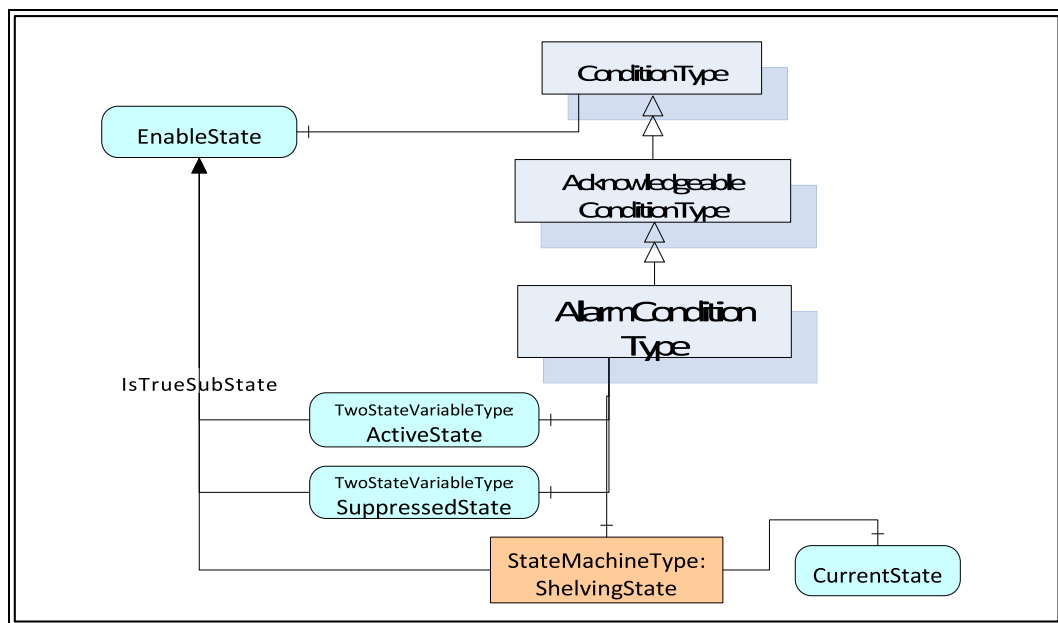
Figure 36 - AlarmConditionType definition

4.10 Programs

Integrated automation facilities manage their operations through the exchange of data and coordinated invocation of system functions like illustrated in Figure 37 – Automation facility control. *Services* are required to perform the data exchanges and to invoke the functions that constitute system operation. These functions may be invoked through human machine interfaces, cell controllers, or other supervisory control and data acquisition type systems. OPC UA defines *Methods* and *Programs* as an interoperable way to advertise, discover, and request these functions. They provide a normalizing mechanism for the semantic description, invocation of, and result reporting of these functions. Together *Methods* and *Programs* complement the other OPC UA *Services* and *ObjectTypes* to facilitate the operation of an automation environment using a client server hierarchy.
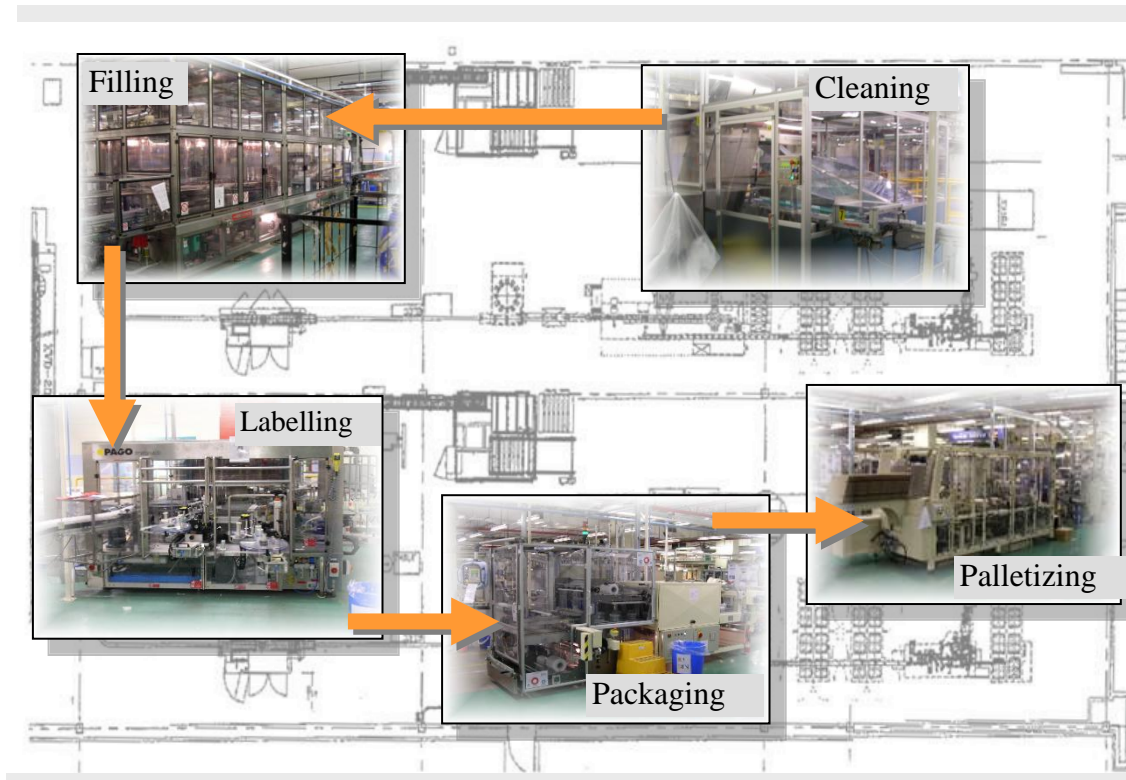
Figure 37 – Automation facility control

Methods and *Programs* model functions typically having different scopes, behaviours, lifetimes, and complexities in *OPC Servers* and the underlying systems. These functions are **not** normally characterized by the reading or writing of data which is accomplished with the OPC UA A*ttribute* service set.

Methods represent basic functions in the server that can be invoked by a client. *Programs* by contrast, model more complex and stateful functionality in the system. For example, a method call may be used to perform a calculation or reset a counter. A *Program* is used to run and control a batch process, execute a machine tool part program, or manage a domain download. *Methods* and their invocation mechanism are described in **Error! Reference source not found.** and**Error! Reference source not found.**.

4.11 Historical Access

This standard defines the handling of historical time series data and historical *Event* data in the OPC Unified Architecture. Included is the specification of the representation of historical data and *Events* in the *AddressSpace.*

A *Server* supporting Historical Access provides *Clients* with transparent access to different historical data and/or historical *Event* sources (e.g. process historians, event historians, etc.).

The historical data or *Events* may be located in a proprietary data collection, database or a short term buffer within the memory. A *Server* supporting Historical Access will provide historical data and *Events* for all or a subset of the available *Variables*, *Objects, Properties* or *Views* within the *Server AddressSpace.*

Figure 38 – Possible OPC UA Server supporting Historical Access illustrates how the *AddressSpace* of a UA *Server* might consist of a broad range of different historical data and/or historical *Event* sources.
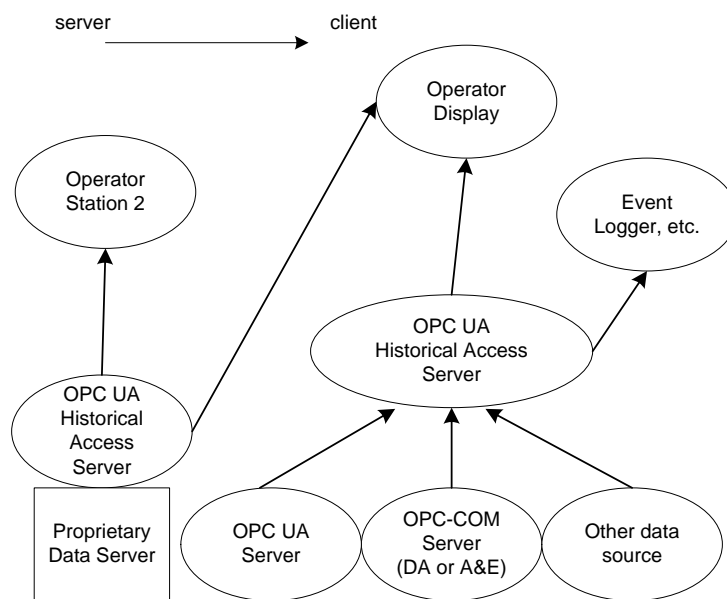
Figure 38 – Possible OPC UA Server supporting Historical Access

The *Server* may be implemented as a standalone OPC UA *Server* that collects data from another OPC UA *Server* or another data source. The *Client* that references the **Error! Unknown document property name.** for historical data may be simple trending packages that just desire values over a given time frame or they may be complex reports that require data in multiple formats.

4.12 Discovery

The discovery process allows the *Clients* to first find *Servers* on the network and then discover how to connect to them. Once a *Client* has this information it can save it and use it to connect directly to the *Server* again without going through the discovery process. If the *Client* finds that it cannot connect then that could mean the *Server* configuration has changed and the *Client* needs to go through the discovery process again.

The discovery process requires that the *Client* locate a service that can provide it with information about *Servers* on the network. The most basic approach relies on manual entry of URLs or *MachineNames* which are known to have *Servers* running on them. A more advanced approach sends multicast messages to *MachineDiscoveryServers* running on the local network which respond with the *MachineName* assigned to them. The *Client* can then use the *MachineNames* returned to communicate with the *LocalDiscoveryServers* running on those machines.

