PLCopen

for efficiency in automation

OPC 30000

OPC UA for Programmable Logic Controllers based on IEC61131-3

Release 1.02

2020-11-25

OPC UA Companion-Specification

Standard Type:	OPC UA Information Model for IEC 61131-3	Comments:	
Title:	OPC Unified Architecture for IEC 61131-3	Date:	2020-11-25
Version:	Release1.02	Software:	MS-Word
Editors:		Source:	OPC 30000 - UA Companion Specification for IEC61131-3 Model 1.02.docx
Owner:	Joint working group "OPCF & PLCopen"	Status:	Release

CONTENTS

FIGURES

TABLES

OPC FOUNDATION, PLCOPEN
AGREEMENT OF USE

REVISION HIGHLIGHTS

This section specifies the interesting changes to the previous revisions:

Version	Date	Description
1.00.00	March 24, 2010	Release first specification 1.0
1.02.00	June 19, 2019	Update to use new UA Companion Specification Template v1.01.11 including additional sections Table 27: Adding additional data types from 3rd edition of IEC61131-3 Table 27: Adding Datatype NodeIDs Table 30: Added SubrangeMin, SubrangeMax Table 31: Added Dimensions, IndexMin, IndexMax Table 34: Added RETAIN, NON_RETAIN, CONSTANT, AT Deprecated Mapping of structure data types Updated NodeSet to meet validation tool

1 Scope

This specification was created by a joint working group of the OPC Foundation and PLCopen. It defines an OPC UA *Information Model* to represent the IEC 61131-3 architectural models.

It is important that the controller as a main component of automation systems is accessible in the vertical information integration which will be strongly influenced by OPC UA. OPC UA servers which represent their underlying manufacturer specific controllers in a similar, IEC 61131-3 based manner provide a substantial advantage for client applications as e.g. visualizations or MES. Controller vendors may reduce costs for the development of these OPC UA servers if an OPC UA Information Model for IEC 61131-3 is used.

OPC Foundation

OPC is the interoperability standard for the secure and reliable exchange of data and information in the industrial automation space and in other industries. It is platform independent and ensures the seamless flow of information among devices from multiple vendors. The OPC Foundation is responsible for the development and maintenance of this standard.

OPC UA is a platform independent service-oriented architecture that integrates all the functionality of the individual OPC Classic specifications into one extensible framework. This multi-layered approach accomplishes the original design specification goals of:

Platform independence: from an embedded microcontroller to cloud-based infrastructure

Secure: encryption, authentication, authorization and auditing

Extensible: ability to add new features including transports without affecting existing applications

Comprehensive information modelling capabilities: for defining any model from simple to complex

PLCopen

PLCopen, as an organization active in industrial control, is creating a higher efficiency in your application software development: in one-off projects as well as in higher volume products. As such it is based on standard available tools to which extensions are and will be defined.

With results like Motion Control Library, Safety, XML specification, Reusability Level and Conformity Level, PLCopen made solid contributions to the community, extending the hardware independence from the software code, as well as reusability of the code and coupling to external software tools. One of the core activities of PLCopen is focused around IEC 61131-3, the only global standard for industrial control programming. It harmonizes the way people design and operate industrial controls by standardizing the programming interface. This allows people with different backgrounds and skills to create different elements of a program during different stages of the software lifecycle: specification, design, implementation, testing, installation and maintenance. Yet all pieces adhere to a common structure and work together harmoniously.

2 Normative references

The following referenced documents are indispensable for the application of this specification. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

IEC 61131-3: 2nd Edition, Subset of 3rd Edition, Programmable Controllers – Part 3: Programming Languages

OPC 10000-1, *OPC Unified Architecture - Part 1: Overview and Concepts*

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

OPC 10000-2, *OPC Unified Architecture - Part 2: Security Model*

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

OPC 10000-3, *OPC Unified Architecture - Part 3: Address Space Model*

http://www.opcfoundation.org/UA/Part3/

OPC 10000-4, *OPC Unified Architecture - Part 4: Services*

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

OPC 10000-5, *OPC Unified Architecture - Part 5: Information Model*

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

OPC 10000-6, *OPC Unified Architecture - Part 6: Mappings*

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

OPC 10000-7, *OPC Unified Architecture - Part 7: Profiles*

http://www.opcfoundation.org/UA/Part7/

OPC 10000-8, *OPC Unified Architecture - Part 8: Data Access*

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

OPC 10000-100, *OPC Unified Architecture – Part 100: Devices*

http://www.opcfoundation.org/UA/Part100/

3 Terms, definitions and conventions

3.1 Overview

It is assumed that basic concepts of OPC UA information modelling and IEC 61131-3 are understood in this specification. This specification will use these concepts to describe the IEC 61131-3 Information Model. For the purposes of this document, the terms and definitions given in OPC 10000-1, OPC 10000-3, OPC 10000-4, OPC 10000-5, OPC 10000-7, OPC 10000-8, OPC 10000-100, IEC 61131-3 as well as the following apply.

To avoid naming conflicts between IEC 61131-3 and OPC UA terms the prefix Ctrl for controller is used together with IEC 61131-3 terms like *Ctrl Variable* or *Ctrl Program*.

Note that OPC UA terms and terms defined in this specification are *italicized* in the specification.

3.2 OPC UA for IEC 61131-3 terms

3.2.1
Controller

a digitally operating electronic system, designed for use in an industrial environment, which uses a programmable memory for the internal storage of user-oriented instructions for implementing specific functions such as logic, sequencing, timing, counting and arithmetic, to control, through digital or analogue inputs and outputs, various types of machines or processes

3.3 Abbreviations and symbols

Ctrl	Controller
DA	Data Access
HDA	Historical Data Access
HMI	Human-Machine Interface
IEC	International Electrotechnical Commission
MES	Manufacturing Execution System
PLC	Programmable Logic Controller
SCADA	Supervisory Control And Data Acquisition
UA	Unified Architecture
XML	Extensible Markup Language

3.4 Conventions used in this document

3.4.1 Conventions for Node descriptions

Node definitions are specified using tables (see Table 2).

Attributes are defined by providing the *Attribute* name and a value, or a description of the value.

References are defined by providing the *ReferenceType* name, the *BrowseName* of the *TargetNode* and its *NodeClass*.

- If the *TargetNode* is a component of the *Node* being defined in the table, the *Attributes* of the composed *Node* are defined in the same row of the table.
- The *DataType* is only specified for *Variables*; "[<number>]" indicates a single-dimensional array, for multi-dimensional arrays the expression is repeated for each dimension (e.g. [2][3] for a two-dimensional array). For all arrays the *ArrayDimensions* is set as identified by <number> values. If no <number> is set, the corresponding dimension is set to 0, indicating an unknown size. If no number is provided at all the *ArrayDimensions* can be omitted. If no brackets are provided, it identifies a scalar *DataType* and the *ValueRank* is set to the corresponding value (see OPC 10000-3). In addition, *ArrayDimensions* is set to null or is omitted. If it can be Any or ScalarOrOneDimension, the value is put into "{<value>}", so either "{Any}" or "{ScalarOrOneDimension}" and the *ValueRank* is set to the corresponding value (see OPC 10000-3) and the *ArrayDimensions* is set to null or is omitted. Examples are given in Table 1.

Table 1 – Examples of DataTypes

Notation	Data-Type	Value-Rank	Array-Dimensions	Description
Int32	Int32	-1	omitted or null	A scalar Int32.
Int32[]	Int32	1	omitted or {0}	Single-dimensional array of Int32 with an unknown size.
Int32[][]	Int32	2	omitted or {0,0}	Two-dimensional array of Int32 with unknown sizes for both dimensions.
Int32[3][]	Int32	2	{3,0}	Two-dimensional array of Int32 with a size of 3 for the first dimension and an unknown size for the second dimension.
Int32[5][3]	Int32	2	{5,3}	Two-dimensional array of Int32 with a size of 5 for the first dimension and a size of 3 for the second dimension.
Int32{Any}	Int32	-2	omitted or null	An Int32 where it is unknown if it is scalar or array with any number of dimensions.
Int32{ScalarOrOneDimension}	Int32	-3	omitted or null	An Int32 where it is either a single-dimensional array or a scalar.

- The TypeDefinition is specified for *Objects* and *Variables*.
- The TypeDefinition column specifies a symbolic name for a *NodeId*, i.e. the specified *Node* points with a *HasTypeDefinition Reference* to the corresponding *Node*.
- The *ModellingRule* of the referenced component is provided by specifying the symbolic name of the rule in the *ModellingRule* column. In the *AddressSpace*, the *Node* shall use a *HasModellingRule Reference* to point to the corresponding *ModellingRule Object*.

If the *NodeId* of a *DataType* is provided, the symbolic name of the *Node* representing the *DataType* shall be used.

Nodes of all other *NodeClasses* cannot be defined in the same table; therefore, only the used *ReferenceType*, their *NodeClass* and their *BrowseName* are specified. A reference to another part of this document points to their definition.

Table 2 illustrates the table. If no components are provided, the DataType, TypeDefinition and ModellingRule columns may be omitted and only a Comment column is introduced to point to the *Node* definition.

Table 2 – Type Definition Table

Attribute	**Value**				
Attribute name	Attribute value. If it is an optional Attribute that is not set "--" will be used.				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
ReferenceType name	*NodeClass* of the Target*Node*.	*BrowseName* of the target *Node*. If the *Reference* is to be instantiated by the server, then the value of the target Node's BrowseName is "--".	*DataType* of the referenced *Node*, only applicable for *Variables*.	*TypeDefinition* of the referenced *Node*, only applicable for *Variables* and *Objects*.	Referenced *ModellingRule* of the referenced *Object*.
NOTE Notes referencing footnotes of the table content.					

Components of *Nodes* can be complex that is containing components by themselves. The *TypeDefinition*, *NodeClass*, *DataType* and *ModellingRule* can be derived from the type definitions, and the symbolic name can be created as defined in 3.4.3.1. Therefore, those containing components are not explicitly specified; they are implicitly specified by the type definitions.

3.4.2 NodeIds and BrowseNames

3.4.2.1 NodeIds

The *NodeIds* of all *Nodes* described in this standard are only symbolic names. Annex A defines the actual *NodeIds*.

The symbolic name of each *Node* defined in this specification is its *BrowseName*, or, when it is part of another *Node*, the *BrowseName* of the other *Node*, a ".", and the *BrowseName* of itself. In this case "part of" means that the whole has a *HasProperty* or *HasComponent Reference* to its part. Since all *Nodes* not being part of another *Node* have a unique name in this specification, the symbolic name is unique.

The NamespaceUri for all *NodeIds* defined in this specification is defined in Annex A. The NamespaceIndex for this NamespaceUri is vendor-specific and depends on the position of the NamespaceUri in the server namespace table.

Note that this specification not only defines concrete *Nodes*, but also requires that some *Nodes* shall be generated, for example one for each *Session* running on the *Server*. The *NodeIds* of those *Nodes* are *Server*-specific, including the namespace. But the NamespaceIndex of those *Nodes* cannot be the NamespaceIndex used for the *Nodes* defined in this specification, because they are not defined by this specification but generated by the *Server*.

3.4.2.2 BrowseNames

The text part of the *BrowseNames* for all *Nodes* defined in this specification is specified in the tables defining the *Nodes*. The NamespaceUri for all BrowseNames defined in this specification is defined in Annex A.

If the *BrowseName* is not defined by this specification, a namespace index prefix like '0:EngineeringUnits' or '2:DeviceRevision' is added to the *BrowseName*. This is typically necessary if a Property of another specification is overwritten or used in the OPC UA types defined in this specification. Table 42 provides a list of namespaces and their indexes as used in this specification.

3.4.3 Common Attributes

3.4.3.1 General

The *Attributes* of *Nodes*, their *DataTypes* and descriptions are defined in OPC 10000-3. Attributes not marked as optional are mandatory and shall be provided by a *Server*. The following tables define if the *Attribute* value is defined by this specification or if it is server-specific.

For all *Nodes* specified in this specification, the *Attributes* named in Table 3 shall be set as specified in the table.

Table 3 – Common Node Attributes

Attribute	Value
DisplayName	The *DisplayName* is a *LocalizedText*. Each server shall provide the *DisplayName* identical to the *BrowseName* of the *Node* for the LocaleId “en”. Whether the server provides translated names for other LocaleIds is server-specific.
Description	Optionally a server-specific description is provided.
NodeClass	Shall reflect the *NodeClass* of the *Node.*
NodeId	The *NodeId* is described by *BrowseNames* as defined in 3.4.2.1.
WriteMask	Optionally the *WriteMask Attribute* can be provided. If the *WriteMask Attribute* is provided, it shall set all non-server-specific *Attributes* to not writable. For example, the *Description Attribute* may be set to writable since a *Server* may provide a server-specific description for the *Node*. The *NodeId* shall not be writable, because it is defined for each *Node* in this specification.
UserWriteMask	Optionally the *UserWriteMask Attribute* can be provided. The same rules as for the *WriteMask Attribute* apply.
RolePermissions	Optionally server-specific role permissions can be provided.
UserRolePermissions	Optionally the role permissions of the current Session can be provided. The value is server-specific and depend on the *RolePermissions Attribute* (if provided) and the current *Session*.
AccessRestrictions	Optionally server-specific access restrictions can be provided.

3.4.3.2 Objects

For all *Objects* specified in this specification, the *Attributes* named in Table 4 shall be set as specified in the Table 4. The definitions for the *Attributes* can be found in OPC 10000-3.

Table 4 – Common Object Attributes

Attribute	Value
EventNotifier	Whether the *Node* can be used to subscribe to *Events* or not is server-specific.

3.4.3.3 Variables

For all *Variables* specified in this specification, the *Attributes* named in Table 5 shall be set as specified in the table. The definitions for the *Attributes* can be found in OPC 10000-3.

Table 5 – Common Variable Attributes

Attribute	Value
MinimumSamplingInterval	Optionally, a server-specific minimum sampling interval is provided.
AccessLevel	The access level for *Variables* used for type definitions is server-specific, for all other *Variables* defined in this specification, the access level shall allow reading; other settings are server-specific.
UserAccessLevel	The value for the *UserAccessLevel Attribute* is server-specific. It is assumed that all *Variables* can be accessed by at least one user.
Value	For *Variables* used as *InstanceDeclarations,* the value is server-specific; otherwise it shall represent the value described in the text.
ArrayDimensions	If the *ValueRank* does not identify an array of a specific dimension (i.e. *ValueRank* <= 0) the *ArrayDimensions* can either be set to null or the *Attribute* is missing. This behaviour is server-specific. If the *ValueRank* specifies an array of a specific dimension (i.e. *ValueRank* > 0) then the *ArrayDimensions Attribute* shall be specified in the table defining the *Variable*.
Historizing	The value for the *Historizing Attribute* is server-specific.
AccessLevelEx	If the *AccessLevelEx Attribute* is provided, it shall have the bits 8, 9, and 10 set to 0, meaning that read and write operations on an individual *Variable* are atomic, and arrays can be partly written.

3.4.3.4 VariableTypes

For all *VariableTypes* specified in this specification, the *Attributes* named in Table 6 shall be set as specified in the table. The definitions for the *Attributes* can be found in OPC 10000-3.

Table 6 – Common VariableType Attributes

Attributes	Value
Value	Optionally a server-specific default value can be provided.
ArrayDimensions	If the *ValueRank* does not identify an array of a specific dimension (i.e. *ValueRank* <= 0) the *ArrayDimensions* can either be set to null or the *Attribute* is missing. This behaviour is server-specific. If the *ValueRank* specifies an array of a specific dimension (i.e. *ValueRank* > 0) then the *ArrayDimensions Attribute* shall be specified in the table defining the *VariableType*.