Servers shall allow themselves to be discovered by *Clients* by implementing a discovery *Endpoint* and registering themselves with the *LocalDiscoveryServer* running on the same machine. *Application Administrators* may also register *Servers* with the *GlobalDirectoryService* for the system.

The URL for a discovery *Endpoint* provides all of the information that the *Client* needs to connect to the *Endpoint*. This implies that no security can be applied to the message, however, some implementations may use transport layer security where the secure protocol is identified in the URL (e.g. HTTPS).

A discovery *Endpoint* shall provide the *FindServers* and *GetEndpoints* services defined in **Error! Reference s ource not found.**. *Clients* use the *FindServers* service request a list of known *Servers* from the discovery *Endpoint* of a *DiscoveryServer*. *Clients* use the *GetEndpoints* service defined in **Error! Reference source not f ound.** to request a list of supported *Endpoints* from the discovery *Endpoint* of a *Server*.

Servers use the *RegisterServer* service defined in **Error! Reference source not found.** to tell the *L ocalDiscoveryServer* that they are available. *Servers* may call this service when they are installed, however, they shall call this service when they start and continue to call it periodically as long as they are running and accepting *Client* connections. Administrators shall be able to disable *Server* registration because they may not want some *Servers* to be discoverable. By default, all *Servers* shall be configured to call *RegisterServer*.

T

4.13 Aggregates

The **Error! Reference source not found.** specification defines the representation of *Aggregate* historical or b uffered real time data in the OPC Unified Architecture. This includes the definition of *Aggregates* used in processed data retrieval and in historical retrieval. This definition includes both standard *Reference* types and *Object* types.

4.13.1 Aggregate Objects

4.13.1.1 General

OPC UA servers can support several different functionalities and capabilities. The following standard *Objects* are used to expose these capabilities in a common fashion, and there are several standard defined concepts that can be extended by vendors.

4.13.1.1.1 AggregateConfigurationType

The *AggregateConfigurationType* defines the general characteristics of a *Node* that defines the *Aggregate* configuration of any *Variable* or *Property*. *AggregateConfiguration Object* represents the browse entry point for information on how the Server treats *Aggregate* specific functionality such as handling *Uncertain* data. It is formally defined in Table 3.

Table 3 – AggregateConfigurationType Definition

Attribute	Value				
BrowseName	AggregateConfigurationType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *BaseObjectType* defined in **Error! Reference source not found.**					
HasProperty	Variable	TreatUncertainAsBad	Boolean	PropertyType	Mandatory
HasProperty	Variable	PercentDataBad	Byte	PropertyType	Mandatory
HasProperty	Variable	PercentDataGood	Byte	PropertyType	Mandatory
HasProperty	Variable	UseSlopedExtrapolation	Boolean	PropertyType	Mandatory

The *TreatUncertainAsBad Variable* indicates how the server treats data returned with a *StatusCode* severity *Uncertain* with respect to *Aggregate* calculations. A value of True indicates the server considers the severity equivalent to *Bad*, a value of False indicates the server considers the severity equivalent to *Good,* unless the aggregate definition says otherwise. The default value is True. Note that the value is still treated as *Uncertain* when the *StatusCode* for the result is calculated.

The *PercentDataBad Variable* indicates the minimum percentage of bad data in a given interval required for the *StatusCode* for the given interval for processed data request to be set to *Bad*. (*Uncertain* is treated as defined above). For details on which *Aggregates* use the *PercentDataBad Variable*, see the definition of each *Aggregate*. The default value is 100.

The *PercentDataGood Variable* indicates the minimum percentage of *Good* data in a given interval required for the *StatusCode* for the given interval for the processed data requests to be set to *Good*. For details on which *Aggregates* use the *PercentDataGood Variable*, see the definition of each *Aggregate*. The default value is 100.

The *PercentDataGood* and *PercentDataBad* must follow the following relationship *PercentDataGood >= (100 – PercentDataBad)*. If they are equal the result of the *PercentDataGood* calculation is used.

The *UseSlopedExtrapolation Variable* indicates how the server interpolates data when no boundary value exists (i.e. extrapolating into the future from the last known value). A value of False indicates that the server will use a *SteppedExtrapolation* format, and hold the last known value constant. A value of True indicates the server

will project the value using *UseSlopedExtrapolation* mode. The default value is False. For *SimpleBounds* this value is ignored.

4.13.1.2   AggregateFunctions Object

This *Object* is used as the browse entry point for information about the *Aggregates* supported by a *Server*. The content of this *Object* is already defined by its type definition. All *Instances* of the *FolderType* use the standard *BrowseName* of 'AggregateFunctions'. The *HasComponent Reference* is used to relate a *ServerCapabilities Object* and/or any *HistoricalServerCapabilites Object* to an *AggregateFunctions Object*.

Table 4 – Aggregate Functions Definition

Attribute	Value				
BrowseName	AggregateFunctions				
References	**Node Class**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasTypeDefinition	Object Type	*FolderType*	Defined in **Error! Reference source not found.**		

Each ServerCapabilities and HistoricalServerCapabilities Object shall reference an AggregateFunctions Object. In addition, each HistoricalConfiguration Object belonging to a HistoricalDataNode may reference an AggregateFunctions Object using the HasComponent Reference.

4.13.1.2.1   AggregateFunctionType

This *ObjectType* defines an *Aggregate* supported by a UA server. This *Object* is formally defined in Table 5.

Table 5 – AggregateFunctionType Definition

Attribute	Value				
BrowseName	AggregateFunctionType				
IsAbstract	False				
References	**Node Class**	**BrowseName**	**DataType**	**Type Definition**	**Mod. Rule**
Subtype of the *BaseObjectType* defined in **Error! Reference source not found.**					

For the *AggregateFunctionType*, the *Description Attribute* (inherited from the *Base NodeClass*), is mandatory. The *Description Attribute* provides a localized description of the *Aggregate*.

Table 6 specifies the *BrowseName* and *Description Attributes* for the standard *Aggregate Objects*. The description is the localized "en" text. For other locales it shall be translated.

T

Table 6 – Standard AggregateType Nodes

BrowseName	Description
	Interpolation Aggregate
Interpolative	At the beginning of each interval, retrieve the calculated value from the data points on either side of the requested timestamp.
	Data Averaging and Summation Aggregates
Average	Retrieve the average value of the data over the interval.
TimeAverage	Retrieve the time weighted average data over the interval using *Interpolated Bounding Values*.
TimeAverage2	Retrieve the time weighted average data over the interval using *Simple Bounding Values.*
Total	Retrieve the total (time integral) of the data over the interval using *Interpolated Bounding Values.*
Total2	Retrieve the total (time integral) of the data over the interval using *Simple Bounding Values*.
	Data Variation Aggregates
Minimum	Retrieve the minimum raw value in the interval with the timestamp of the start of the interval.
Maximum	Retrieve the maximum raw value in the interval with the timestamp of the start of the interval.
MinimumActualTime	Retrieve the minimum value in the interval and the Timestamp of the minimum value.
MaximumActualTime	Retrieve the maximum value in the interval and the Timestamp of the maximum value.
Range	Retrieve the difference between the minimum and maximum Value over the interval.
Minimum2	Retrieve the minimum value in the interval including the *Simple Bounding Values.*
Maximum2	Retrieve the maximum value in the interval including the *Simple Bounding Values.*
MinimumActualTime2	Retrieve the minimum value with the actual timestamp including the *Simple Bounding Values.*
MaximumActualTime2	Retrieve the maximum value with the actual timestamp including the *Simple Bounding Values.*
Range2	Retrieve the difference between the Minimum2 and Maximum2 value over the interval.
	Counting Aggregates
Count	Retrieve the number of raw values over the interval.
DurationInStateZero	Retrieve the time a Boolean or numeric was in a zero state using *Simple Bounding Values*.
DurationInStateNonZero	Retrieve the time a Boolean or numeric was in a non-zero state using *Simple Bounding Values*.
NumberOfTransitions	Retrieve the number of changes between zero and non-zero that a Boolean or Numeric value experienced in the interval.
	Time Aggregates
Start	Retrieve the value at the beginning of the interval using *Interpolated Bounding Values*.
End	Retrieve the value at the end of the interval using *Interpolated Bounding Values*.
Delta	Retrieve the difference between the Start and End value in the interval.

StartBound	Retrieve the value at the beginning of the interval using *Simple Bounding Values*.
EndBound	Retrieve the value at the end of the interval using *Simple Bounding Values*.
DeltaBounds	Retrieve the difference between the StartBound and EndBound value in the interval.
	Data Quality Aggregates
DurationGood	Retrieve the total duration of time in the interval during which the data is good.
DurationBad	Retrieve the total duration of time in the interval during which the data is bad.
PercentGood	Retrieve the percent of data (0 to 100) in the interval which has good *StatusCode*.
PercentBad	Retrieve the percent of data (0 to 100) in the interval which has bad *StatusCode*.
WorstQuality	Retrieve the worst *StatusCode* of data in the interval.
WorstQuality2	Retrieve the worst *StatusCode* of data in the interval including the *Simple Bounding Values*.
	Historical Annotation Aggregates
AnnotationCount	Retrieve the number of *Annotation*s in the interval. (applies to Historical *Aggregates* only)
	Statistical Aggregates
StandardDeviationSample	Retrieve the standard deviation for the interval for a sample of the population (n-1).
VarianceSample	Retrieve the variance for the interval as calculated by the StandardDeviationSample.
StandardDeviation Population	Retrieve the standard deviation for the interval for a complete population (n) which includes *Simple Bounding Values*.
VariancePopulation	Retrieve the variance for the interval as calculated by the StandardDeviationPopulation which includes *Simple Bounding Values*.

4.13.2 MonitoredItem AggregateFilters

4.13.2.1   MonitoredItem AggregateFilter Defaults

The default values used for *MonitoredItem Aggregates* are the same as those used for historical *Aggregates*. For additional information on *MonitoredItem AggregateFilters* see **Error! Reference source not found.**.

4.13.2.2 MonitoredItem Aggregates and Bounding Values

When calculating *MonitoredItem* Aggregates that require the use of Bounding Values, the bounds may not be known. The calculation is done in the same manner as a historical read with the Partial Bit set. The historian may wait some amount of time (normally no more than one processing interval) before calculating the interval to allow for any latency in data collection and reduce the use of the Partial Bit.

A historical read done after data collection and the data from *MonitoredItems* over the same interval may not be the same.

4.13.3 Exposing Supported Functions and Capabilities

Figure 39 – Representation of Aggregate Configuration information in the AddressSpace outlines a possible representation of *Aggregate* information in the *AddressSpace*. In this example, although the Server at the highest level may support aggregate functionality for Interpolative, Total, Average, and others, *DataVariable* X only supports Interpolative, Total and Average, while *DataVariable* Y supports Average, a vendor defined Aggregate and other (unstated) Aggregates.

Figure 39 – Representation of Aggregate Configuration information in the AddressSpace

5 OPC UA Security

This chapter is based on the Whitepaper **The OPC UA Security Model for Administrators, Version 1.00 from July, 2010** (OPC-UA_Security_Model_for_Administrators_V1.00.pdf) but contains several adaptions valid for the OPC UA .NET Standard based solutions.

5.1 Background

A security model is an architecture that allows developers, administrators and end users to use applications in distributed environment while ensuring that the applications, the computers they run on and the information exchanged is not compromised. A complete security model has several facets including application security, transport security, user authorization and authentication and traceability. This white paper describes how to use the OPC UA security model to ensure application and transport security. The target audience for this document are systems administrators and end users. A second whitepaper will discuss the security model from the perspective of a software developer.

The OPC UA security model has been designed to meet the requirements of many different systems while using the same infrastructure. In order to accommodate different security and administrative requirements the OPC UA security model offers four tiers for application authentication and two tiers for certificate management. It is up to the administrator to decide which tiers best match their needs. Applications should support all tiers. This document also discusses the administrative procedures required by a tier. Applications must allow administrators to configure the level of security enforced by their application just like web browsers allow administrators to configure the security level enforced by the browser.

The UA Security Model defines four principal actors: The Application Instance, the Application Administrator, the Application Operator and the Certificate Authority. The relationships between these actors are shown in the following figure. Each of the entities is described in the text that follows.

T
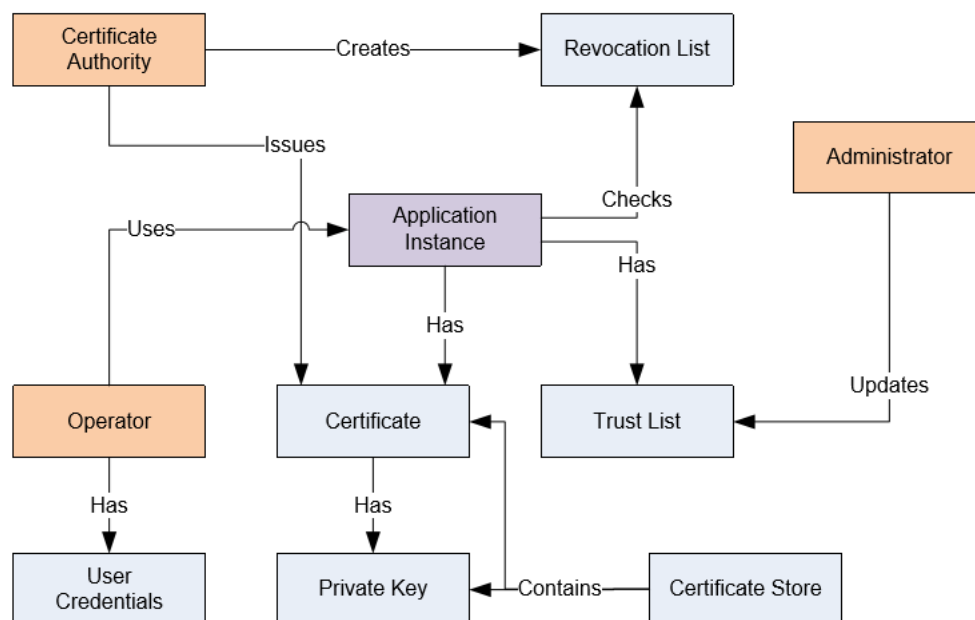
Application Instance – An OPC UA Application installed on a single machine is called an Application Instance. Each instance must have its own Certificate which it uses to identify itself when connecting to other applications. Each Application Instance has a globally unique URI which identifies it.

Administrator - An Application Instance must have an Administrator which manages the security settings for the Application.

Operator – An Operator is person who uses the Application Instance. More than one Operator may exist for any given application. An Operator may have User Credentials which are used to determine access rights and to track activities.

Certificate Authority (CA) - A Certificate Authority (CA) is an administrator or organization which is responsible for creating Certificates. The Certificate Authority must verify that information placed in the Certificate is correct and add a digital signature to the Certificate that is used to verify that the information has not been changed. Each CA must have its own Certificate which is used to create the digital signatures.

Certificate - A Certificate is an electronic ID that can be held by an application. The ID includes information that identifies the holder, the issuer and a unique key that is used to create and verify digital signatures. The syntax of these certificates conforms to the X509 specification, as a result, these certificates are also called "X509 Certificates". Certificates also have a Private Key associated with them.

User Credential – A User Credential is a generic term for an electronic ID which identifies an Operator. It may be passed to a Server after the Application Certificate is used to create a secure channel. It is used to determine access rights and to track activities.

Private Key - A Private Key is a secret number known only to the holder of a Certificate. This secret allows the holder to create digital signatures and decrypt data. If this secret is revealed to unauthorized parties, then the associated Certificate can no longer be trusted or used.

Certificate Store - A Certificate Store is a place where Certificates and Private Keys can be stored on a file system. All Windows systems provide a registry-based store called the Windows Certificate Store. All systems support a directory containing the Certificates stored in a file which is also called an OpenSSL Certificate Store.

Self-Signed Certificate - A Self-Signed Certificate is a Certificate which has no Certificate Authority. These Certificates can be created by anyone and can be used in situations where the administrators of UA Applications are able to verify the claims by reviewing the contents themselves and security is addressed in another manner.

Trust List - A Trust List is a list of Certificates which are trusted by an Application Instance. When security is enabled UA Applications must reject connections from peers if they do not have a Certificate that is in the trusted or issued by a CA that is in the Trust List.

Revocation List - A Revocation List is a list of Certificates which have been revoked by a CA and must not be accepted by an Application Instance.

T

5.2 Security Tiers

5.2.1 The Basics

In OPC UA, each installation of an application must have an application instance certificate that uniquely identifies the application and the machine that it is running on. These certificates come with private keys that allow applications to create secure communication channels that cannot be viewed by 3rd parties or modified while in transit. These certificates also allow OPC UA applications to be identified by peers and to block communication from a peer if it is not authorized.

5.2.2 Tier 1 - No Authentication

In this tier the client and server allow any peer to communicate which means that all valid certificates are trusted. The application certificates are used only to provide unverifiable information about the peer. The receiver has no way to know if the sender is the legitimate holder of the certificate.

In this mode the client and server automatically accept valid certificates even if they have not been explicitly added to the trust lists managed by the client and server applications. This mode requires no configuration at the server or client.

This tier cannot ensure the privacy of any information transmitted, including user credentials. This tier would only be appropriate in a system that has guaranteed security in some other manner, such as a physically secured and isolated system or where communications is secured via VPN or other such transport layer security. It would also be appropriate in a situation where all information is public and access to it is open.

A developer need only configure an installation procedure that generates an application instance certificate for an application on installation.

5.2.3 Tier 2 - Server Authentication

In this tier the server allows any client to connect and if user authentication is required it is done by sending user credentials such as a username/password after the secure channel has been established. Clients, on the other hand, must be configured by the administrator to trust the Server.

Clients will trust a Server if an administrator has explicitly placed the Server certificate into its trust list or if the Server's certificate was issued by a CA which is in its trust list.

If the Server's certificate was not explicitly in the trust list (i.e. the certificate was issued by a CA that is in the trust list) the client should compare the DNS name in the Server's certificate to the DNS name it used to connect to server. If they match the client knows it is connecting to the machine it thinks it is connecting to. This does not guarantee that the client has connected to correct server, only that the machine is the correct machine.

This tier is used by most Internet banking applications where the bank's web server has a certificate issued by Certificate Authorities like Verisign which are automatically placed in the browsers trust list by the Windows operating system. It provides a fairly good security, but the server cannot restrict the client applications.

5.2.4 Tier 3 - Client Authentication

In this tier the client connects to any server, but the server only allows trusted clients to connect. Clients never provide sensitive information since it does not know if it the server is legitimate.

In this tier clients need no pre-configuration other than the URL of the server. However, Servers will only trust clients with certificates that have been placed by administrators in the server's trust list or if the Clients certificate was issued by a CA which is in its trust list

This mode is used by discovery services which need to ensure that only authorized applications have access to them, but clients don't care if the server is not legitimate. The local discovery server (LDS) operates in this mode and only allows authorized applications to register themselves.

5.2.5 Tier 4 - Mutual Authentication

In this tier both the client and server only allow trusted peers to connect. It offers the highest level of security but requires that both the client and server be configured in advance. This is the recommended mode for any public or semi-public deployment of OPC UA or for deployments where security is a primary concern.

As in Tier 2, clients should check the DNS name if the Server certificate was not explicitly placed in the trust list.