3.4.3.5 Methods

For all *Methods* specified in this specification, the *Attributes* named in Table 7 shall be set as specified in the table. The definitions for the *Attributes* can be found in OPC 10000-3.

Table 7 – Common Method Attributes

Attributes	Value
Executable	All *Methods* defined in this specification shall be executable (*Executable Attribute* set to “True”), unless it is defined differently in the *Method* definition.
UserExecutable	The value of the *UserExecutable Attribute* is server-specific. It is assumed that all *Methods* can be executed by at least one user.

3.4.4 Reference to IEC 61131-3 Definitions

Referenced key words in this document like VAR_GLOBAL are defined in the IEC 61131-3 specification. See table delimiters and key words in the IEC 61131-3 for a complete list.

4 General information to IEC 61131-3 and OPC UA

4.1 Introduction to IEC 61131-3

IEC 61131-3 is the first real endeavour to standardize programming languages for industrial automation. With its worldwide support, it is independent of any single company.

IEC 61131-3 is the third part of the IEC 61131 family. This consists of: (1) General information, (2) Equipment requirements and tests, (3) Programming languages, (4) User guidelines, (5) Communications, (6) Safety, (7) Fuzzy control programming, and (8) Guidelines for the application and implementation of programming languages.

IEC 61131-3 basically describes the Common Elements and Programming Languages.

4.1.1 Common Elements

4.1.1.1 Data Typing

Within the common elements, the data types are defined. Data typing prevents errors in an early stage. It is used to define the type of any parameter used. This avoids for instance dividing a Date by an Integer.

Common data types are Boolean, Integer, Real and Byte and Word, but also Date, Time_of_Day and String. Based on these, one can define own personal data types, known as derived data types. In this way one can define an analogue input channel as data type, and re-use this over and over again.

4.1.1.2 Ctrl Variables

Ctrl Variables are only assigned to explicit hardware addresses (e.g. input and outputs) in *Ctrl Configurations*, *Ctrl Resources* or *Ctrl Programs*. In this way a high level of hardware independency is created, supporting the reusability of the software.

The scopes of the *Ctrl Variables* are normally limited to the organization unit in which they are declared, e.g. local. This means that their names can be reused in other parts without any conflict, eliminating another source of errors. If the *Ctrl Variables* should have global scope, they have to be declared as such (VAR_GLOBAL). *Ctrl Variables* can be assigned an initial value at start up and cold restart, in order to have the right setting.

4.1.1.3 Ctrl Configuration, Ctrl Resources and Ctrl Tasks

These elements are integrated within the software model as defined in the standard (see below).

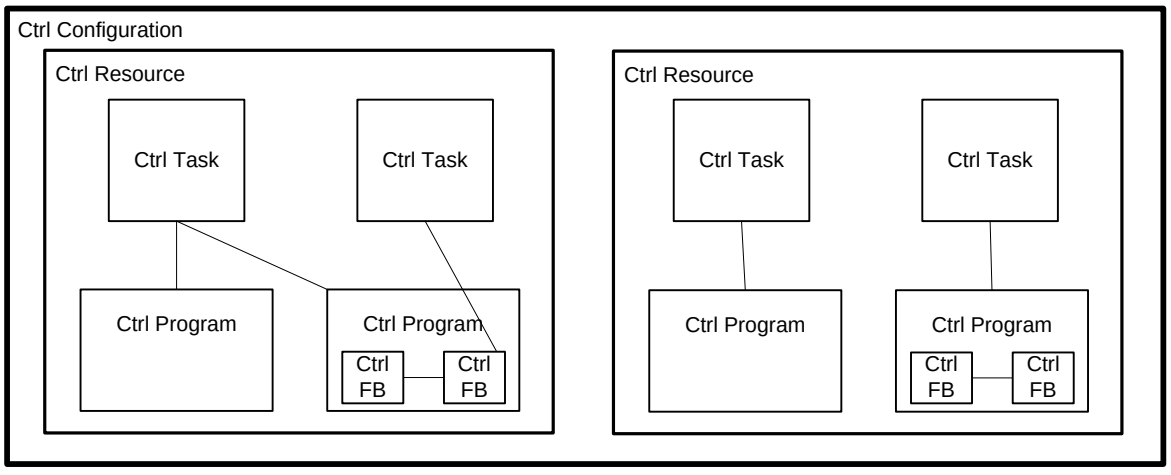

Figure 1 – Software Model

At the highest level, the entire software required to solve a particular control problem can be formulated as a *Ctrl Configuration*. A *Ctrl Configuration* is specific to a particular type of control system, including the arrangement of the hardware, i.e. processing resources, memory addresses for I/O channels and system capabilities.

Within a *Ctrl Configuration* one can define one or more *Ctrl Resources*. One can look at a *Ctrl Resource* as a processing facility that is able to execute *Ctrl Programs*.

Within a *Ctrl Resource*, one or more *Ctrl Tasks* can be defined. *Ctrl Tasks* control the execution of a set of *Ctrl Programs* and/or *Ctrl Function Blocks*. These can either be executed periodically or upon the occurrence of a specified trigger, such as the change of a *Ctrl Variable*.

Ctrl Programs are built from a number of different software elements written in any of the defined programming languages. Typically, a *Ctrl Program* consists of a network of *Ctrl Function*s and *Ctrl Function Blocks*, which are able to exchange data. *Ctrl Functions* and *Ctrl Function Blocks* are the basic building blocks, containing a data structure and an algorithm.

4.1.1.4 Ctrl Program Organization Units

Within IEC 61131-3, the *Ctrl Programs*, *Ctrl Function Blocks* and *Ctrl Functions* are called *Ctrl Program Organization Units*, POUs.

4.1.1.5 Ctrl Functions

IEC 61131-3 has defined standard *Ctrl Functions* and user defined *Ctrl Functions*. Standard *Ctrl Functions* are for instance ADD(addition), ABS(absolute), SQRT, SIN(sinus) and COS(cosinus). User defined *Ctrl Functions*, once defined, can be used over and over again.

4.1.1.6 Ctrl Function Blocks

Ctrl Function Blocks are the equivalent to integrated circuits, representing a specialized control function. They contain data as well as the algorithm, so they can keep track of the past (which is one of the differences w.r.t. *Ctrl Functions*). They have a well-defined interface and hidden internals, like an integrated circuit or black box. In this way they give a clear separation between different levels of programmers, or maintenance people.

A temperature control loop, or PID, is an excellent example of a *Ctrl Function Block*. Once defined, it can be used over and over again, in the same *Ctrl Program*, different *Ctrl Programs*, or even different projects. This makes them highly re-usable.

Ctrl Function Blocks can be written in any of the languages, and in most cases even in “C”. This way they can be defined by the user. Derived *Ctrl Function Blocks* are based on the standard defined *Ctrl Function Blocks*, but also completely new, customized *Ctrl Function Blocks* are possible within the standard: it just provides the framework.

The interfaces of *Ctrl Functions* and *Ctrl Function Blocks* are described in the same way.

4.1.1.7 Sequential Function Chart

Within the standard Sequential Function Chart (SFC) is defined as a structuring tool. This means that syntax and semantics have been defined, leaving no room for dialects. The language consists of a textual and a graphical version.

4.1.1.8 Ctrl Programs

A *Ctrl Program* is a network of *Ctrl Functions* and *Ctrl Function Blocks*. A *Ctrl Program* can be written in any of the defined programming languages.

4.1.2 Programming Languages

Within the standard four programming languages are defined. This means that their syntax and semantics have been defined, leaving no room for dialects. The languages consist of textual and graphical versions:

Instruction List, IL (textual)

Structured Text, ST (textual)

Ladder Diagram, LD (graphical)

Function Block Diagram, FBD (graphical)

4.2 Introduction to OPC Unified Architecture

4.2.1 What is OPC UA?

OPC UA is an open and royalty free set of standards designed as a universal communication protocol. While there are numerous communication solutions available, OPC UA has key advantages:

A state of art security model (see OPC 10000-2).

A fault tolerant communication protocol.

An information modelling framework that allows application developers to represent their data in a way that makes sense to them.

OPC UA has a broad scope which delivers for economies of scale for application developers. This means that a larger number of high-quality applications at a reasonable cost are available. When combined with semantic models such as IEC61131-3, OPC UA makes it easier for end users to access data via generic commercial applications.

The OPC UA model is scalable from small devices to ERP systems. OPC UA *Servers* process information locally and then provides that data in a consistent format to any application requesting data - ERP, MES, PMS, Maintenance Systems, HMI, Smartphone or a standard Browser, for examples. For a more complete overview see OPC 10000-1.

4.2.2 Basics of OPC UA

As an open standard, OPC UA is based on standard internet technologies, like TCP/IP, HTTP, Web Sockets.

As an extensible standard, OPC UA provides a set of *Services* (see OPC 10000-4) and a basic information model framework. This framework provides an easy manner for creating and exposing vendor defined information in a standard way. More importantly all OPC UA *Clients* are expected to be able to discover and use vendor-defined information. This means OPC UA users can benefit from the economies of scale that come with generic visualization and historian applications. This specification is an example of an OPC UA *Information Model* designed to meet the needs of developers and users.

OPC UA *Clients* can be any consumer of data from another device on the network to browser based thin clients and ERP systems. The full scope of OPC UA applications is shown in Figure 2.

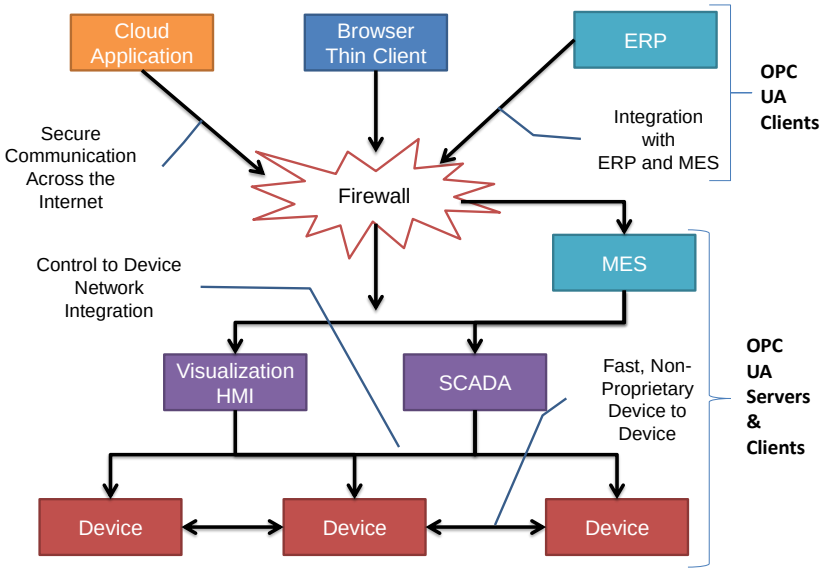

Figure 2 – The Scope of OPC UA within an Enterprise

OPC UA provides a robust and reliable communication infrastructure having mechanisms for handling lost messages, failover, heartbeat, etc. With its binary encoded data, it offers a high-performing data exchange solution. Security is built into OPC UA as security requirements become more and more important especially since environments are connected to the office network or the internet and attackers are starting to focus on automation systems.

4.2.3 Information modelling in OPC UA

4.2.3.1 Concepts

OPC UA provides a framework that can be used to represent complex information as *Objects* in an *AddressSpace* which can be accessed with standard services. These *Objects* consist of *Nodes* connected by *References*. Different classes of *Nodes* convey different semantics. For example, a *Variable Node* represents a value that can be read or written. The *Variable Node* has an associated *DataType* that can define the actual value, such as a string, float, structure etc. It can also describe the *Variable* value as a variant. A *Method Node* represents a function that can be called. Every *Node* has a number of *Attributes* including a unique identifier called a *NodeId* and non-localized name called as *BrowseName*. An *Object* representing a 'Reservation' is shown in Figure 3.

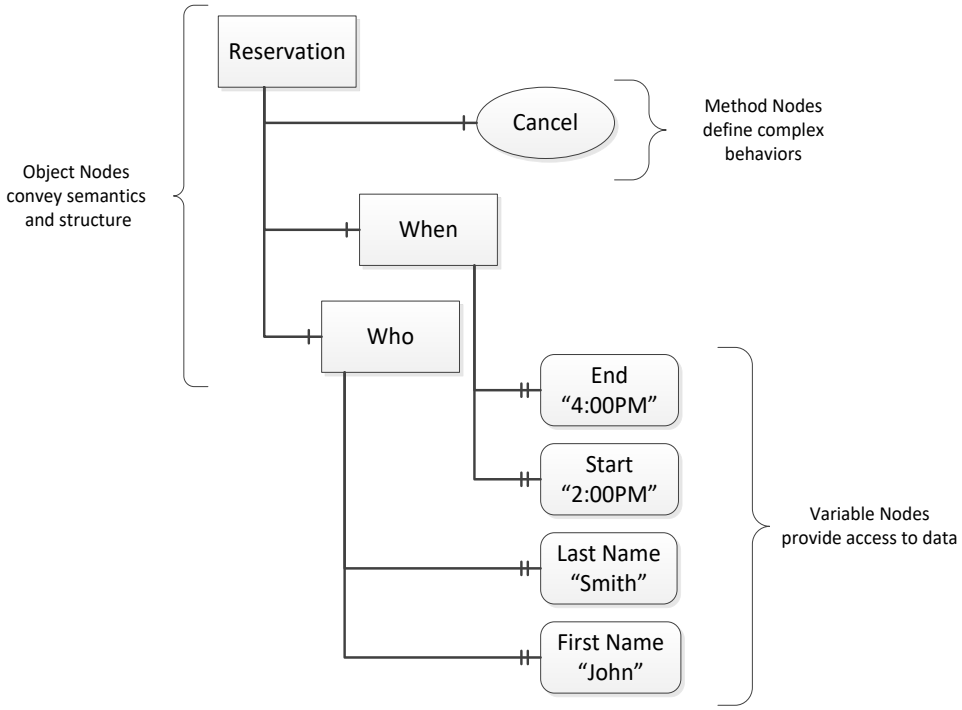

Figure 3 – A Basic Object in an OPC UA Address Space

Object and *Variable Nodes* represent instances and they always reference a *TypeDefinition* (*ObjectType* or *VariableType*) *Node* which describes their semantics and structure. Figure 4 illustrates the relationship between an instance and its *TypeDefinition*.

The type *Nodes* are templates that define all of the children that can be present in an instance of the type. In the example in Figure 4 the PersonType *ObjectType* defines two children: First Name and Last Name. All instances of PersonType are expected to have the same children with the same *BrowseNames*. Within a type the *BrowseNames* uniquely identify the children. This means *Client* applications can be designed to search for children based on the *BrowseNames* from the type instead of *NodeIds*. This eliminates the need for manual reconfiguration of systems if a *Client* uses types that multiple *Servers* implement.

OPC UA also supports the concept of sub-typing. This allows a modeller to take an existing type and extend it. There are rules regarding sub-typing defined in OPC 10000-3, but in general they allow the extension of a given type or the restriction of a *DataType*. For example, the modeller may decide that the existing *ObjectType* in some cases needs an additional *Variable*. The modeller can create a subtype of the *ObjectType* and add the *Variable*. A *Client* that is expecting the parent type can treat the new type as if it was of the parent type. Regarding *DataTypes*, subtypes can only restrict. If a *Variable* is defined to have a numeric value, a sub type could restrict it to a float.