It will be used in environments where administrators want complete control over which applications can talk to each other. It also provides the most secure environment.

Application installation should default to Tier 4 mode

5.3 Certificates and Certificate Stores

5.3.1 Overview

When working with certificates it is important to understand the formats associated with a certificate and where the certificates are stored. Both the formats and storage location vary from platform to platform and vary by the cryptographic library that is being used by an application.

5.3.2 Certificates and Private Keys

Certificates are typically stored in files that can have several formats.

The formats used by UA applications are shown in the following tables.

DER	An ASN.1 blob encoded with the DER (distinguished encoding rules).
	File extension is *.der or *.cer on Windows systems.
	Use only for storing the Certificate (not the Private Key).
	Certificates can be imported or exported to/from Windows Certificate Stores using this format. It is also the file format used to store a certificate in a directory store.
PKSC#12	A binary format used to store RSA private keys with their certificates.
	File extension is *.pfx.
	May be password protected.
	Private keys can be imported or exported to/from Windows Certificate Stores using this format.
PEM	A text format used to store private keys
	File extension is *.pem.
	May be password protected.
	This format is only used by some OpenSSL based applications or windows application, but they include items such as CRL lists.
	Other formats such as *.crt, or *.crl may occur in some systems, but all others can be converted or matched to one of the above, by the operating system

Other formats such as *.crt, or *.crl may occur in some systems, but all others can be converted or matched to one of the above, by the operating system

5.3.3 Windows Certificate Stores

Windows Certificate Stores are accessible via standard Windows tools and WIN32 APIs. They are physically stored in the registry but must be accessed via the standard APIs.

There are two special store locations: LocalMachine and CurrentUser:

- The LocalMachine location contains Certificate stores shared by all users and services on a machine. All users have read access to these stores.

- The CurrentUser (or CurrentService) location contains Certificate stores which are only accessible to the current user or service. These stores can be accessed by administrators via the standard APIs.

The Windows Certificate Stores are identified by the location and a store name separated by a backslash. e.g. LocalMachine\UA Applications or CurrentUser\UA Applications.

Certificates placed in Windows Certificate Stores may have private keys associated with them; however, users will not be able to access these private keys unless they have been granted permission. The UA Certificate Tool can be used to manage permissions for private keys.

Important:

The Windows Certificate Store is no longer supported with the .NET Standard version.

5.3.4 Directory Stores

Any directory can contain *.der, *.pfx or *.pem files. These directories are called a certificate store and identified by the full path of the directory.

By convention the UA Stacks also support a directory store which places the private keys in a separate subdirectory. The subdirectory for certificates is called „certs" and the subdirectory for private keys is called „private". This style of certificate store is also called an OpenSSL store.

An OpenSSL store is identified by the full path of the root directory. The presence of the „certs" subdirectory is used to distinguish between a simple directory store and an OpenSSL store.

OpenSSL may also make use of directories such as crl, which would contain certificate revocation lists. This directory is at the same level as the „certs" directory.

The standard locations of the different used folders are:

ApplicationCertificate:	Where the application instance certificate is stored (MachineDefault): `%CommonApplicationData%\OPC Foundation\pki\own`
TrustedIssuerCertificates:	Where the issuer certificates are stored (certificate authorities). Typical web browsing applications trust any certificate issued by a CA in the "Trusted Root Certification Authorities" certificate store. However, this approach is not appropriate for UA because Adminstrators have no control over the CAs that get placed in that Root store to facilitate web browsing. This means Adminstrators must specify a different store that is used only for UA related CAs and/or they must explicitly specify the certificate for each trusted certification authority: `%CommonApplicationData%\OPC Foundation\pki\issuer`
TrustedPeerCertificates:	Where the trust list is stored. Some UA applications will use self-signed certificates (certificates without a CA) which means that every application which communicates with it must be configured to trust it. Adminstrators may designate a certificate store that contains trusted UA application instance certificates (this store should not be the same as the store used for Cas certificates). Alternately, Administrators may enter the certificates explicitly in this list. Note that entries in this list may either reference a certificate in the store or may contained the entire certificate encoded as base64 data: `%CommonApplicationData%\OPC Foundation\pki\trusted`
RejectedCertificateStore:	The directory used to store invalid certficates for later review by the administrator: `%CommonApplicationData%\OPC Foundation\pki\rejected`
UserIssuerCertificates:	Where the User issuer certificates are stored: `%CommonApplicationData%\OPC Foundation\pki\issuerUser`
TrustedUserCertificates:	Where the User trust list is stored: `%CommonApplicationData%\OPC Foundation\pki\trustedUser`

T

5.3.5 X509 Stores

With the .NET Standard version, the Windows Certificate Stores was replaced with the Folder & OS-level certificate-store X509Store.

This certificate store is best suited for cross platform support.

5.3.5.1 Windows .Net applications

By default, the self signed certificates are stored in a X509Store called CurrentUser\UA_MachineDefault. The certificates can be viewed or deleted with the Windows Certificate Management Console (certmgr.msc). The trusted, issuer and rejected stores remain in a folder called OPC Foundation\CertificateStores with a root folder which is specified by the SpecialFolder variable %CommonApplicationData%. On Windows 7/8/8.1/10 this is usually the invisible folder C:\ProgramData.

5.3.5.2 .Net Standard Console applications on Windows, Linux, iOS etc.

The self signed certificates are stored in a folder called **OPC Foundation/CertificateStores/MachineDefault** with a root folder which is specified by the SpecialFolder variable **%LocalApplicationData%** or in a **X509Store** called **CurrentUser\My**, depending on the configuration. For best cross platform support the personal store **CurrentUser\My** was chosen to support all platforms with the same configuration. Some platforms, like macOS, do not support arbitrary certificate stores.

The *trusted*, *issuer* and *rejected* stores remain in a shared folder called **OPC Foundation\CertificateStores** with a root folder specified by the SpecialFolder variable **%LocalApplicationData%**. Depending on the target platform, this folder maps to a hidden location under the user home directory. The standard locations of the different used folders are:

ApplicationCertificate:	Where the application instance certificate is stored (MachineDefault): `CurrentUser\My`
TrustedIssuerCertificates:	Where the issuer certificates are stored (certificate authorities). Typical web browsing applications trust any certificate issued by a CA in the "Trusted Root Certification Authorities" certificate store. However, this approach is not appropriate for UA because Adminstrators have no control over the CAs that get placed in that Root store to facilitate web browsing. This means Adminstrators must specify a different store that is used only for UA related CAs and/or they must explicitly specify the certificate for each trusted certification authority: `%LocalApplicationData%/OPC Foundation/pki/issuer`
TrustedPeerCertificates:	Where the trust list is stored. Some UA applications will use self-signed certificates (certificates without a CA) which means that every application which communicates with it must be configured to trust it. Adminstrators may designate a certificate store that contains trusted UA application instance certificates (this store should not be the same as the store used for Cas certificates). Alternately, Administrators may enter the certificates explicitly in this list. Note that entries in this list may either reference a certificate in the store or may contained the entire certificate encoded as base64 data: `%LocalApplicationData%/OPC Foundation/pki/trusted`
RejectedCertificateStore:	The directory used to store invalid certficates for later review by the administrator: `%LocalApplicationData%/OPC Foundation/pki/rejected`

5.4 Key Certificate Properties

A certificate contains several fields that have functionality as defined in the x.509 specification. A few key fields are listed here:

Common Name:	This is usually the Application Name obtained from the configuration file associated with the application for which the certificate is being generated, but it must be an appropriate name for the application.
Organization:	This is usually a description of the organization where the application is installed.
SubjectAltName: URI:	This is the unique Application URI associated with the application and is obtained from the configuration file associated with the application for which the certificate is being generated. Only one URI field may be in a certificate.
SubjectAltName: DNSName IPAddress	The DNS name or IP Address of the machine where this application is installed. Multiple DNS Names and/or IP Addresses can be in a single certificate.
Valid from Valid to:	This is the starting and ending date for which a certificate is valid. The Automatic certificate checking will verify that the current date is between these dates when validating a certificate. The administrator should check these dates periodically and replace any certificates that are about to expire with new certificates.

These fields are usually used by an administrator to help identify and match a certificate to an application. OPC UA applications also automatically check some of these fields when verifying a received certificate from an application. Certificates contain additional information, but this information is usually provided by the tool that generates the certificate.

T

6 Standard Node Classes

OPC UA 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. This chapter gives an introduction in the mappings from OPC UA related objects to .NET classes. As specified in chapter 4.3 of the OPC UA .NET Introduction the following *NodeClasses* exists:

- Base NodeClass (NodeState in .NET)
- ReferenceType NodeClass (ReferenceTypeState in .NET)
- View NodeClass (ViewState in .NET)
- Object NodeClass (BaseObjectState in .NET)
- ObjectType NodeClass (BaseObjectTypeState in .NET)
- Variable NodeClass (BaseVariableState in .NET)
- VariableType NodeClass (BaseVariableTypeState in .NET)
- Method NodeClass (MethodState in .NET)

6.1 DataTypes

- DataType NodeClass (DataTypeState in .NET)

An overview of the class hierarchy in .NET is given in the following picture:
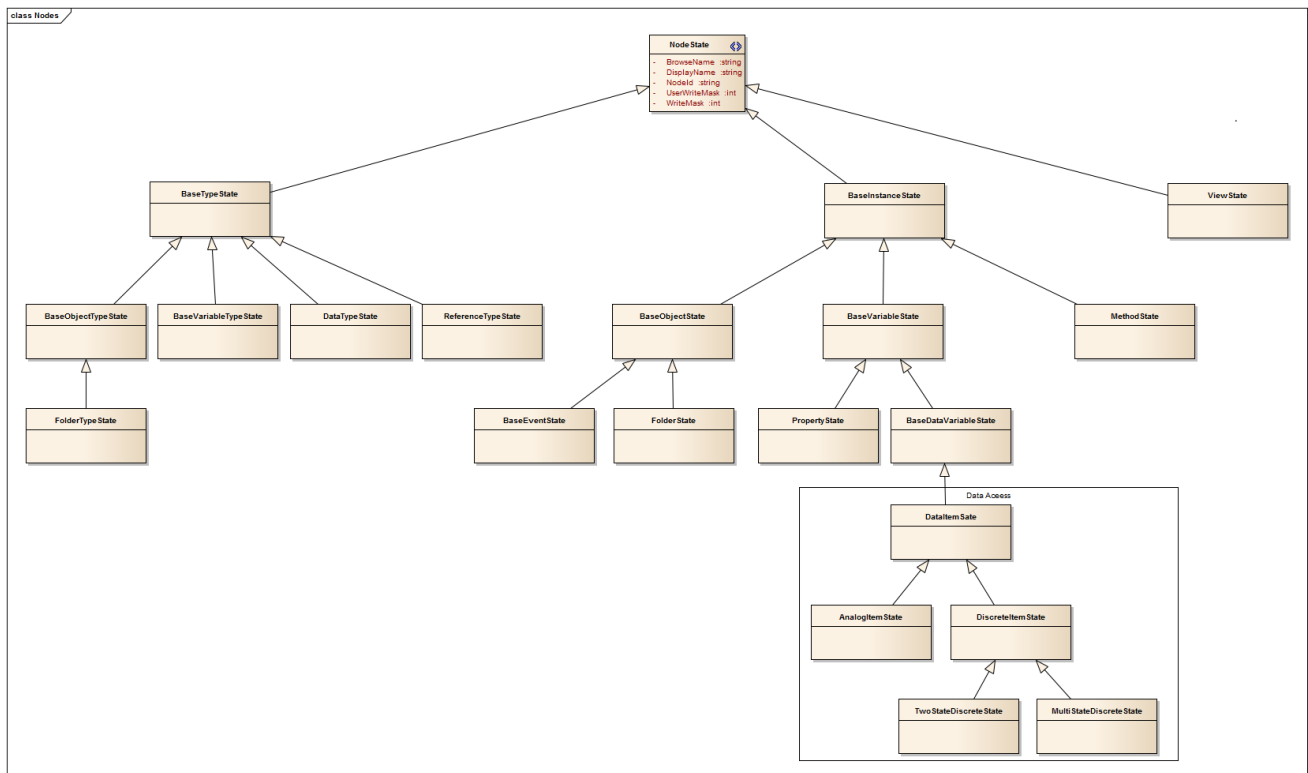
6.2 Base NodeClass

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 chapter 4.3.1 of the OPC UA .NET Standard Introduction). The Attributes of the Base NodeClass are specified in Part 3 of the OPC UA specification. There are no References specified for the Base NodeClass. The Base NodeClass is represented by the NodeState class in .NET.

Name	Use	Data Type
Attributes		
NodeId	M	NodeId
NodeClass	M	NodeClass
BrowseName	M	QualifiedName
DisplayName	M	LocalizedText
Description	O	LocalizedText
WriteMask	O	AttributeWriteMask
UserWriteMask	O	AttributeWriteMask
RolePermissions	O	RolePermissionType[]
UserRolePermissions	O	RolePermissionType[]
AccessRestrictions	O	AccessRestrictionsType
References		

NodeState
Abstract Class

▷ Fields

◢ Properties

- AccessRestrictions
- AreEventsMonitored
- BrowseName
- ChangeMasks
- Description
- DisplayName
- Extensions
- Handle
- Initialized
- NodeClass
- NodeId
- RolePermissions
- SymbolicName
- UserRolePermissions
- UserWriteMask
- ValidationRequired
- WriteMask

▷ Methods

▷ Events

▷ Nested Types

T

Name	Description
NodeId	Nodes are unambiguously identified using a constructed identifier called the NodeId. Some Servers may accept alternative NodeIds in addition to the canonical NodeId represented in this Attribute. A Server shall persist the NodeId of a Node, that is, it shall not generate new NodeIds when rebooting.
NodeClass	The NodeClass Attribute identifies the NodeClass of a Node.
BrowseName	Nodes have a BrowseName Attribute that is used as a non-localised human-readable name when browsing the AddressSpace to create paths out of BrowseNames. The TranslateBrowsePathsToNodeIds Service defined in OPC 10000-4 can be used to follow a path constructed of BrowseNames. A BrowseName should never be used to display the name of a Node. The DisplayName should be used instead for this purpose. Unlike NodeIds, the BrowseName cannot be used to unambiguously identify a Node. Different Nodes may have the same BrowseName. Subclause 8.3 defines the structure of the BrowseName. It contains a namespace and a string. The namespace is provided to make the BrowseName unique in some cases in the context of a Node (e.g. Properties of a Node) although not unique in the context of the Server. If different organizations define BrowseNames for Properties, the namespace of the BrowseName provided by the organization makes the BrowseName unique, although different organizations may use the same string having a slightly different meaning. Servers may often choose to use the same namespace for the NodeId and the BrowseName. However, if they want to provide a standard Property, its BrowseName shall have the namespace of the standards body although the namespace of the NodeId reflects something else, for example the local Server. It is recommended that standard bodies defining standard type definitions use their namespace for the NodeId of the TypeDefinitionNode as well as for the BrowseName of the TypeDefinitionNode. The string-part of the BrowseName is case sensitive. That is, Clients shall consider them case sensitive. Servers are allowed to handle BrowseNames passed in Service requests as case insensitive. Examples are the TranslateBrowsePathsToNodeIds Service or Event filter.
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 OPC 10000-4 for a description of session establishment and locales. Subclause 8.5 defines the structure of the DisplayName. The string part of the DisplayName is restricted to 512 characters.
Description	The optional Description Attribute shall explain the meaning of the Node in a localised text using the same mechanisms for localisation as described for the DisplayName.
WriteMask	The optional WriteMask Attribute exposes the possibilities of a client to write the Attributes of the Node. The WriteMask Attribute does not take any user access rights into account, that is, although an Attribute is writable this may be restricted to a certain user/user group. If the OPC UA Server does not have the ability to get the WriteMask information for a specific Attribute from the underlying system, it should state that it is writable. If a write operation is called on the Attribute, the Server should transfer this request and return the corresponding StatusCode if such a request is rejected. StatusCodes are defined in OPC 10000-4. The AttributeWriteMask DataType is defined in 0.

UserWriteMask	The optional UserWriteMask Attribute exposes the possibilities of a client to write the Attributes of the Node taking user access rights into account. It uses the AttributeWriteMask DataType which is defined in 0.
	The UserWriteMask Attribute can only further restrict the WriteMask Attribute, when it is set to not writable in the general case that applies for every user.
	Clients cannot assume an Attribute can be written based on the UserWriteMask Attribute.It is possible that the Server may return an access denied error due to some server specific change which was not reflected in the state of this Attribute at the time the Client accessed it.
RolePermissions	The optional RolePermissions Attribute specifies the Permissions that apply to a Node for all Roles which have access to the Node.
UserRolePermissions	The optional UserRolePermissions Attribute specifies the Permissions that apply to a Node for all Roles granted to current Session.
AccessRestrictions	The optional AccessRestrictions Attribute specifies the AccessRestrictions that apply to a Node. Its data type is defined in 8.56. If a Server supports AccessRestrictions for a particular Namespace it adds the DefaultAccessRestrictions Property to the NamespaceMetadata Object for that Namespace (see Figure 8). If a particular Node in the Namespace needs to override the default value the Server adds the AccessRestrictions Attribute to the Node.
	If a Server implements a vendor specific access restriction model for a Namespace, it does not add the DefaultAccessRestrictions Property to the NamespaceMetadata Object.

6.3 ReferenceType NodeClass

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

This standard defines a set of ReferenceTypes provided as an inherent part of the OPC UA Address Space Model. These ReferenceTypes are defined in Clause 7 and their representation in the AddressSpace is defined in OPC 10000-5. Servers may also define ReferenceTypes. In addition, OPC 10000-4 defines NodeManagement Services that allow Clients to add ReferenceTypes to the AddressSpace.

ReferenceTypeState
Class
→ BaseTypeState

▷ Fields

▲ Properties
  🔧 InverseName
  🔧 Symmetric

▷ Methods

Name	Use	Data Type
Attributes		
Base NodeClass Attributes	M	--
IsAbstract	M	Boolean
Symmetric	M	Boolean
InverseName	O	LocalizedText
References		
HasProperty	0..*	
HasUbtype	0..*	
Standard Properties		
NodeVersion	O	String