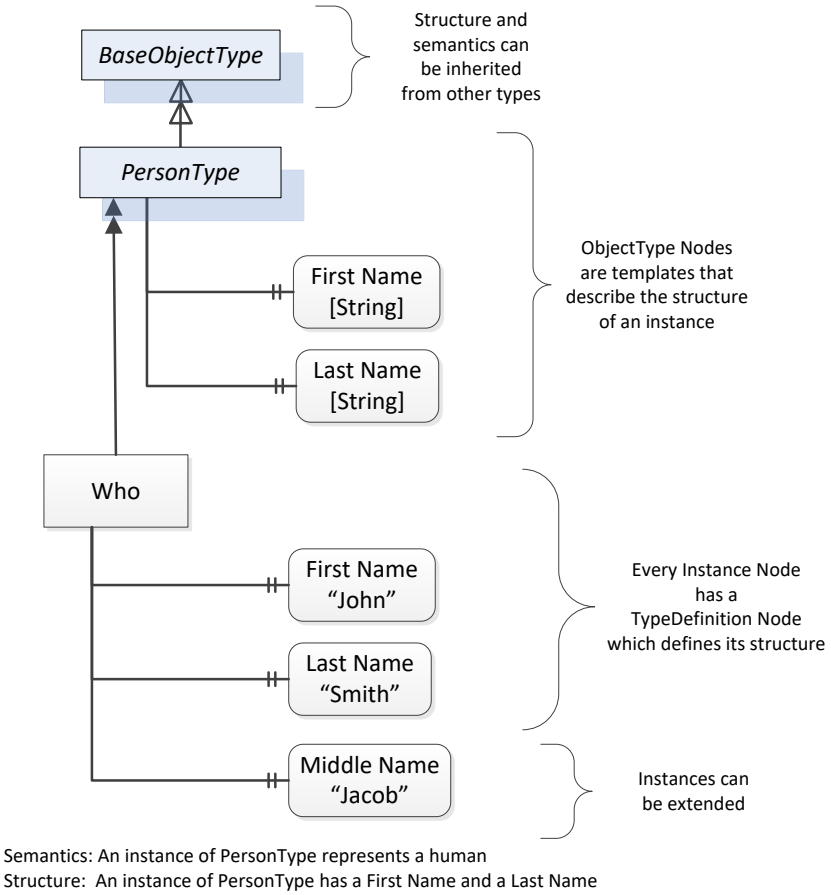

Figure 4 – The Relationship between Type Definitions and Instances

References allow *Nodes* to be connected in ways that describe their relationships. All *References* have a *ReferenceType* that specifies the semantics of the relationship. *References* can be hierarchical or non-hierarchical. Hierarchical references are used to create the structure of *Objects* and *Variables*. Non-hierarchical are used to create arbitrary associations. Applications can define their own *ReferenceType* by creating subtypes of an existing *ReferenceType*. Subtypes inherit the semantics of the parent but may add additional restrictions. Figure 5 depicts several *References*, connecting different *Objects*.

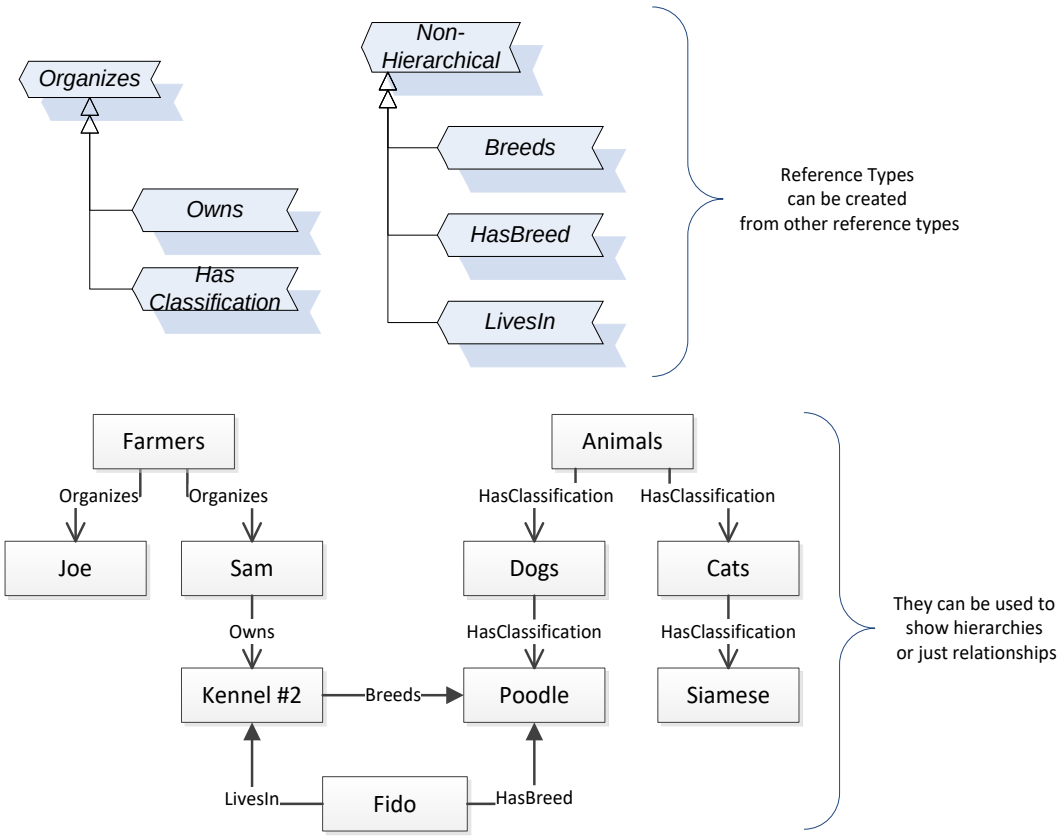

Figure 5 – Examples of References between Objects

The figures above use a notation that was developed for the OPC UA specification. The notation is summarized in Figure 6. UML representations can also be used; however, the OPC UA notation is less ambiguous because there is a direct mapping from the elements in the figures to *Nodes* in the *AddressSpace* of an OPC UA *Server*.

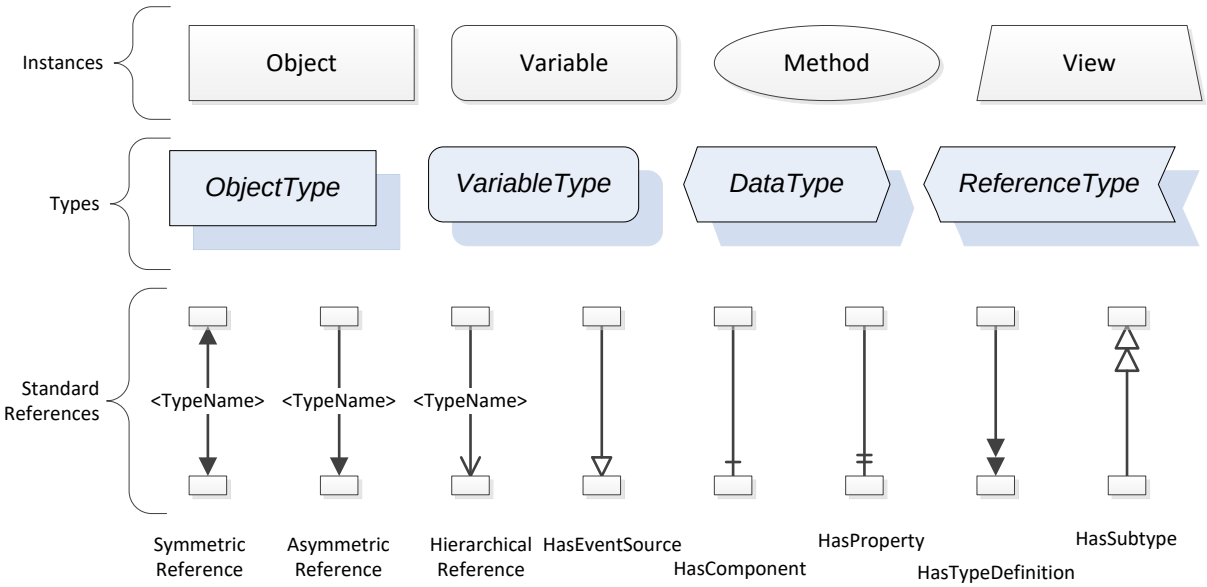

Figure 6 – The OPC UA Information Model Notation

A complete description of the different types of Nodes and References can be found in OPC 10000-3 and the base structure is described in OPC 10000-5.

OPC UA specification defines a very wide range of functionality in its basic information model. It is not expected that all *Clients* or *Servers* support all functionality in the OPC UA specifications. OPC UA includes the concept of *Profiles*, which segment the functionality into testable certifiable units. This allows the definition of functional subsets (that are expected to be implemented) within a companion specification. The *Profiles* do not restrict functionality, but generate requirements for a minimum set of functionalities (see OPC 10000-7)

4.2.3.2 Namespaces

OPC UA allows information from many different sources to be combined into a single coherent *AddressSpace*. Namespaces are used to make this possible by eliminating naming and id conflicts between information from different sources. Namespaces in OPC UA have a globally unique string called a NamespaceUri and a locally unique integer called a NamespaceIndex. The NamespaceIndex is only unique within the context of a *Session* between an OPC UA *Client* and an OPC UA *Server*. The *Services* defined for OPC UA use the NamespaceIndex to specify the Namespace for qualified values.

There are two types of values in OPC UA that are qualified with Namespaces: NodeIds and QualifiedNames. NodeIds are globally unique identifiers for *Nodes*. This means the same *Node* with the same NodeId can appear in many *Servers*. This, in turn, means Clients can have built in knowledge of some *Nodes*. OPC UA *Information Models* generally define globally unique *NodeIds* for the *TypeDefinitions* defined by the *Information Model*.

QualifiedNames are non-localized names qualified with a Namespace. They are used for the *BrowseNames* of *Nodes* and allow the same names to be used by different information models without conflict. *TypeDefinitions* are not allowed to have children with duplicate *BrowseNames*; however, instances do not have that restriction.

4.2.3.3 Companion Specifications

An OPC UA companion specification for an industry specific vertical market describes an *Information Model* by defining *ObjectTypes*, *VariableTypes*, *DataTypes* and *ReferenceTypes* that represent the concepts used in the vertical market, and potentially also well-defined Objects as entry points into the AddressSpace.

4.2.3.4 Introduction to OPC UA Devices

The OPC 10000-100 specification is an extension of the overall OPC Unified Architecture specification series and defines the information model associated with *Devices*. The model is intended to provide a unified view of *Devices* irrespective of the underlying *Device* protocols. Controllers are physical or logical *Devices* and the *Devices* model is therefore used as base for the IEC 61131-3 information model.

The *Devices* information model specifies different *ObjectTypes* and procedures used to represent *Devices* and related components like the communication infrastructure in an OPC UA *Address Space*. The main use cases are *Device* configuration and diagnostic, but it allows a general and standardized way for any kind of application to access *Device* related information. The following examples illustrate the concepts used in this specification. See OPC 10000-100 for the complete definition of the *Devices* information model.

Figure 7 shows an example for a temperature controller represented as *Device Object*. The component *ParameterSet* contains all *Variables* describing the *Device*. The component *MethodSet* contains all *Methods* provided by the *Device*. Both components are inherited from the *TopologyElementType* which is the root *Object* type of the *Device Object* type hierarchy. Objects of the type *FunctionalGroupType* are used to group the *Parameters* and Methods of the Device into logical groups. The *FunctionalGroupType* and the grouping concept are defined in OPC 10000-100 but the groups are Device type specific i.e. the groups ProcessData and Configuration are defined by the *TemperatureControllerType* in this example.

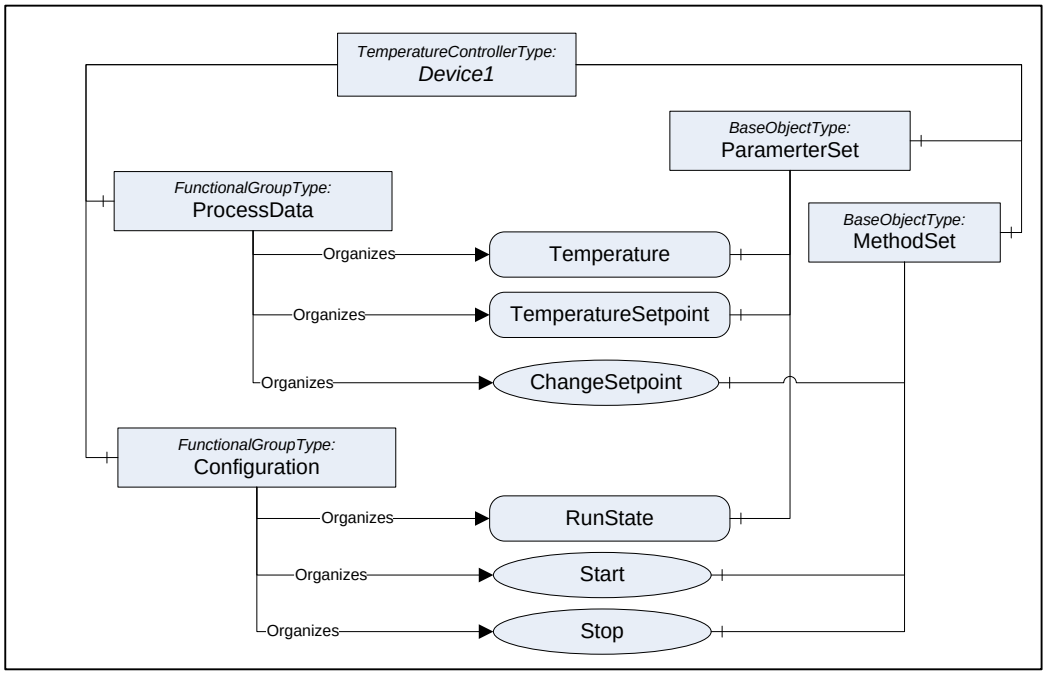

Figure 7 – OPC UA Devices Example

Another OPC 10000-100 concept used in this specification is described in Figure 8. The *ConfigurableObjectType* is used to provide a way to group subcomponents of a Device and to indicate which types of subcomponents can be instantiated. The allowed types are referenced from the *SupportedTypes* folder. This information can be used by configuration clients to allow a user to select the type to instantiate as subcomponent of the *Device*.

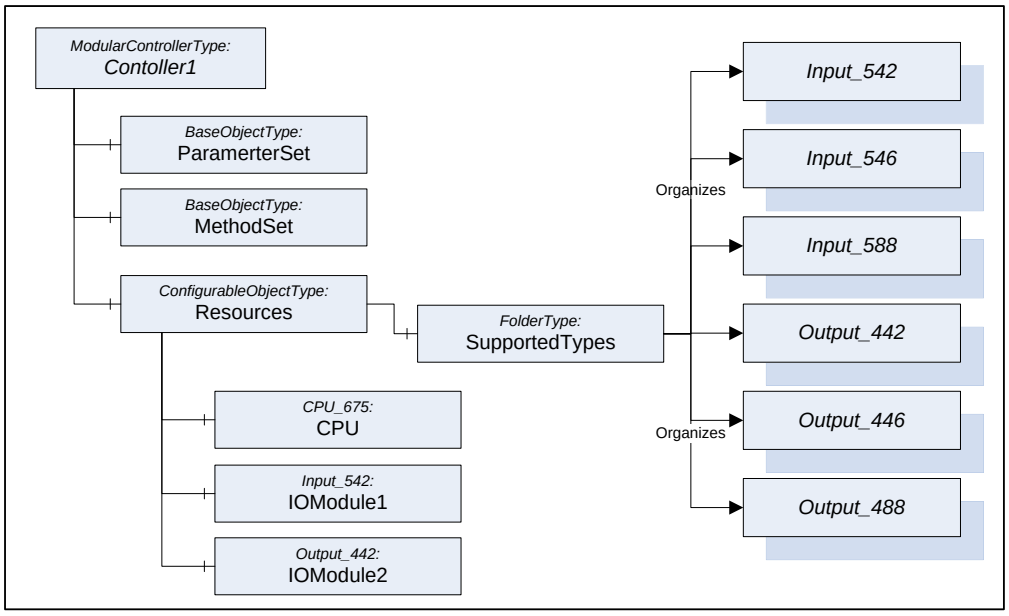

Figure 8 – OPC UA Devices Example

The *SupportedTypes* Folder can contain different subsets of *ObjectTypes* for different instances of the *ModularControlerType* depending on their current configuration since the list contains only types that can be instantiated for the current configuration. The example expects that only one CPU can be used in the *ModularControlerType* and this CPU is already configured. The *SupportedTypes* Folder on the *ModularControlerType* contains all possible types including CPU types that can be used in the *ModularControlerType*.

4.3 Introductory Example

A simple example shall be used to explain how the above introduced OPC UA concepts are used to represent elements in an OPC UA Server.

According to IEC 61131-3, *Ctrl Function Blocks* consist of a name, *Ctrl Variables* with associated data types (input, output, internal), and a body containing the algorithm to be executed. These data are represented by OPC UA *ObjectTypes* derived from the *ObjectType CtrlFunctionBlockType* (see Figure 11).

To start, IEC 61131-3 requires a *Ctrl Function Block* type declaration. Here an integer up-counter is used which follows the standard counter *Ctrl Function Block*. CTU_INT contains three input *Ctrl Variables* (CU – counter up, R – reset, PV – primary value), one local *Ctrl Variable* (PVmax) and two output *Ctrl Variables* (Q, CV – counter value) with their respective data types. Furthermore, CTU_INT has a body containing the algorithm to do the actual counting. The formal declaration of CTU_INT using the Structured Text programming language is shown in Figure 9.

```
FUNCTION_BLOCK CTU_INT


VAR_INPUT
  CU: BOOL;
  R: BOOL;
  PV: INT;
END_VAR


VAR
  PVmax: INT := 32767;
END_VAR


VAR_OUTPUT
  Q: BOOL;
  CV: INT;
END_VAR


  IF R THEN
    CV := 0;
  ELSIF CU AND (CV < PVmax) THEN
    CV := CV + 1;
  END_IF ;
  Q := (CV >= PV);


END_FUNCTION_BLOCK
```

Figure 9 – *Ctrl Function Block* CTU_INT declaration

The OPC UA representation of CTU_INT is shown in Figure 11. The *ObjectType* CTU_INT is a subtype of the *ObjectType CtrlFunctionBlockType*. Its components are defined by instance declaration and referenced by *HasInputVar*, *HasLocalVar*, and *HasOutputVar References*.

After declaration of CTU_INT it is instantiated twice (MyCounter, MyCounter2) and used within a *Ctrl Program* MyTestProgram shown in Figure 10. Signal and Signal 2 are counted, the *Ctrl Function Block* output *Ctrl Variables* are transferred to some temporary *Ctrl Variables* but are not further processed in this example.

```
PROGRAM MyTestProgram


VAR_INPUT
  Signal: BOOL;
  Signal2: BOOL;
END_VAR


VAR
  MyCounter: CTU_INT;
  MyCounter2: CTU_INT;
END_VAR


VAR_TEMP
  QTemp: BOOL;
  CVTemp: INT;
END_VAR


  MyCounter(CU := Signal, R := FALSE, PV := 24);


  QTemp := MyCounter.Q;
  CVTemp := MyCounter.CV;



  MyCounter2(CU := Signal2, R := FALSE, PV := 19);


  QTemp := MyCounter2.Q;
  CVTemp := MyCounter2.CV;


END_PROGRAM
```

Figure 10 – *Ctrl Function Block* MyCounter / MyCounter2 instantiation and usage

The OPC UA representation of the *Objects* MyCounter and MyCounter2 is shown in Figure 11. The *Objects* are instances of the *ObjectType* CTU_INT which is indicated by the *HasTypeDefinition References*. The example specific *ObjectType* CTU_INT is derived from the *ObjectType*

CtrlFunctionBlockType which is indicated by the *HasSubType Reference*. Current values at a certain point in time are provided by the instances, e. g. the current counter value of MyCounter equals 11. The *Ctrl Program* MyTestProgram is not represented in the figure.

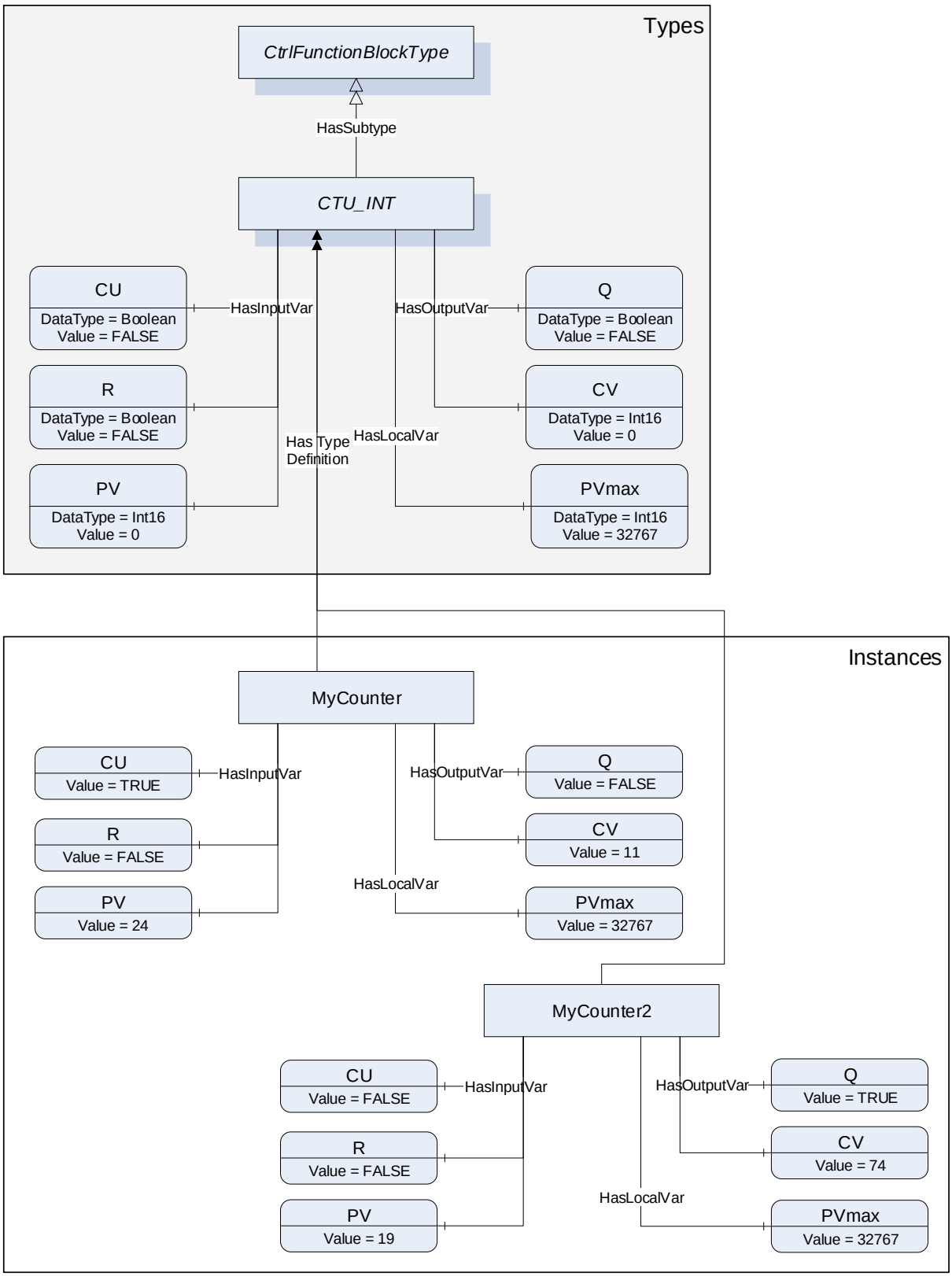

Figure 11 – Introductory Example – OPC UA representation

5 Use cases

The following use cases illustrate the usage of the information model. Not all necessary *Objects* must be realized within a concrete OPC UA Server.

Observation

Observation comprises reading and monitoring data of *Ctrl Configurations, Ctrl Resources, Ctrl Tasks, Ctrl Programs, Ctrl Function Blocks, Ctrl Variables*, and their *ObjectTypes* represented in the OPC UA Server.

Example 1: In a brewery, several tanks of the same type are operated. They are controlled by the same *Ctrl Function Block* which is instantiated in the *Controller* once for each tank. For developing the visualization it is useful to create first a template for operating a tank, which is based on the tank *CtrlFunctionBlockType* provided by the OPC UA server. Then this template can be instantiated and connected to the *Ctrl Function Block* instances within the OPC UA server as often as required (reuse).

Example 2: In a brewery, the number of bottles produced in the current shift shall be presented on a visualization panel. The bottles are counted by the *Controller* and the result provided as an output *Ctrl Variable* of a *Ctrl Function Block*. The visualization panel subscribes to the corresponding *Variable* in the OPC UA server, gets the current number of bottles delivered each time it is changing, and presents it to the user.

Operation

Operation inherits the functionality of observation and extends it.

Operation comprises writing data of *Ctrl Variables* represented in the OPC UA Server and execution control of *Ctrl Programs* and *Ctrl Function Blocks* using *Ctrl Tasks* represented in the OPC UA Server.

Example: In a brewery, several recipes are used to produce different kinds of beer. To choose the recipe for the next batch, the number of that recipe is written from an HMI to an input *Ctrl Variable* of a *Ctrl Function Block* via a corresponding *Variable* in the OPC UA server. After this, the batch is started using a *Ctrl Task* in the OPC UA server which triggers the *Ctrl Function Block*.

Engineering (Programming / Maintenance)

Engineering inherits the functionality of operation and extends it.

Engineering comprises writing of *Ctrl Configurations, Ctrl Resources, Ctrl Tasks, Ctrl Programs, Ctrl Function Blocks, Ctrl Functions, Ctrl Variables*, and their *ObjectTypes* into the OPC UA Server.

Example: The *Ctrl Program* of a machine tool shall be updated via remote access (internet). This download is done using programming software by writing the corresponding *Ctrl Program ObjectType* into the OPC UA server while observing strict security (and safety) regulations.

Service

Service inherits the functionality of engineering and extends it.

Service comprises the carrying out of service specific functions, e. g. reading / writing of special data and firmware updates.

The following Figure 12 shows the use case diagram.

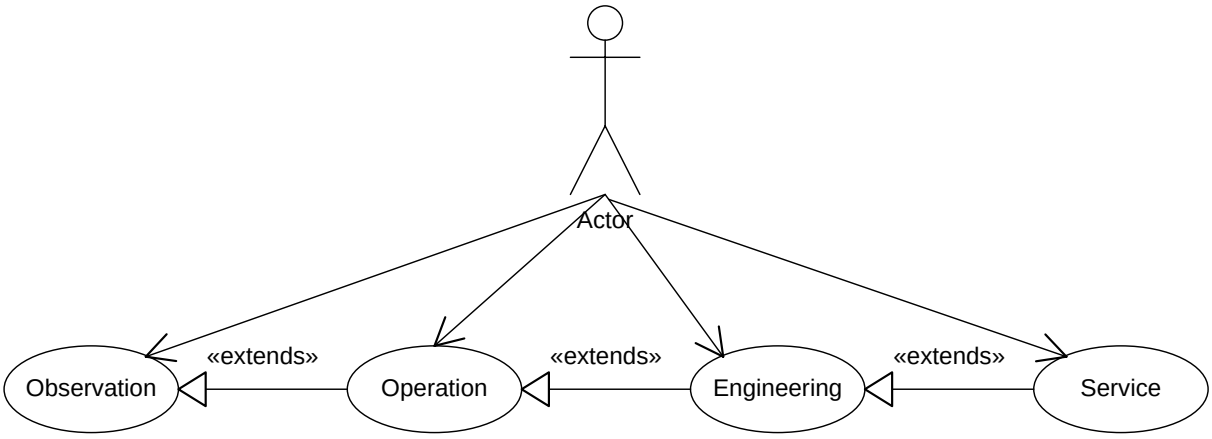

Figure 12 – Use case diagram

6 IEC 61131-3 Information Model overview

Figure 13 depicts the main *ObjectTypes* of this specification and their relationships. The drawing is not intended to be complete. For the sake of simplicity only a few components and relations were captured so as to give a rough idea of the overall structure of the *IEC 61131-3 Information Model*.

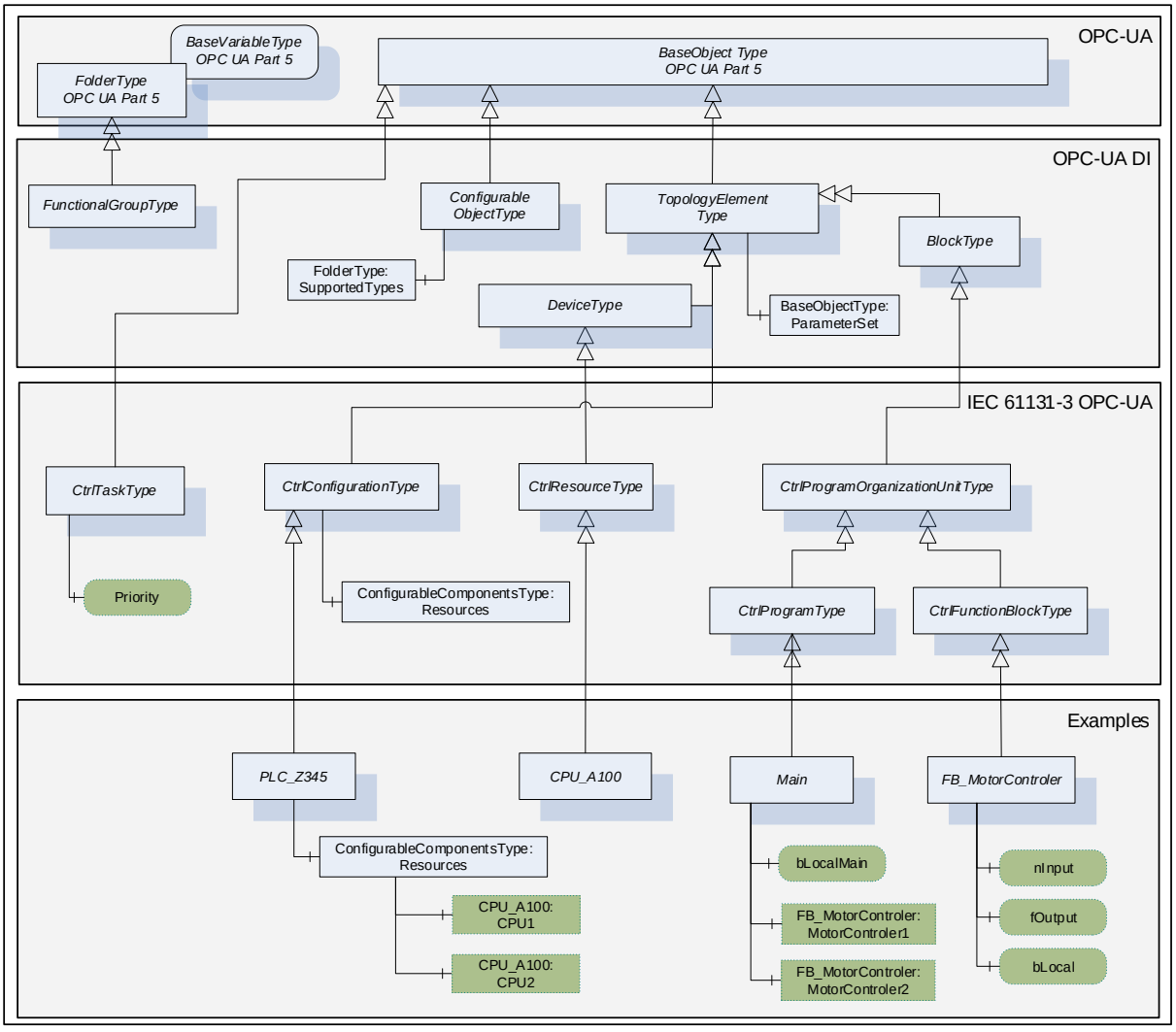

Figure 13 – *OPC UA IEC 61131-3 ObjectTypes* Overview

The boxes in this drawing show the *ObjectTypes* used in this specification as well as some elements from other specifications that help understand the overall context. The upper grey box shows the OPC UA core *ObjectTypes* from which the OPC UA Device Integration Types are derived. The Device Integration model and its Types in the second level are used as base for the *IEC 61131-3 ObjectTypes*. The grey box in the third level shows the *IEC 61131-3 ObjectTypes* that this specification introduces. The components of those *ObjectTypes* are illustrated only in an abstract way in this overall picture. The grey box in the lowest level represents examples of sub types defined by vendors or *Controller* programmers.