T

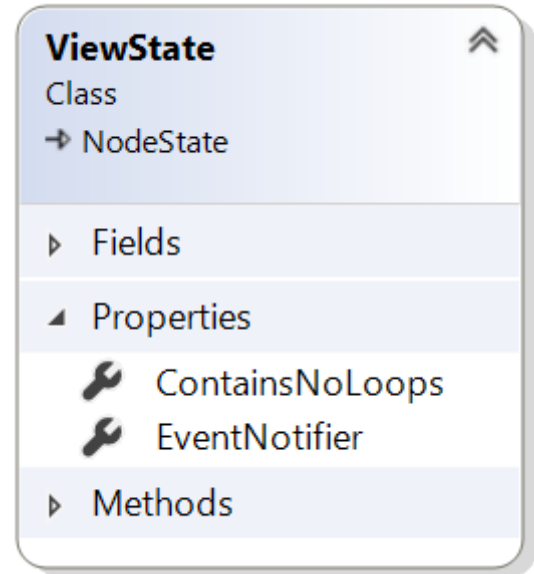| Name	Description
Base NodeClass Attributes	Inherited from the Base NodeClass.
IsAbstract	A boolean Attribute with the following values: TRUE it is an abstract ReferenceType, i.e. no Reference of this type shall exist, only of its subtypes. FALSE it is not an abstract ReferenceType, i.e. References of this type can exist.
Symmetric	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	The inverse name of the Reference, which is the meaning of the ReferenceType as seen from the TargetNode.
HasProperty	HasProperty References are used to identify the Properties of a ReferenceType and shall only refer to Nodes of the Variable NodeClass. The Property NodeVersion is used to indicate the version of the ReferenceType. There are no additional Properties defined for ReferenceTypes in this standard. Additional parts this series of standards may define additional Properties for ReferenceTypes
HasSubtype	HasSubtype References are used to define subtypes of ReferenceTypes. It is not required to provide the HasSubtype Reference for the supertype, but it is required that the subtype provides the inverse Reference to its supertype. The following rules for subtyping apply. a) The semantic of a ReferenceType (e.g. "spans a hierarchy") is inherited to its subtypes and can be refined there (e.g. "spans a special hierarchy"). The DisplayName, and also the InverseName for non-symmetric ReferenceTypes, reflect the specialization. b) If a ReferenceType specifies some constraints (e.g. "allow no loops") this is inherited and can only be refined (e.g. inheriting "no loops" could be refined as "shall be a tree – only one parent") but not lowered (e.g. "allow loops"). c) The constraints concerning which NodeClasses can be referenced are also inherited and can only be further restricted. That is, if a ReferenceType "A" is not allowed to relate an Object with an ObjectType, this is also true for its subtypes. d) A ReferenceType shall have exactly one supertype, except for the References ReferenceType defined in 7.2 as the root type of the ReferenceType hierarchy. The ReferenceType hierarchy does not support multiple inheritances.
NodeVersion	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.

6.4 View NodeClass

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

To address this problem, this standard 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.

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

ViewState
Class
→ NodeState

▷ Fields

◢ Properties
  🔧 ContainsNoLoops
  🔧 EventNotifier

▷ Methods

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

Name	Use	Data Type
Attributes		
Base NodeClass Attributes	M	--
ContainsNoLoops	M	Boolean
EventNotifier	M	Byte
References		
HierarchicalReferences	0..*	
HasProperty	0..*	
Standard Properties		
NodeVersion	O	String
ViewVersion	O	UInt32

Name	Description
Base NodeClass Attributes	Inherited from the Base NodeClass.
ContainsNoLoops	If set to "true" this Attribute indicates that by following the References in the context of the View there are no loops, i.e. starting from a Node "A" contained in the View and following the forward References in the context of the View Node "A" will not be reached again. It does not specify that there is only one path starting from the View Node to reach a Node contained in the View. If set to "false" this Attribute indicates that following References in the context of the View may lead to loops.

EventNotifier	The EventNotifier Attribute is used to indicate if the Node can be used to subscribe to Events or to read / write historic Events. The EventNotifier is an 8-bit unsigned integer

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

	The second two bits also indicate if the history of the Events is available via the OPC UA Server.
HierarchicalReferences	Top level Nodes in a View are referenced by hierarchical References
HasProperty	HasProperty References identify the Properties of the View.
NodeVersion	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.
ViewVersion	The version number for the View. When Nodes are added to or removed from a View, the value of the ViewVersion Property is updated. Clients may detect changes to the composition of a View using this Property. The value of the ViewVersion shall always be greater than 0.

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

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

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

To avoid recursion, i.e. getting all Events of the Server, the Server Object defined in OPC 10000-5 shall never be part of any View since it provides all Events of the Server.

Views are defined by the Server. The browsing and querying Services defined in OPC 10000-4 expect the NodeId of a View Node to provide these Services in the context of the View.

HasProperty References are used to identify the Properties of a View. The Property NodeVersion is used to indicate the version of the View Node. The ViewVersion Property indicates the version of the content of the View. In contrast to the NodeVersion, the ViewVersion Property is updated even if Nodes not directly referenced by the View Node are added to or deleted from the View. This Property is optional because it might not be possible for Servers to detect changes in the View contents. Servers may also generate a

ModelChangeEvent, described in 9.32, if Nodes are added to or deleted from the View. There are no additional Properties defined for Views in this document. Additional parts of this series of standards may define additional Properties for Views.
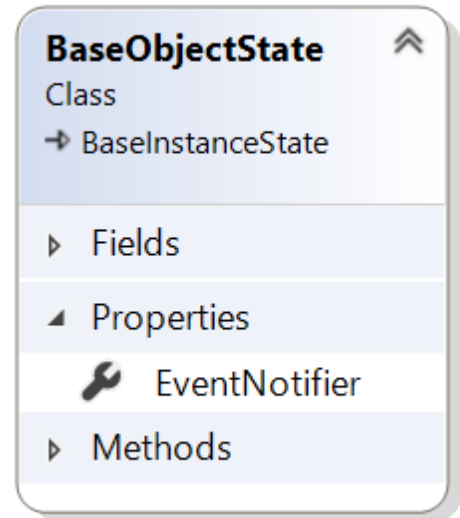
Views can be the SourceNode of any hierarchical Reference. They shall not be the SourceNode of any non-hierarchical Reference.

6.5 Objects

6.5.1 Object NodeClass

Objects are used to represent systems, system components, real-world objects and software objects. Objects are defined using the BaseObjectState class.

Name	Use	Data Type
Attributes		
Base NodeClass Attributes	M	--
EventNotifier	M	EventNotifierType
References		
HasComponent	0..*	
HasProperty	0..*	
HasModellingRule	0..1	
HasTypeDefinition	1	
HasEventSource	0..*	
HasNotifier	0..*	
Organizes	0..*	
<other References>	0..*	
Standard Properties		
NodeVersion	O	String
Icon	O	Image
NamingRule	O	NamingRuleType

BaseObjectState
Class
→ BaseInstanceState

▷ Fields

▲ Properties

🔧 EventNotifier

▷ Methods

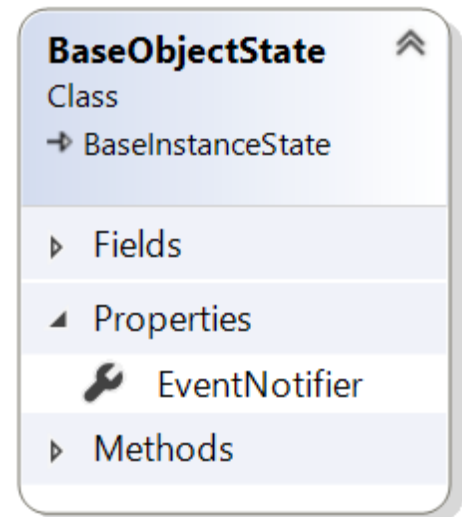| Name	Description
Base NodeClass Attributes	Inherited from the Base NodeClass.
EventNotifier	The EventNotifier Attribute is used to indicate if the Node can be used to subscribe to Events or the read / write historic Events.
HasComponent	HasComponent References identify the DataVariables, the Methods and Objects contained in the Object.
HasProperty	HasProperty References identify the Properties of the Object.
HasModellingRule	Objects can point to at most one ModellingRule Object using a HasModellingRule Reference
HasTypeDefinition	The HasTypeDefinition Reference points to the type definition of the Object. Each Object shall have exactly one type definition and therefore be the SourceNode of exactly one HasTypeDefinition Reference pointing to an ObjectType.
HasEventSource	The HasEventSource Reference points to event sources of the Object. References of this type can only be used for Objects having their "SubscribeToEvents" bit set in the EventNotifier Attribute.
Organizes	This Reference should be used only for Objects of the ObjectType FolderType
<other References>	Objects may contain other References.

NodeVersion	The NodeVersion Property is used to indicate the version of a Node. The NodeVersion Property is updated each time a Reference is added or deleted to the Node the Property belongs to. Attribute value changes do not cause the NodeVersion to change. Clients may read the NodeVersion Property or subscribe to it to determine when the structure of a Node has changed.
Icon	The Icon Property provides an image that can be used by Clients when displaying the Node. It is expected that the Icon Property contains a relatively small image.
NamingRule	The NamingRule Property defines the NamingRule of a ModellingRule. This Property shall only be used for Objects of the type ModellingRuleType.

6.5.2 ObjectType NodeClass

ObjectTypes provde definitions for Objects. ObjectTypes are defined using the ObjectType NodeClass.

BaseObjectState
Class
→ BaseInstanceState

▷ Fields

◢ Properties
    🔧 EventNotifier

▷ Methods

Name	Use	Data Type
Attributes		
Base NodeClass Attributes	M	--
IsAbstract	M	Boolean
References		
HasComponent	0..*	
HasProperty	0..*	
HasModellingRule	0..1	
HasTypeDefinition	1	
HasEventSource	0..*	
HasNotifier	0..*	
Organizes	0..*	
<other References>	0..*	
Standard Properties		
NodeVersion	O	String
Icon	O	Image

Name	Description
Base NodeClass Attributes	Inherited from the Base NodeClass.
EventNotifier	The EventNotifier Attribute is used to indicate if the Node can be used to subscribe to Events or the read / write historic Events.
HasComponent	HasComponent References identify the DataVariables, the Methods and Objects contained in the Object.
HasProperty	HasProperty References identify the Properties of the Object.
HasModellingRule	Objects can point to at most one ModellingRule Object using a HasModellingRule Reference
HasTypeDefinition	The HasTypeDefinition Reference points to the type definition of the Object. Each Object shall have exactly one type definition and therefore be the SourceNode of exactly one HasTypeDefinition Reference pointing to an ObjectType.