Typically, the components of an *ObjectType* are fixed and can be extended by subtyping. However, since each *Object* of an *ObjectType* can be extended with additional components, this specification allows extending the standard *ObjectTypes* defined in this document with additional components. Thereby, it is possible to express the additional information in the type definition that would already be contained in each *Object*. Some *ObjectTypes* already provide entry points for server specific extensions. However, it is not allowed to restrict the components of the standard *ObjectTypes* defined in this specification. An example of extending the *ObjectTypes* is putting the standard *Property NodeVersion* defined in OPC 10000-3 into the *BaseObjectType*, stating that each *Object* of the server will provide a *NodeVersion*.

It is not the objective to map all IEC 61131-3 constraints to the OPC UA Information Model, but to define an OPC UA Information Model which is capable to hold at least all possible data of one or more IEC 61131-3 compliant *Ctrl Configurations*.

A *Ctrl Configuration* compliant to IEC 61131-3 represents the special case of a complete engineered *Controller* with an OPC UA server providing access to all data of one or more IEC 61131-3 compliant *Ctrl Configurations*. In general, an OPC UA server may provide incomplete *Ctrl Configurations*, e.g. during the engineering process or because not all data shall be accessed from outside.

Examples for *Object* and *Variable* instances of the vendor or controller programmer specific types are shown in Figure 14. The *Root* and the *Objects Folder* are *Nodes* defined by OPC 10000-5. The *Objects Folder* is the main entry point for *Object* instances.

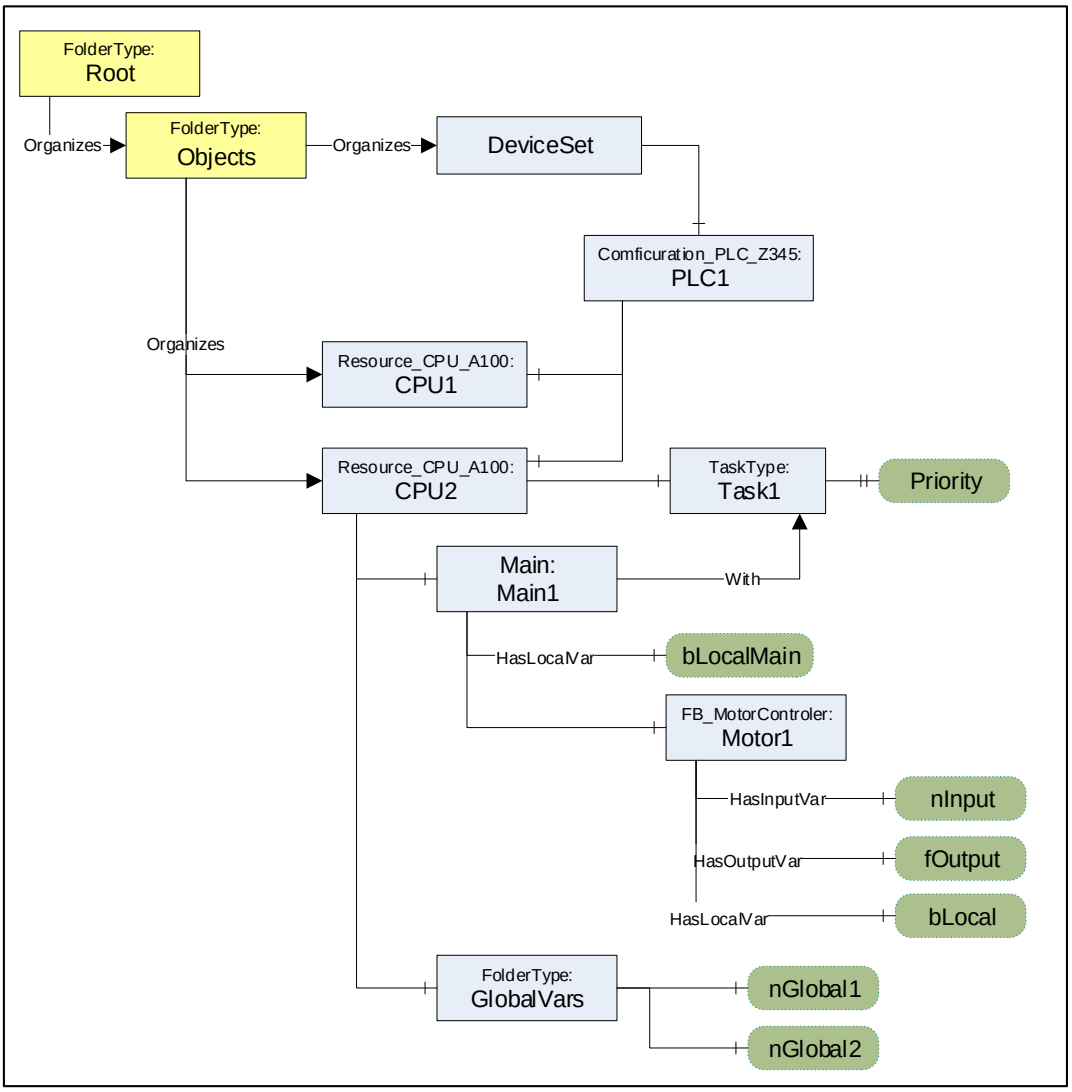

Figure 14 – OPC UA IEC 61131-3 Object Instance Example

7 OPC UA ObjectTypes

7.1 CtrlConfigurationType ObjectType Definition

7.1.1 Overview

This *ObjectType* defines the representation of a *Ctrl Configuration* of a programmable *Controller* system in an OPC UA *Address Space*. It introduces *Objects* to group *Ctrl Resources* and different types of *Ctrl Variables*. The *CtrlConfigurationType* is derived from the *TopologyElementType* defined in OPC 10000-100. Figure 15 shows the *CtrlConfigurationType*. It is formally defined in Table 8. The dark grey nodes in the figure are examples and are not part of the *ObjectType* definition.

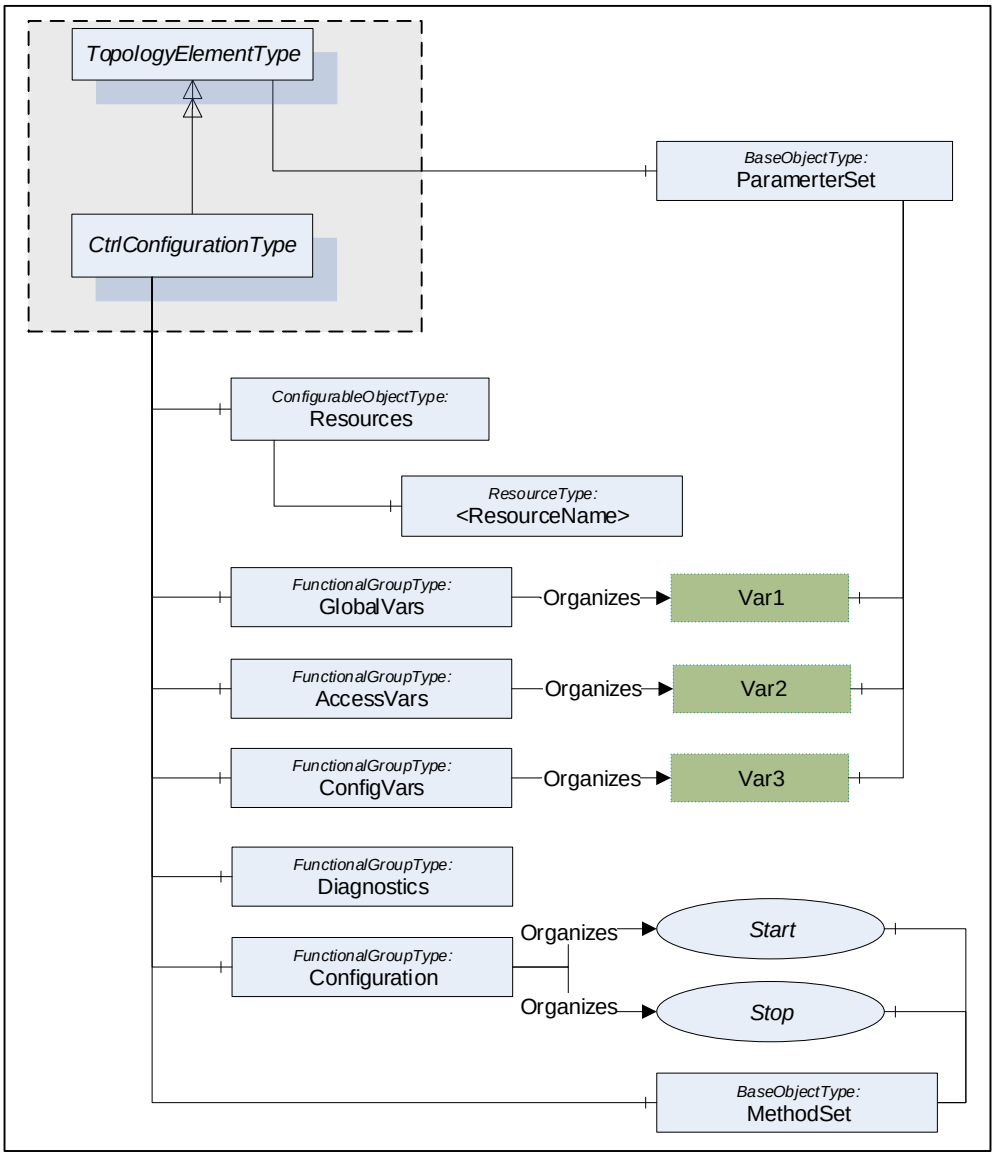

Figure 15 – *CtrlConfigurationType* Overview

The *Ctrl Configuration* type is formally defined in Table 8.

Table 8 – *CtrlConfigurationType* Definition

Attribute	**Value**				
BrowseName	CtrlConfigurationType				
IsAbstract	False				
References	**Node Class**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *TopologyElementType* defined in OPC 10000-100, i.e. inheriting the InstanceDeclarations of that Node.					
HasComponent	Object	2:MethodSet		BaseObjectType	Optional
HasComponent	Object	Resources		ConfigurableObjectType	Mandatory
HasComponent	Object	GlobalVars		FunctionalGroupType	Optional
HasComponent	Object	AccessVars		FunctionalGroupType	Optional
HasComponent	Object	ConfigVars		FunctionalGroupType	Optional
HasComponent	Object	Configuration		FunctionalGroupType	Optional
HasComponent	Object	Status		FunctionalGroupType	Optional

The *CtrlConfigurationType ObjectType* is a concrete type and can be used directly. It is recommended to create subtypes for vendor or user specific configurations.

A concrete Ctrl Configuration type or instance may have *ParameterSet*, *Parameters* and *FunctionalGroup*s as defined for the *TopologyElementType* in OPC 10000-100.

The *MethodSet Object* is defined by the *TopologyElementType* and is overwritten in the *CtrlConfigurationType* to add the *HasComponent References* to the *Methods* defined for the *CtrlConfigurationType*.

The *Object Resources* is used to group *Ctrl Resources* that are part of the *Ctrl Configuration*. It uses the concept of configurable *Objects* defined OPC 10000-100. It contains *Objects* of the type *CtrlResourceType* representing a *Ctrl Resource* and a Folder with possible *Ctrl Resource* types that can be instantiated in the *Ctrl Configuration*. For a complete configuration at least one resource is necessary from an IEC 61131-3 point of view but not necessary from an OPC UA point of view. Temporary, incomplete configurations are allowed, e.g. during a configuration process.

The *FunctionalGroup GlobalVars* contains the corresponding list of *Ctrl Variables* declared by the key word VAR_GLOBAL.

The *FunctionalGroup AccessVars* contains the corresponding list of *Ctrl Variables* declared by the key word VAR_ACCESS.

The *FunctionalGroup ConfigVars* contains the corresponding list of *Ctrl Variables* declared by the key word VAR_CONFIG.

The *FunctionalGroup Configuration* contains configuration *Variables* and *Methods* like start and stop.

The *FunctionalGroup Status* contains diagnostic and status information like system variables, status variables or diagnostic codes.

Starting a *Ctrl Configuration* causes the initialization of global *Ctrl Variables* and the start of all *Ctrl Resources*. Stopping a *Ctrl Configuration* stops all *Ctrl Resources*.

7.1.2 Resources components

The configurable *Object Resources* of the *CtrlConfigurationType* is formally defined in Table 9.

Table 9 – Components of the *Resources Object*

Attribute	**Value**			
BrowseName	Resources			
References	**NodeClass**	**BrowseName**	**TypeDefinition**	**ModellingRule**
HasComponent	Object	<ResourceName>	*CtrlResourceType*	OptionalPlaceholder

7.1.3 MethodSet components

The *Methods* available as parts of the *CtrlConfigurationType* are formally defined in Table 10.

Table 10 – Components of the *CtrlConfigurationType MethodSet*

Attribute	**Value**			
BrowseName	MethodSet			
References	**Node Class**	**BrowseName**	**Description**	**Modelling Rule**
Configuration FunctionalGroup The following components are also referenced from the *FunctionalGroup Configuration* using *Organizes References*.				
HasComponent	Method	Start	This Method is used to start a *Ctrl Configuration*. Only the browse name is defined for this *Method*. The *Method* parameters are vendor specific.	Optional
HasComponent	Method	Stop	This Method is used to stop a *Ctrl Configuration*. Only the browse name is defined for this *Method*. The *Method* parameters are vendor specific.	Optional

7.2 CtrlResourceType ObjectType Definition

7.2.1 Overview

This *ObjectType* defines the representation of a *Crtl Resources* of a programmable *Controller* system in an OPC UA *Address Space*. It introduces *Objects* to group Configuration and Diagnostic capabilities,

GlobalVars and *Ctrl Programs* executed under the control of *Tasks*. The *CtrlResourceType* is derived from the *DeviceType* defined in OPC 10000-100. Figure 16 shows the *CtrlResourceType*. It is formally defined in Table 11. The dark grey node in the figure is an example and is not part of the *ObjectType* definition.

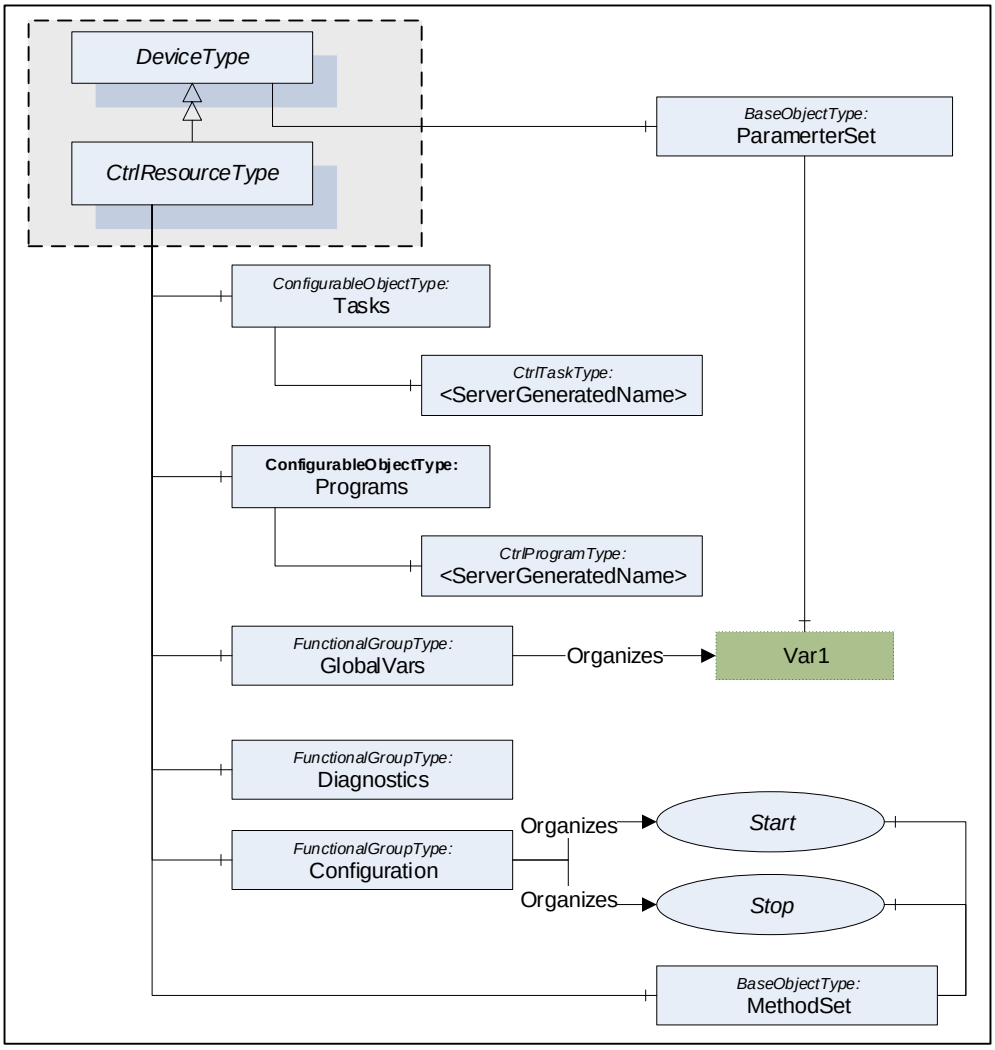

Figure 16 – *CtrlResourceType* Overview

The *Ctrl Resource ObjectType* is formally defined in Table 11.

Table 11 – *CtrlResourceType* Definition

Attribute	**Value**				
BrowseName	CtrlResourceType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *DeviceType* defined in OPC 10000-100, i.e. inheriting the InstanceDeclarations of that Node.					
HasComponent	Object	2:MethodSet		BaseObjectType	Optional
HasComponent	Object	Tasks		ConfigurableObjectType	Mandatory
HasComponent	Object	Programs		ConfigurableObjectType	Mandatory
HasComponent	Object	GlobalVars		FunctionalGroupType	Optional
HasComponent	Object	Configuration		FunctionalGroupType	Optional
HasComponent	Object	Status		FunctionalGroupType	Optional

The *CtrlResourceType ObjectType* is a concrete type and can be used directly. It is recommended to create subtypes for vendor or user specific resources.

A concrete Ctrl Resource type or instance may have *ParameterSet*, *Parameters* and *FunctionalGroup*s as defined for the *TopologyElementType* in OPC 10000-100.

The *MethodSet Object* is defined by the *TopologyElementType* and is overwritten in the *CtrlResourceType* to add the *HasComponent References* to the *Methods* defined for the *CtrlResourceType*.

The *Object Tasks* is used to group *Ctrl Tasks* that are part of the *Ctrl Resource*. It uses the concept of configurable *Objects* defined OPC 10000-100. It contains *Objects* of the type *CtrlTaskType* representing a *Ctrl Resource* and a Folder with possible *Ctrl Task* types that can be instantiated in the *Ctrl Resource*.

The configurable *Object Programms* is used to group *Ctrl Programs* that are part of the *Ctrl Resource*. It contains *Objects* of the type *CtrlTaskType* representing a *Ctrl Resource* and a Folder with possible *Ctrl Program* types that can be instantiated in the *Ctrl Resource*.

The *FunctionalGroup GlobalVars* contains the corresponding list of *GlobalVars* that may be accessed in the programmable *Controller* system within the scope of the *Ctrl Resource*.

The *FunctionalGroup Configuration* contains configuration *Variables* and *Methods* like start and stop.

The *FunctionalGroup Status* contains diagnostic and status information like system variables, system events or diagnostic codes.

7.2.2 Tasks components

The configurable *Object Tasks* of the *CtrlResourceType* is formally defined in Table 12.

Table 12 – Components of the *Tasks Object*

Attribute	**Value**			
BrowseName	Tasks			
References	**NodeClass**	**BrowseName**	**TypeDefinition**	**ModellingRule**
HasComponent	Object	<TaskName>	*CtrlTaskType*	OptionalPlaceholder

7.2.3 Programs components

The configurable *Object Programs* of the *CtrlResourceType* is formally defined in Table 13.

Table 13 – Components of the *Programs Object*

Attribute	**Value**			
BrowseName	Programs			
References	**NodeClass**	**BrowseName**	**TypeDefinition**	**ModellingRule**
HasComponent	Object	<ProgramName>	*CtrlProgramType*	OptionalPlaceholder

7.2.4 MethodSet components

The *Methods* available as parts of the *CtrlResourceType* are formally defined in Table 14.

Table 14 – Components of the *CtrlResourceType MethodSet*

Attribute	**Value**			
BrowseName	MethodSet			
References	**NodeClass**	**BrowseName**	**Description**	**ModellingRule**
Configuration FunctionalGroup The following components are also referenced from the *FunctionalGroup Configuration* using *Organizes References*.				
HasComponent	Method	Start	This Method is used to start a *Ctrl Resource*. Only the browse name is defined for this *Method*. The *Method* parameters are vendor specific.	Optional
HasComponent	Method	Stop	This Method is used to stop a *Ctrl Resource*. Only the browse name is defined for this *Method*. The *Method* parameters are vendor specific.	Optional

7.3 CtrlProgramOrganizationUnitType ObjectType Definition

This *ObjectType* defines the representation of a *Ctrl Program Organization Unit* of a programmable *Controller* system in an OPC UA *Address Space*. It defines how components of the *Ctrl Program Organization Unit like Variables* and *Ctrl Function Blocks* are represented. The

CtrlProgramOrganizationUnitType is derived from the *BlockType* defined in OPC 10000-100. Figure 17 shows the *CtrlProgramOrganizationUnitType*. It is formally defined in Table 15.

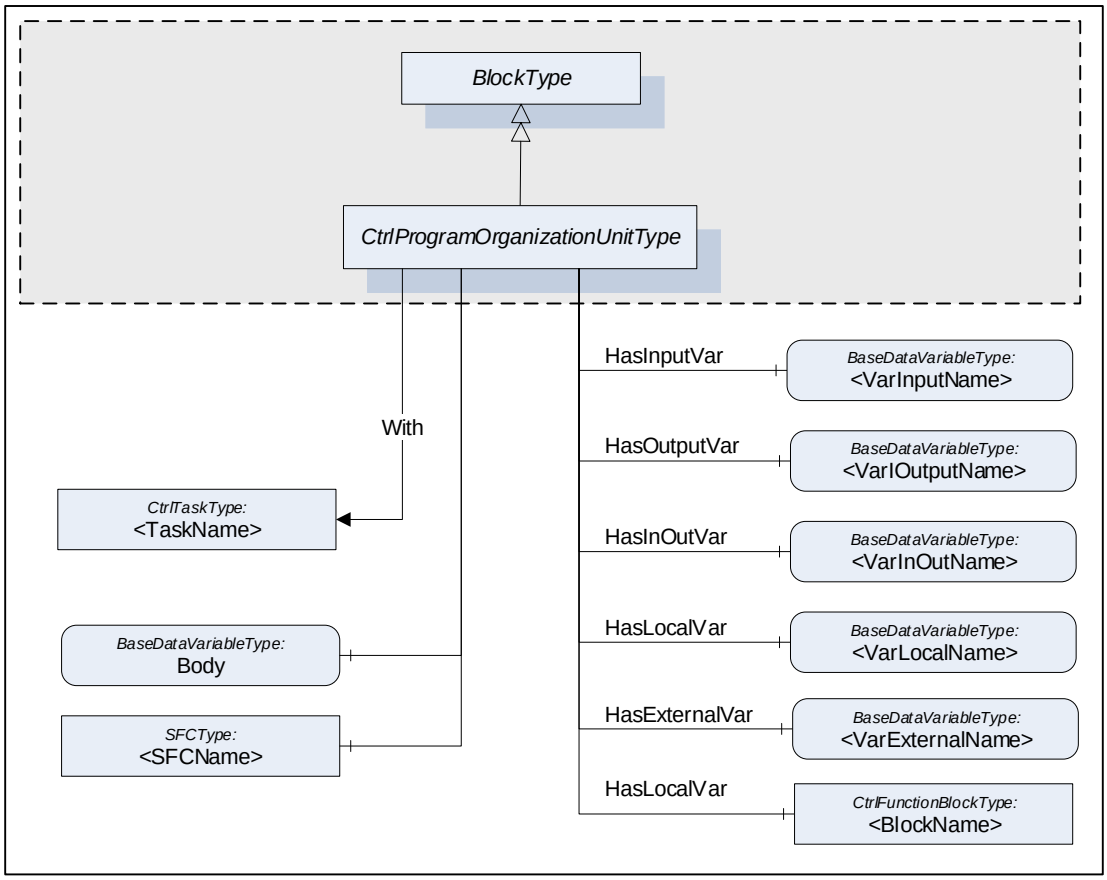

Figure 17 – *CtrlProgramOrganizationUnitType* Overview

The *Ctrl Program Organization Unit ObjectType* is formally defined in Table 15.

Table 15 – *CtrlProgramOrganizationUnitType* Definition

Attribute	**Value**				
BrowseName	CtrlProgramOrganizationUnitType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *BlockType* defined in **OPC 10000-100**, i.e. inheriting the InstanceDeclarations of that Node.					
HasSubtype	ObjectType	CtrlProgramType	Defined in Clause 7.4		
HasSubtype	ObjectType	CtrlFunctionBlock Type	Defined in Clause 7.5		
With	Object	<TaskName>		CtrlTaskType	OptionalPlaceholder
HasInputVar	Variable	<VarInputName>	BaseDataType	BaseDataVariableType	OptionalPlaceholder
HasOutputVar	Variable	<VarOutputName>	BaseDataType	BaseDataVariableType	OptionalPlaceholder
HasInOutVar	Variable	<VarInOutName>	BaseDataType	BaseDataVariableType	OptionalPlaceholder
HasLocalVar	Variable	<VarLocalName>	BaseDataType	BaseDataVariableType	OptionalPlaceholder
HasExternalVar	Variable	<VarExternalName>	BaseDataType	BaseDataVariableType	OptionalPlaceholder
HasLocalVar	Object	<BlockName>		CtrlFunctionBlockType	OptionalPlaceholder
HasComponent	Variable	Body	XmlElement	BaseDataVariableType	Optional
HasComponent	Object	<SFCName>		SFCType	OptionalPlaceholder

The *CtrlProgramOrganizationUnitType ObjectType* is abstract. It is the common base type for all *Ctrl Program Organization Unit* specific types.

The *With Reference* defined in 8.7 is used to reference the Ctrl Task that is used to execute the *Ctrl Program Organization Unit*.

Variables declared for a *Ctrl Program Organization Unit* type are referenced with different subtypes of the *HasComponent Reference*. The used *Reference* type depends on the IEC 61131-3 variable declaration keywords. The characteristics of the Variables like data type and access rights and their mapping from IEC 61131-3 information and key words is defined in chapter 9. The name of the Variable depends on the *Variable* name in the *Ctrl Program Organization Unit*.

Variables declared with the key word VAR_INPUT are referenced with *HasInputVar* defined in 8.2.

Variables declared with the key word VAR_OUTPUT are referenced with *HasOutputVar* defined in 8.3.

Variables declared with the key word VAR_IN_OUT are referenced with *HasInOutVar* defined in 8.4.

Variables declared with the key word VAR are referenced with *HasLocalVar* defined in 8.5.

Variables declared with the key word VAR_EXTERNAL are referenced with *HasExternalVar* defined in 8.6.

Ctrl Function Blocks declared with the key word VAR are referenced with *HasLocalVar* defined in 8.5. The name of the *Object* depends on the name of the block in the *Ctrl Program Organization Unit*.

The *Variable Body* contains the body of the *Ctrl Program Organisation Unit* as *XmlElement*.

Sequential function charts (SFC) declared in the *Ctrl Program Organisation Unit* are represented as Objects of the type *SFCType* defined in chapter 7.7.

7.4 CtrlProgramType ObjectType Definition

This *ObjectType* defines the representation of a *Ctrl Program* of a programmable *Controller* system in an OPC UA *Address Space*. It is derived from *CtrlProgramOrganizationUnitType* and introduces additional Variables in addition to the components of the base type. Figure 18 shows the *CtrlProgramType*. It is formally defined in Table 16.

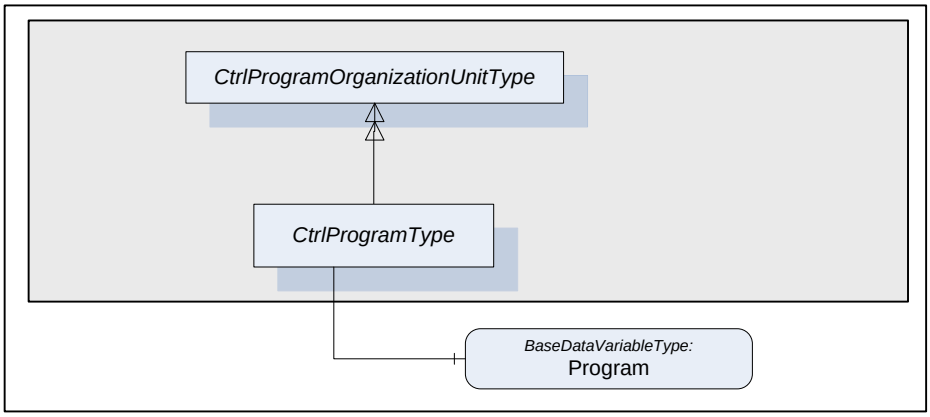

Figure 18 – *CtrlProgramType* Overview

The *Ctrl Program ObjectType* is formally defined in Table 16.