T

HasEventSource	The HasEventSource Reference points to event sources of the Object. References of this type can only be used for Objects having their "SubscribeToEvents" bit set in the EventNotifier Attribute.
Organizes	This Reference should be used only for Objects of the ObjectType FolderType
<other References>	Objects may contain other References.
NodeVersion	The NodeVersion Property is used to indicate the version of a Node. The NodeVersion Property is updated each time a Reference is added or deleted to the Node the Property belongs to. Attribute value changes do not cause the NodeVersion to change. Clients may read the NodeVersion Property or subscribe to it to determine when the structure of a Node has changed.
Icon	The Icon Property provides an image that can be used by Clients when displaying the Node. It is expected that the Icon Property contains a relatively small image.

T

6.6 Variables

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

6.6.1 Variable NodeClass

Variables are used to represent values which may be simple or complex. Variables are defined by VariableTypes.

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.

Name	Use	Data Type
Attributes		
Base NodeClass Attributes	M	--
Value	M	Defined by the DataType Attribute
DataType	M	NodeId
ValueRank	M	Int32
ArrayDimensions	O	UInt32[]
AccessLevel	M	AccessLevelType
UserAccessLevel	M	AccessLevelType
MinimumSamplingInterval	O	Duration
Historizing	M	Boolean
AccessLevelEx	O	AccessLevelExTy pe
References		
HasModellingRule	0..1	
HasProperty	0..*	
HasComponent	0..*	
HasTypeDefinition	1	
<other References>	0..*	
Standard Properties		
NodeVersion	O	String
LocalTime	O	TimeZone DataType
AllowNulls	O	Boolean
ValueAsText	O	Localized Text
MaxStringLength	O	UInt32
MaxCharacters	O	UInt32
MaxByteStringLength	O	UInt32
MaxArrayLength	O	UInt32
EngineeringUnits	O	EU Information

BaseVariableState
Abstract Class
→ BaseInstanceState

▷ Fields

▲ Properties

🔧 AccessLevel
🔧 AccessLevelEx
🔧 ArrayDimensions
🔧 CopyPolicy
🔧 DataType
🔧 Historizing
🔧 IsValueType
🔧 MinimumSamplingInterval
🔧 StatusCode
🔧 Timestamp
🔧 UserAccessLevel
🔧 Value
🔧 ValueRank
🔧 WrappedValue

▷ Methods

T

Name	Description
Base NodeClass Attributes	Inherited from the Base NodeClass.
Value	The most recent value of the Variable that the Server has. Its data type is defined by the DataType Attribute. It is the only Attribute that does not have a data type associated with it. This allows all Variables to have a value defined by the same Value Attribute.
DataType	NodeId of the DataType definition for the Value Attribute
ValueRank	This Attribute indicates whether the Value Attribute of the Variable is an array and how many dimensions the array has.
	It may have the following values:
	n > 1: the Value is an array with the specified number of dimensions.
	OneDimension (1): The value is an array with one dimension.
	OneOrMoreDimensions (0): The value is an array with one or more dimensions.
	Scalar (−1): The value is not an array.
	Any (−2): The value can be a scalar or an array with any number of dimensions.
	ScalarOrOneDimension (−3): The value can be a scalar or a one dimensional array.
	All DataTypes are considered to be scalar, even if they have array-like semantics like ByteString and String.
ArrayDimensions	This Attribute specifies the maximum supported length of each dimension. If the maximum is unknown the value shall be 0.
	The number of elements shall be equal to the value of the ValueRank Attribute. This Attribute shall be null if ValueRank ≤ 0.
	For example, if a Variable is defined by the following C array:
	Int32 myArray[346];
	then this Variable's DataType would point to an Int32 and the Variable's ValueRank has the value 1 and the ArrayDimensions is an array with one entry having the value 346.
	The maximum number of elements of an array transferred on the wire is 2147483647 (max Int32).
AccessLevel	The AccessLevel Attribute is used to indicate how the Value of a Variable can be accessed (read/write) and if it contains current and/or historic data. The AccessLevel does not take any user access rights into account, i.e. although the Variable is writable this may be restricted to a certain user / user group.
UserAccessLevel	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.
MinimumSamplingInterval	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 OPC 10000-4 for a detailed description of sampling interval).
	A MinimumSamplingInterval of 0 indicates that the Server is to monitor the item continuously. A MinimumSamplingInterval of -1 means indeterminate.
Historizing	The Historizing Attribute indicates whether the Server is actively collecting data for the history of the Variable. This differs from the AccessLevel Attribute which identifies if the Variable has any historical data. A value of TRUE indicates that the Server is actively collecting data. A value of FALSE indicates the Server is not actively collecting data. Default value is FALSE.

T

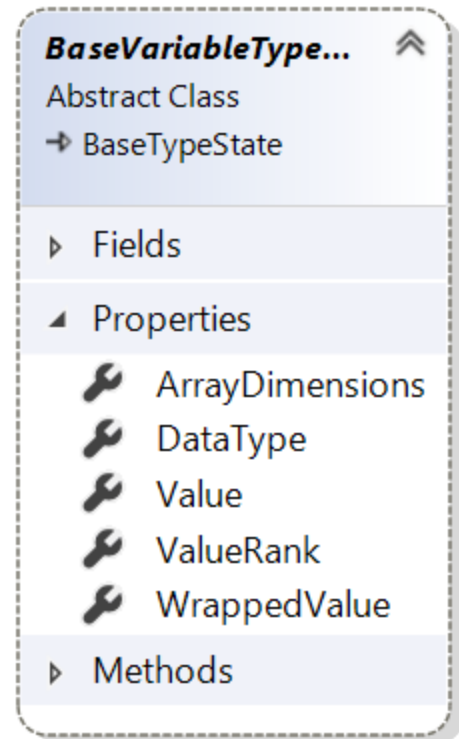|	
AccessLevelEx	The AccessLevelEx Attribute is used to indicate how the Value of a Variable can be accessed (read/write), if it contains current and/or historic data and its atomicity. The AccessLevelEx does not take any user access rights into account, i.e. although the Variable is writable this may be restricted to a certain user / user group. The AccessLevelEx is an extended version of the AccessLevel attribute and as such contains the 8 bits of the AccessLevel attribute as the first 8 bits. The AccessLevelEx is a 32-bit unsigned. If this Attribute is not provided the information provided by these additional Fields is unknown.
HasModellingRule	Variables can point to at most one ModellingRule Object using a HasModellingRule Reference.
HasProperty	HasProperty References are used to identify the Properties of a DataVariable. Properties are not allowed to be the SourceNode of HasProperty References.
HasComponent	HasComponent References are used by complex DataVariables to identify their composed DataVariables. Properties are not allowed to use this Reference.
HasTypeDefinition	The HasTypeDefinition Reference points to the type definition of the Variable. Each Variable shall have exactly one type definition and therefore be the SourceNode of exactly one HasTypeDefinition Reference pointing to a VariableType.
<other References>	Data Variables may be the SourceNode of any other References. Properties may only be the SourceNode of any non-hierarchical Reference.
NodeVersion	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.
LocalTime	The LocalTime Property is only used for DataVariables. It does not apply to Properties. This Property is a structure containing the Offset and the DaylightSavingInOffset flag. The Offset specifies the time difference (in minutes) between the SourceTimestamp (UTC) associated with the value and the time at the location in which the value was obtained. The SourceTimestamp is defined in OPC 10000-4. If DaylightSavingInOffset is TRUE, then Standard/Daylight savings time (DST) at the originating location is in effect and Offset includes the DST correction. If FALSE then the Offset does not include DST correction and DST may or may not have been in effect.
AllowNulls	The AllowNulls Property is only used for DataVariables. It does not apply to Properties. This Property specifies if a null value is allowed for the Value Attribute of the DataVariable. If it is set to true, the Server may return null values and accept writing of null values. If it is set to false, the Server shall never return a null value and shall reject any request writing a null value. If this Property is not provided, it is Server-specific if null values are allowed or not.
ValueAsText	It is used for DataVariables with a finite set of LocalizedTexts associated with its value. For example any DataVariables having an Enumeration DataType. This optional Property provides the localized text representation of the value. It can be used by Clients only interested in displaying the text to subscribe to the Property instead of the value attribute..

MaxStringLength	Only used for DataVariables having a String DataType.
	This optional Property indicates the maximum number of bytes supported by the DataVariable.
MaxCharacters	Only used for DataVariables having a String DataType.
	This optional Property indicates the maximum number of Unicode characters supported by the DataVariable
MaxByteStringLength	Only used for DataVariables having a ByteString DataType.
	This optional Property indicates the maximum number of bytes supported by the DataVariable.
MaxArrayLength	Only used for DataVariables having its ValueRank Attribute not set to scalar.
	This optional Property indicates the maximum length of an array supported by the DataVariable. In a multidimensional array it indicates the overall length. For example, a three-dimensional array of 2 x 3 x 10 has the array length of 60.
	NOTE In order to expose the length of an array of bytes do not use the DataType ByteString but an array of the DataType Byte. In that case the MaxArrayLength applies.
EngineeringUnits	Only used for DataVariables having a Number DataType.
	This optional Property indicates the engineering units for the value of the DataVariable (e.g. hertz or seconds). Details about the Property and what engineering units should be used are defined in OPC 10000-8. The DataType EUInformation is also defined in OPC 10000-8.

6.6.2 VariableType NodeClass

VariableTypes are used to provide type definitions for Variables. VariableTypes are defined using the VariableType NodeClass.

Name	Use	Data Type
Attributes		
Base NodeClass Attributes	M	--
Value	O	Defined by the DataType Attribute
DataType	M	NodeId
ValueRank	M	Int32
ArrayDimensions	O	UInt32[]
IsAbstract	M	AccessLevelType
References		
HasProperty	0..*	
HasComponent	0..*	
HasSubType	0..*	
GeneratesEvent	0..*	
\<other References\>	0..*	
Standard Properties		
NodeVersion	O	String

Name	Description
Base NodeClass Attributes	Inherited from the Base NodeClass.
Value	The default Value for instances of this type.
DataType	NodeId of the data type definition for instances of this type.
ValueRank	This Attribute indicates whether the Value Attribute of the VariableType is an array and how many dimensions the array has. It may have the following values: n > 1: the Value is an array with the specified number of dimensions. OneDimension (1): The value is an array with one dimension. OneOrMoreDimensions (0): The value is an array with one or more dimensions. Scalar (−1): The value is not an array. Any (−2): The value can be a scalar or an array with any number of dimensions. ScalarOrOneDimension (−3): The value can be a scalar or a one dimensional array. NOTE All DataTypes are considered to be scalar, even if they have array-like semantics like ByteString and String.

T

ArrayDimensions	This Attribute specifies the length of each dimension for an array value. The Attribute specifies the maximum supported length of each dimension. If the maximum is unknown the value is 0.
	The number of elements shall be equal to the value of the ValueRank Attribute. This Attribute shall be null if ValueRank ≤ 0.
	For example, if a VariableType is defined by the following C array:
	Int32 myArray[346];
	then this VariableType's DataType would point to an Int32, the VariableType's ValueRank has the value 1 and the ArrayDimensions is an array with one entry having the value 346.
IsAbstract	A boolean Attribute with the following values:
	TRUE: it is an abstract VariableType, i.e. no Variable of this type shall exist, only of its subtypes.
	FALSE: it is not an abstract VariableType, i.e. Variables of this type can exist.
HasProperty	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.
HasComponent	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.
HasSubtype	HasSubtype References identify VariableTypes that are subtypes of this type. The inverse subtype of Reference identifies the parent type of this type.
GeneratesEvent	GeneratesEvent References identify the type of Events instances of this type may generate.
<other References>	VariableTypes may contain other References that can be instantiated by Variables defined by this VariableType. ModellingRules.
NodeVersion	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.

6.7 Method NodeClass

Methods define callable functions. Methods are invoked using the Call Service defined in OPC 10000-4. 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.

Name	Use	Data Type
Attributes		
Base NodeClass Attributes	M	--
Executable	M	Boolean
UserExecutable	M	Boolean
References		
HasProperty	0..*	
HasSubType	0..*	
HasEncoding	0..*	
Standard Properties		
NodeVersion	O	String
InputArguments	O	Argument[]
OutputArguments	O	Argument[]

MethodState
Class
→ BaseInstanceState

▷ Fields

▲ Properties
    🔧 Executable
    🔧 InputArguments
    🔧 MethodDeclarationId
    🔧 OutputArguments
    🔧 UserExecutable

▷ Methods

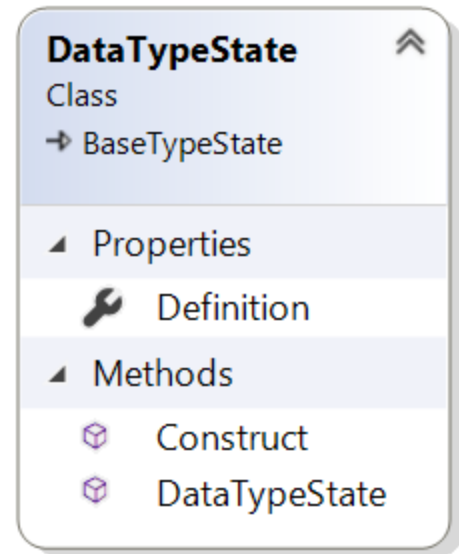| Name	Description
Base NodeClass Attributes	Inherited from the Base NodeClass.
Executable	The Executable Attribute indicates if the Method is currently executable ("False" means not executable, "True" means executable). The Executable Attribute does not take any user access rights into account, i.e. although the Method is executable this may be restricted to a certain user / user group.
UserExecutable	The UserExecutable Attribute indicates if the Method is currently executable taking user access rights into account ("False" means not executable, "True" means executable).
HasProperty	HasProperty References are used to identify the Properties of a DataVariable. Properties are not allowed to be the SourceNode of HasProperty References.
HasModellingRule	Variables can point to at most one ModellingRule Object using a HasModellingRule Reference.
GeneratesEvent	GeneratesEvent References identify the type of Events that will be generated whenever the Method is called..
AlwaysGeneratesEvent	AlwaysGeneratesEvent References identify the type of Events that shall be generated whenever the Method is called..
\<other References\>	Data Variables may be the SourceNode of any other References. Properties may only be the SourceNode of any non-hierarchical Reference.

NodeVersion	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	The InputArguments Property is used to specify the arguments that shall be used by a client when calling the Method.
OutputArguments	The OutputArguments Property specifies the result returned from the Method call.

T

6.8 DataTypes

6.8.1 DataType NodeClass

The DataType NodeClass describes the syntax of a Variable Value. DataTypes are defined using the DataType NodeClass.

Name	Use	Data Type
Attributes		
Base NodeClass Attributes	M	--
IsAbstract	M	Boolean
DataTypeDefinition	O	DataTypeDefinition
References		
HasProperty	0..*	
HasModellingRule	0..1	
GeneratesEvent	0..*	
AlwaysGeneratesEvent	0..*	
<other References>	0..*	
Standard Properties		
NodeVersion	O	String
EnumStrings	O	LocalizedText[]
EnumValues	O	EnumValueType[]
OptionSetValues	O	LocalizedText[]

DataTypeState
Class
→ BaseTypeState

▲ Properties
🔧 Definition

▲ Methods
◈ Construct
◈ DataTypeState

Name	Description
Base NodeClass Attributes	Inherited from the Base NodeClass.
IsAbstract	A boolean Attribute with the following values: TRUE: it is an abstract DataType. FALSE: it is not an abstract DataType.
DataTypeDefinition	The DataTypeDefinition Attribute is used to provide the meta data and encoding information for custom DataTypes. The abstract DataTypeDefinition DataType is defined in 8.48. Structure and Union DataTypes The Attribute is mandatory for DataTypes derived from Structure and Union. For such DataTypes, the Attribute contains a structure of the DataType StructureDefinition. The StructureDefinition DataType is defined in 8.49. It is a subtype of DataTypeDefinition. Enumeration and OptionSet DataTypes The Attribute is mandatory for DataTypes derived from Enumeration, OptionSet and subtypes of UInteger representing an OptionSet. For such DataTypes, the Attribute contains a structure of the DataType EnumDefinition. The EnumDefinition DataType is defined in 8.50. It is a subtype of DataTypeDefinition.
HasProperty	HasProperty References identify the Properties for the DataType.
HasSubtype	HasSubtype References may be used to span a data type hierarchy.

HasEncoding	HasEncoding References identify the encodings of the DataType represented as Objects of type DataTypeEncodingType.
	Only concrete Structured DataTypes may use HasEncoding References. Abstract, Built-in, Enumeration, and Simple DataTypes are not allowed to be the SourceNode of a HasEncoding Reference.
	Each concrete Structured DataType shall point to at least one DataTypeEncoding Object with the BrowseName "Default Binary" or "Default XML" having the NamespaceIndex 0. The BrowseName of the DataTypeEncoding Objects shall be unique in the context of a DataType, i.e. a DataType shall not point to two DataTypeEncodings having the same BrowseName.
NodeVersion	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. Clients shall not use the content for programmatic purposes except for equality comparisions.
EnumStrings	The EnumStrings Property only applies for Enumeration DataTypes. It shall not be applied for other DataTypes. If the EnumValues Property is provided, the EnumStrings Property shall not be provided.
	Each entry of the array of LocalizedText in this Property represents the human-readable representation of an enumerated value. The Integer representation of the enumeration value points to a position of the array.
EnumValues	The EnumValues Property only applies for Enumeration DataTypes. It shall not be applied for other DataTypes. If the EnumStrings Property is provided, the EnumValues Property shall not be provided.
	Using the EnumValues Property it is possible to represent Enumerations with integers that are not zero-based or have gaps (e.g. 1, 2, 4, 8, and 16).
	Each entry of the array of EnumValueType in this Property represents one enumeration value with its integer notation, human-readable representation and help information.
OptionSetValues	The OptionSetValues Property only applies for OptionSet DataTypes and UInteger DataTypes.
	An OptionSet DataType is used to represent a bit mask and the OptionSetValues Property contains the human-readable representation for each bit of the bit mask.
	The OptionSetValues Property provides an array of LocalizedText containing the human-readable representation for each bit.

T

Why Technosoftware GmbH?...

↗ Professionalism
Technosoftware GmbH is, measured by the number of employees, truly not a big company. However, when it comes to flexibility, service quality, and adherence to schedules and reliability, we are surely a great company which can compete against the so-called leaders in the industry. And this is THE crucial point for our customers.

↗ Continuous progress
Lifelong learning and continuing education are, especially in the information technology, essential for future success. Concerning our customers, we will constantly be accepting new challenges and exceeding their requirements again and again. We will continue to do everything to fulfill the needs of our customers and to meet our own standards.

↗ High Quality of Work
We reach this by a small, competent, and dynamic team of coworkers, which apart from the satisfaction of the customer; take care of a high quality of work. We concern the steps necessary for it together with consideration of the customers' requirements.

↗ Support
We support you in all phases – consultation, direction of the project, analysis, architecture & design, implementation, test, and maintenance. You decide on the integration of our coworkers in your project, for an entire project or for selected phases.

Technosoftware GmbH
Windleweg 3, CH-5235 Rüfenach

sales@technosoftware.com
www.technosoftware.com