Table 16 – *CtrlProgramType* Definition

Attribute	**Value**				
BrowseName	CtrlProgramType				
IsAbstract	True				
References	**Node Class**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the CtrlProgramOrganizationUnitType defined in 7.3, i.e. inheriting the InstanceDeclarations of that Node.					
HasComponent	Variable	Program	Structure	BaseDataVariableType	Optional

The *CtrlProgramType ObjectType* is abstract. There will be no instances of a *CtrlProgramType* itself, but there will be instances of subtypes of this type like instances of vendor or user specific *Ctrl Programs*.

The *Program Variable* component contains the complete *Ctrl Program* data in a complex *Variable*.

7.5 CtrlFunctionBlockType ObjectType Definition

This *ObjectType* defines the representation of a *Ctrl Function Blocks* of a programmable *Controller* system in an OPC UA *Address Space*. It is derived from *CtrlProgramOrganizationUnitType* and introduces *Ctrl Function Block* specific components in addition to the components of the base type. Figure 19 shows the *CtrlFunctionBlockType*. It is formally defined in Table 17.

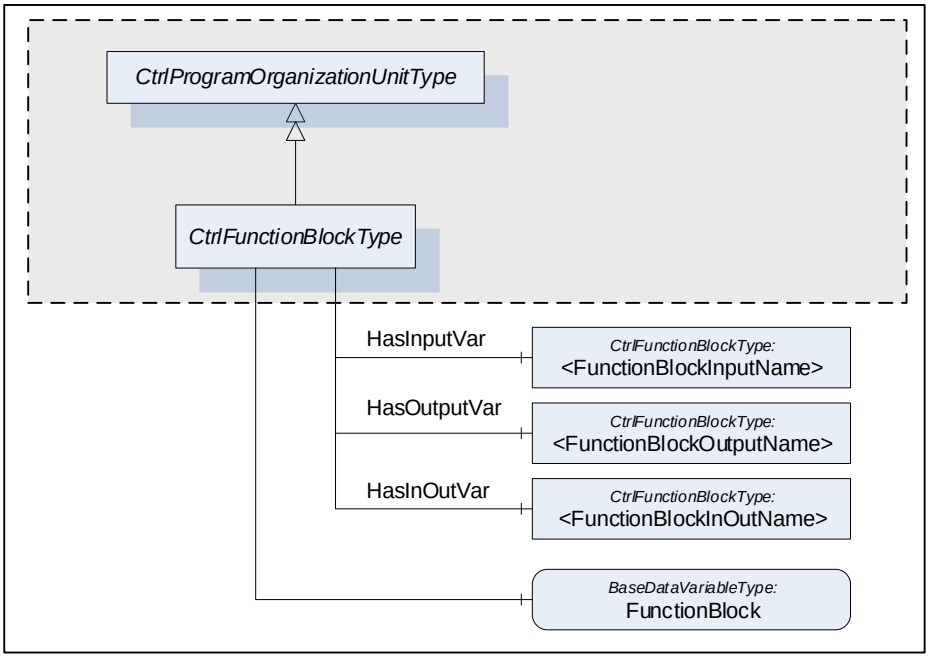

Figure 19 – *CtrlFunctionBlockType* Overview

The *CtrlFunctionBlock ObjectType* is formally defined in Table 17.

Table 17 – *CtrlFunctionBlockType* Definition

Attribute	**Value**			
BrowseName	CtrlFunctionBlockType			
IsAbstract	True			
References	**NodeClass**	**BrowseName**	**TypeDefinition**	**ModellingRule**
Inherit the *Properties* and components of the CtrlProgramOrganizationUnitType				
HasInputVar	Object	<FunctionBlockInputName>	CtrlFunctionBlockType	OptionalPlaceholder
HasOutputVar	Object	<FunctionBlockOutputName>	CtrlFunctionBlockType	OptionalPlaceholder
HasInOutVar	Object	<FunctionBlockInOutName>	CtrlFunctionBlockType	OptionalPlaceholder
HasComponent	Variable	FunctionBlock	BaseDataVariableType	Optional

The *CtrlFunctionBlockType ObjectType* is abstract. There will be no instances of a *CtrlFunctionBlockType* itself, but there will be instances of subtypes of this type like instances of vendor or user specific *Ctrl Function Blocks*.

Ctrl Function Block instances declared for a *Ctrl Function Block* type are referenced with different subtypes of the *HasComponent Reference*. The used *Reference* type depends on the IEC 61131-3 declaration keywords. The name of the *Object* depends on the name of the block in the *Ctrl Function Block*.

Ctrl Function Block instances declared with the key word VAR_INPUT are referenced with *HasInputVar* defined in 8.2.

Ctrl Function Block instances declared with the key word VAR_OUTPUT are referenced with *HasOutputVar* defined in 8.3.

Ctrl Function Block instances declared with the key word VAR_IN_OUT are referenced with *HasInOutVar* defined in 8.4.

The *FunctionBlock Variable* component contains the complete *Ctrl Function Block* data in a complex *Variable*. The *DisplayName* for the Variable is FunctionBlock.

7.6 CtrlTaskType ObjectType Definition

This *ObjectType* defines the representation of a *Ctrl Task* of a programmable *Controller* system in an OPC UA *Address Space*. It introduces Properties providing information about the *Ctrl Task*. Figure 20 shows the *CtrlTaskType*. It is formally defined in Table 18.

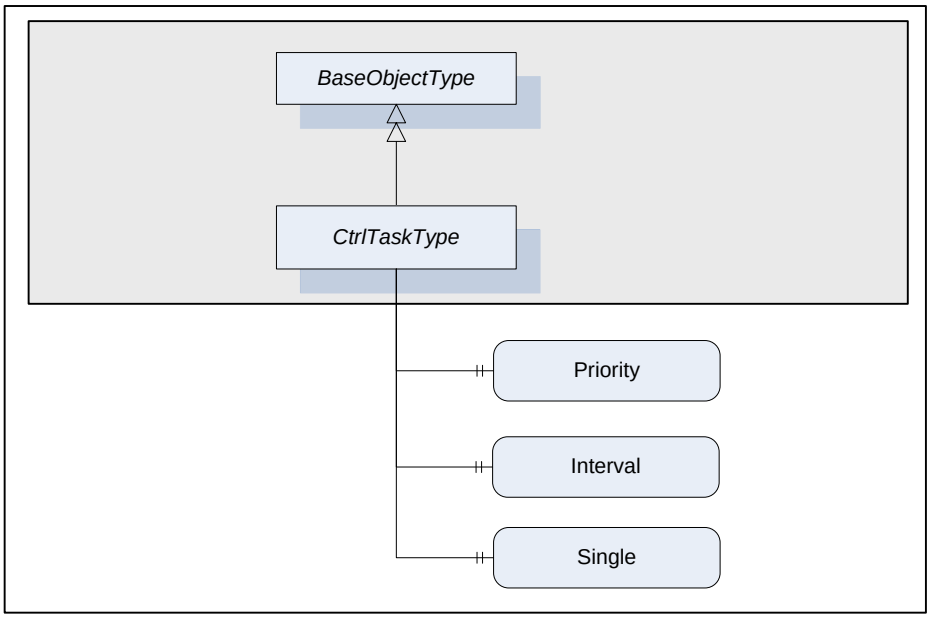

Figure 20 – *CtrlTaskType* Overview

The *Ctrl Task ObjectType* is formally defined in Table 18.

Table 18 – *CtrlTaskType* Definition

Attribute	**Value**				
BrowseName	CtrlTaskType				
IsAbstract	False				
References	**Node Class**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *BaseObjectType* defined in OPC 10000-5, i.e. inheriting the InstanceDeclarations of that Node.					
HasProperty	Variable	Priority	UInt32	PropertyType	Mandatory
HasProperty	Variable	Interval	String	PropertyType	Optional
HasProperty	Variable	Single	String	PropertyType	Optional

The *Priority Property* indicates the scheduling priority of the associated *Ctrl Program Organization Unit*.

The *Interval Property* indicates the periodical scheduling of the associated *Ctrl Program Organization Unit* at the specified interval.

The *Single Property* indicates a single scheduling of the associated *Ctrl Program Organization Unit* at each rising edge.

7.7 SFCType ObjectType Definition

The SFC *ObjectType* is formally defined in Table 19. This type is a container for Sequential Function Chart (SFC) related information. The representation of this information is vendor specific. Future versions of this specification may define standard representations.

Table 19 – SFC*Type* Definition

Attribute	Value				
BrowseName	SFCType				
IsAbstract	False				
References	**Node Class**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *BaseObjectType* defined in OPC 10000-5, i.e. inheriting the InstanceDeclarations of that Node.					

8 Reference Types

8.1 General

Figure 21 depicts the main *ReferenceTypes* of this specification and their relationship.

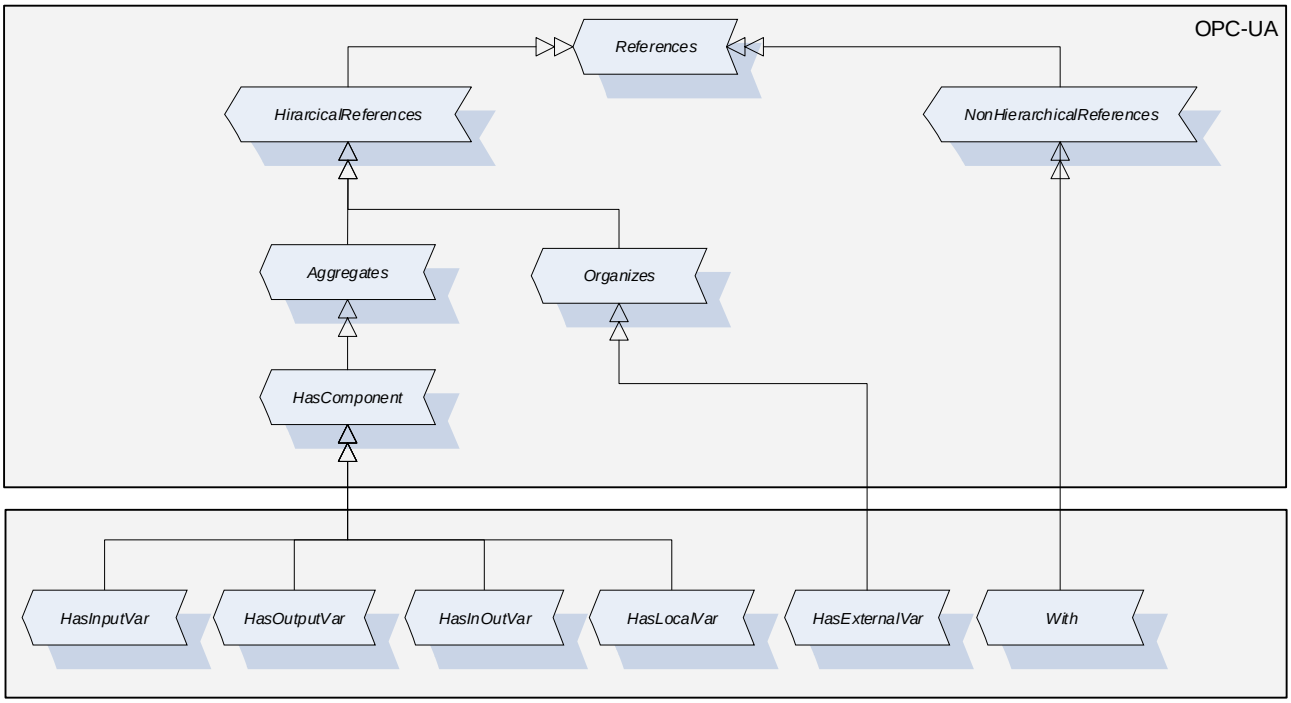

Figure 21 – Reference Types Overview

The upper grey box shows the OPC UA core *ReferenceTypes* from which the *IEC 61131-3 ReferenceTypes* are derived. The grey box in the second level shows the *IEC 61131-3 ReferenceTypes* that this specification introduces.

8.2 HasInputVar

This *ReferenceType* is a subtype of the *HasComponent ReferenceType* defined in OPC 10000-5. Its representation in the *AddressSpace* is specified in Table 20.

Table 20 – HasInputVar ReferenceType

Attributes	Value		
BrowseName	HasInputVar		
InverseName	InputVarOf		
Symmetric	False		
IsAbstract	False		
References	**NodeClass**	**BrowseName**	**Comment**
Subtype of HasComponent Reference*Type* defined in OPC 10000-5			

The *HasInputVar ReferenceType* is a concrete *ReferenceType* and can be used directly.

The semantic of this *ReferenceType* is to reference components of a *Ctrl Program Organization Unit* declared with the key word VAR_INPUT.

The *SourceNode* of *References* of this type shall be a subtype of *CtrlProgramOrganizationUnitType* or an instance of one of its subtypes.

The *TargetNode* of this *ReferenceType* shall be a *Variable* or an *Object* of the *ObjectType CtrlFunctionBlockType* or one of its subtypes.

8.3 HasOutputVar

This *ReferenceType* is a subtype of the *HasComponent ReferenceType* defined in OPC 10000-5. Its representation in the *AddressSpace* is specified in Table 21.

Table 21 – HasOutputVar ReferenceType

Attributes	Value		
BrowseName	HasOutputVar		
InverseName	OutputVarOf		
Symmetric	False		
IsAbstract	False		
References	**NodeClass**	**BrowseName**	**Comment**
Subtype of HasComponent Reference*Type* defined in OPC 10000-5			

The *HasOutputVar ReferenceType* is a concrete *ReferenceType* and can be used directly.

The semantic of this *ReferenceType* is to reference components of a *Ctrl Program Organization Unit* declared with the key word VAR_OUTPUT.

The *SourceNode* of *References* of this type shall be a subtype of *CtrlProgramOrganizationUnitType* or an instance of one of its subtypes.

The *TargetNode* of this *ReferenceType* shall be a *Variable* or an *Object* of the *ObjectType CtrlFunctionBlockType* or one of its subtypes.

8.4 HasInOutVar

This *ReferenceType* is a subtype of the *HasComponent ReferenceType* defined in OPC 10000-5. Its representation in the *AddressSpace* is specified in Table 22.

Table 22 – HasInOutVar ReferenceType

Attributes	Value		
BrowseName	HasInOutVar		
InverseName	InOutVarOf		
Symmetric	False		
IsAbstract	False		
References	**NodeClass**	**BrowseName**	**Comment**
Subtype of HasComponent Reference*Type* defined in OPC 10000-5			

The *HasInOutVar ReferenceType* is a concrete *ReferenceType* and can be used directly.

The semantic of this *ReferenceType* is to reference components of a *Ctrl Program Organization Unit* declared with the key word VAR_INOUTPUT.

The *SourceNode* of *References* of this type shall be a subtype of *CtrlProgramOrganizationUnitType* or an instance of one of its subtypes.

The *TargetNode* of this *ReferenceType* shall be a *Variable* or an *Object* of the ObjectType *CtrlFunctionBlockType* or one of its subtypes.

8.5 HasLocalVar

This *ReferenceType* is a subtype of the *HasComponent ReferenceType* defined in OPC 10000-5. Its representation in the *AddressSpace* is specified in Table 23.

Table 23 – HasLocalVar ReferenceType

Attributes	**Value**		
BrowseName	HasLocalVar		
InverseName	LocalVarOf		
Symmetric	False		
IsAbstract	False		
References	**NodeClass**	**BrowseName**	**Comment**
Subtype of HasComponent Reference*Type* defined in OPC 10000-5			

The *HasLocalVar ReferenceType* is a concrete *ReferenceType* and can be used directly.

The semantic of this *ReferenceType* is to reference components of a *Ctrl Program Organization Unit* declared with the key word VAR.

The *SourceNode* of *References* of this type shall be a subtype of *CtrlProgramOrganizationUnitType* or an instance of one of its subtypes.

The *TargetNode* of this *ReferenceType* shall be a *Variable* or an *Object* of the *ObjectType CtrlFunctionBlockType* or one of its subtypes.

8.6 HasExternalVar

This *ReferenceType* is a subtype of the *Organizes ReferenceType* defined in OPC 10000-5. Its representation in the *AddressSpace* is specified in Table 24.

Table 24 – HasExternalVar ReferenceType

Attributes	**Value**		
BrowseName	HasExternalVar		
InverseName	ExternalVarOf		
Symmetric	False		
IsAbstract	False		
References	**NodeClass**	**BrowseName**	**Comment**
Subtype of Organizes Reference*Type* defined in OPC 10000-5			

The *HasExternalVar ReferenceType* is a concrete *ReferenceType* and can be used directly.

The semantic of this *ReferenceType* is to reference components of a *Ctrl Program Organization Unit* declared with the key word VAR_EXTERNAL.

The *SourceNode* of *References* of this type shall be a subtype of *CtrlProgramOrganizationUnitType* or an instance of one of its subtypes.

The *TargetNode* of this *ReferenceType* shall be a *Variable* or an *Object* of the *ObjectType CtrlFunctionBlockType* or one of its subtypes.

8.7 With

This *ReferenceType* is a subtype of the *NonHierarchicalReferences ReferenceType* defined in OPC 10000-5. Its representation in the *AddressSpace* is specified in Table 25.

Table 25 – With ReferenceType

Attributes	Value		
BrowseName	With		
InverseName	Executes		
Symmetric	False		
IsAbstract	False		
References	**NodeClass**	**BrowseName**	**Comment**
Subtype of NonHierarchicalReferences Reference*Type* defined in OPC 10000-5			

The *With ReferenceType* is a concrete *ReferenceType* and can be used directly.

The semantic of this *ReferenceType* is to reference the *Ctrl Task* that executes a *Ctrl Program Organization Unit*.

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

The *TargetNode* of this *ReferenceType* shall be an *Object* of the *ObjectType CtrlTaskType* or one of its subtypes.

9 Definition of Ctrl Variable Attributes and Properties

9.1 Common Attributes

The common *Attributes* of *OPC UA Address Space Nodes* and their mapping from IEC 61131-3 are defined in Table 26.

Table 26 – Common Node Attributes

Attribute	Use	DataType	Description
NodeId	Mandatory	NodeId	The *NodeId* is a unique identifier for a *Node* in an *OPC UA Address Space*. The identifier is server specific and its format is not defined in this specification.
NodeClass	Mandatory	NodeClass	The *NodeClass* is Variable for all *Ctrl Variables*.
BrowseName	Mandatory	QualifiedName	The *BrowseName* is a *QualifiedName* composed of a name string and a namespace index. It is used to create paths that can be passed to the *TranslateBrowsePathsToNodeIds Service* to get the *NodeId* of a *Variable Node*. This is typically used to get the NodeId of Variable in an *Object* instance based on the path known from the *Object* type. The *BrowseName* is not used to display the name of the *Node*. The name part is generated from the *Ctrl Variable* name. The namespace part depends of the scope where the Variable is defined. Chapter 12.5 describes the handling of namespaces.
DisplayName	Mandatory	LocalizedText	The *DisplayName* is a LocalizedText used by clients to display the name of a Node. It is composed of a localized text part and a LocaleId identifying the language of the text. The *DisplayName* is server specific if the server supports localization of *Variable* names. The *DisplayName* is composed of the *Ctrl Variable* name and an empty *LocaleId* string if the server does not support localization.
Description	Optional	LocalizedText	The optional Description shall describe the meaning of the Node using a localized text. The *Description* may correspond to the element *Documentation* of the element *Variable* in PLCopen XML.
WriteMask	Optional	UInt32	The *WriteMask* provides the optional information which attributes of the *Node* can be written by a client. This excludes the *Value Attribute* where the access is described by the *AccessLevel Attribute*. The value of this Attribute is server specific. Servers only supporting the use cases *Observation* and *Operation* are typically setting this *Attribute* to 0 or are not providing this optional *Attribute*. Servers supporting also the use cases *Engineering* and *Service* may allow clients to change *Node Attributes*.
UserWriteMask	Optional	UInt32	The user specific settings for the *WriteMask*.

9.2 DataType

9.2.1 Mapping of elementary data types

The mapping of IEC 61131-3 elementary data types to OPC UA data types is formally defined in Table 27. The OPC UA built in data types are used for the wire representation of the data type. Additional PLCopen specific OPC UA data type definitions are used to provide the special semantic if necessary.

Table 27 – Mapping IEC 61131-3 elementary data types to OPC UA built in data types

No.	IEC 61131-3 elementary data types (Keyword / Description)	DataType NodeID	OPC UA built in data types	PLCopen specific OPC UA simple data type definitions	Comment
1	BOOL / Boolean	---	Boolean (UA:1)	-	A one bit value (true or false).
2	SINT / Short integer	---	SByte (UA:2)	-	An 8 bit signed integer value.
3	INT / Integer	---	Int16 (UA:4)	-	A 16 bit signed integer value.
4	DINT	---	Int32 (UA:6)	-	A 32 bit signed integer value.
5	LINT / Long integer	---	Int64 (UA:8)	-	A 64 bit signed integer value.
6	USINT / Unsigned short integer	---	Byte (UA:3)	-	An 8 bit unsigned integer value.
7	UINT / Unsinged integer	---	UInt16 (UA:5)	-	A 16 bit unsigned integer value.
8	UDINT / Unsigned double	---	UInt32 (UA:7)	-	A 32 bit unsigned integer value.
9	ULINT / Unsigned long integer	---	UInt64 (UA:9)	-	A 64 bit unsigned integer value.
10	REAL / Real numbers	---	Float (UA:10)	-	OPC UA definition: An IEEE-754 single precision (32 bit) floating point value. IEC 61131-3 definition: Real (32 bit) with a range of values as defined in IEC 60559 for the basic single width floating-point format. Both standards are identical.
11	LREAL / Long reals	---	Double (UA:11)	-	OPC UA definition: An IEEE-754 double precision (64 bit) floating point value. IEC 61131-3 definition: Long real (64 bit) with a range of values as defined in IEC 60559 for the basic double width floating-point format. Both standards are identical.
12a	TIME / Duration	x:3005	Int64 (UA:8)	TIME	The OPC UA simple data type TIME/Duration is derived from the built-in data type Int64. It describes that the type is used as interval of time in milliseconds. The range of valid values is vendor specific.
12b	LTIME / Duration	x:3006	Int64 (UA:8)	LTIME	The PLCopen simple data type LTIME is derived from the build in data type Int64. It describes that the type is used as interval of time in nanoseconds. The valid range is LT#-106751d23h47m16s854ms775us808ns to LT#+106751d23h47m16s854ms775us807ns. The representation contains information for days (d), hours (h), minutes (m), seconds (s) milliseconds (ms), microseconds (us) and nanoseconds (ns).
13a	DATE / Date (only)	x:3007	DateTime (UA:13)	DATE	The PLC open specific OPC UA simple data type DATE is derived from the built-in data type DateTime. It describes that the type is used as a date only.
13b	LDATE / Long date (only)	X:3014	Int64 (UA:8)	LDATE	The PLCopen specific OPC UA simple data type LDATE is derived from the built-in data type Int64. It describes that the type is used as date only. The interval is nanoseconds since 1970-01-01.
14a	TOD Time of day (only)	x:3008	UInt32 (UA:7)	TOD	TOD (TIME_OF_DAY) stores number of milliseconds since the beginning of the day: TOD#00:00:00.000 to TOD#23:59:59.999.
14b	LTOD (Time of day)	x:3009	Int64 (UA:8)	LTOD	LTOD (LTIME_OF_DAY) stores the number of nanoseconds since the beginning of the day: LTOD#00:00:00.000000000 to LTOD#23:59:59.999999999.
15a	DT Date and time of day	x:3010	DateTime (UA:13)	DT	The range and resolution of this type is vendor specific.

15b	LDT Date and time of day	x:3015	Int64 (UA:8)	LDT	The PLCopen specific OPC UA Simple datatype LDT is derived from the build in data type Int64. It describes the number of nanoseconds elapsed since 1970-01-01-00:00:00
16a	STRING variable-length single-byte character string	x:3013	String (UA:12)	STRING	The PLC open specific OPC UA simple data type STRING is derived from the built-in data type String. It describes that the type is used as a variable-length single-byte character string.
16b	WSTRING variable-length double-byte character string	---	String (UA:12)	-	OPC UA definition: A sequence of UTF8 characters. IEC 61131-3 definition: Variable-length double-byte character string
17a	CHAR single-byte character	x:3011	Byte (UA:3)	CHAR	The PLC open specific OPC UA simple data type CHAR is derived from the built-in data type Byte. It describes that the type is used as single-byte character
17b	WCHAR double-byte character	x:3012	UInt16	WCHAR	The PLC open specific OPC UA simple data type WCHAR is derived from the built-in data type UInt16. It describes that the type is used as double-byte character.
18	BYTE Bit string of length 8	x:3001	Byte	BYTE	The PLC open specific OPC UA simple data type BYTE is derived from the built-in data type Byte. It describes that the type is used as bit string of length 8.
19	WORD Bit string of length 16	x:3002	UInt16	WORD	The PLC open specific OPC UA simple data type WORD is derived from the built-in data type UInt16. It describes that the type is used as bit string of length 16
20	DWORD Bit string of length 32	x:3003	UInt32	DWORD	The PLC open specific OPC UA simple data type DWORD is derived from the built-in data type UInt32. It describes that the type is used as bit string of length 32
21	LWORD Bit string of length 64	x:3004	UInt64	LWORD	The PLC open specific OPC UA simple data type LWORD is derived from the built-in data type UInt64. It describes that the type is used as bit string of length 64

9.2.2 Mapping of generic data types

The mapping of IEC 61131-3 generic data types to OPC UA data types is formally defined in Table 28. Since the generic data type should not be used in user-declared Ctrl Program Organization Units, this mapping definition is defined for completeness but is normally not used in an OPC UA AddressSpace.

Table 28 – Mapping IEC 61131-3 generic data types to OPC UA data types

IEC 61131-3 generic data types	OPC UA data types	Description
ANY	BaseDataType	This abstract OPC UA *DataType* defines a value that can have any valid OPC UA *DataType*.
ANY_DERIVED	BaseDataType	
ANY_ELEMENTARY	BaseDataType	
ANY_MAGNITUDE	BaseDataType	
ANY_NUM	Number	This abstract OPC UA *DataType* defines a number value that can have any of the OPC UA Number subtypes.
ANY_REAL	Number	
ANY_INT	Number	
ANY_BIT	Number	
ANY_STRING	String	This OPC UA Built-in DataType defines a Unicode character string that should exclude control characters that are not whitespaces (0x00 - 0x08, 0x0E-0x1F or 0x7F).
ANY_DATE	DateTime	This OPC UA *Built-in DataType* defines a Gregorian calendar date. It is a 64-bit signed integer which represents the number of 100 nanosecond intervals since January 1, 1601.

9.2.3 Mapping of derived data types

9.2.3.1 Mapping of enumerated data types

Both OPC UA and IEC 61131-3 allow the definition of enumerations on a data type or on a variable instance.

In OPC UA the enumerated data types are defined as subtypes of Enumeration. The data has an *EnumStrings Property* that contains the possible string values. The value is transferred as integer on the wire where the integer defines the index into the *EnumStrings* array. The index is zero based and has no gaps. Another option is to provide the possible string values in the *Property EnumValues*. This option is used if individual integer values are assigned to the string. The used option depends on the way the string enumeration is defined in the *Controller* program. If integer values are assigned to the string values the *Property EnumValues* is used to represent the enumeration values. If the integer value is zero based and has no gaps the *EnumStrings Property* should be used since the processing on the client side is more efficient.

The definition on a variable instance is using the MultiStateDiscreteType Variable Type which defines also the *EnumStrings* or the *EnumValues Property* containing the enumeration values as string array.

Example for an enumerated data type declaration in IEC 61131-3:

```
TYPE

  ANALOG_SIGNAL_TYPE : (SINGLE_ENDED, DIFFERENTIAL) ;

END_TYPE
```

Example for use of an enumeration in a *Ctrl Variable* instantiation in IEC 61131-3:

```
VAR

  Y : (Red, Yellow, Green) ;

END_VAR
```

The IEC 61131-3 enumeration data type declaration is mapped to an OPC UA Enumeration data type. The representation in the address space is formally defined in Table 29.

Table 29 – Enumeration Data Type Definition

References	Node Class	BrowseName	DataType	TypeDefinition	Modelling Rule
Subtype of the Enumeration defined in OPC 10000-5 i.e. inheriting the InstanceDeclarations of that Node.					
HasProperty	Variable	EnumString	String [ ]	PropertyType	Optional
HasProperty	Variable	EnumValues	EnumValueDataType [ ]	PropertyType	Optional

The Property *EnumString* is defined in OPC 10000-5

The *Property EnumValues* is defined in OPC 10000-5.

The IEC 61131-3 enumeration in a *Ctrl Variable* declaration is mapped to a MultiStateDiscreteType Variable Type defined in OPC 10000-8.

9.2.3.2 Mapping of subrange data types

IEC 61131-3 defines the subrange for all integer data types (ANY_INT) which excludes real values.

OPC UA has no standard concept to limit the range on the data type.

Example for a subrange data type declaration in IEC 61131-3:

```
TYPE

  ANALOG_DATA : INT (-4095..4095) ;

END_TYPE
```

Example for use of a subrange in a *Ctrl Variable* instantiation in IEC 61131-3:

```
VAR

  Z : SINT (5..95) ;
```

```
END_VAR
```

The IEC 61131-3 subrange is mapped to two OPC UA properties defined in Table 30.

Table 30 – Subrange Property Definition

References	Node Class	BrowseName	DataType	TypeDefinition	Modelling Rule
Instance of any Variable Type or a Data Type Node.					
HasProperty	Variable	SubrangeMin	Number	PropertyType	Mandatory
HasProperty	Variable	SubrangeMax	Number	PropertyType	Mandatory

The Property SubrangeMin contains the lower bound of the subrange. The data type depends on the elementary data type used for the subrange.

The Property SubrangeMax contains the upper bound of the subrange. The data type depends on the elementary data type used for the subrange.

The IEC 61131-3 subrange data type is mapped to an OPC UA number data type derived from the corresponding elementary data types defined in Table 27. The data type has the two Properties defined in Table 30. The IEC example in this chapter is mapped to an OPC UA data type with the name ANALOG_DATA which is a subtype of Int16.

The IEC 61131-3 subrange in a *CtrlVariable* declaration is mapped to the two Properties defined in Table 30. The Properties are children of the OPC UA Variable representing the *CtrlVariable*.

9.2.3.3 Mapping of array data types

OPC UA provides the information if a value is an array in the Variable Attributes ValueRank and ArrayDimensions. Every data type can be exposed as array. Arrays can have multiple dimensions. The dimension is defined through the Attribute ValueRank. Arrays can have variable or fixed lengths. The length of each dimension is defined by the Attribute ArrayDimensions. The array index starts with zero.

IEC 61131-3 allows the declaration of array data types with one or multiple dimensions and an index range instead of a length.

OPC UA has no standard concept for defining special array data types or exposing index ranges.

Example for an array data type declaration in IEC 61131-3:

```
TYPE

  ANALOG_16_INPUT_DATA : ARRAY [1..16] OF INT ;

END_TYPE
```

Example for use of an array in a *CtrlVariable* instantiation in IEC 61131-3:

```
VAR

  MyArray : ARRAY [1..16] OF INT;

END_VAR
```

The IEC 61131-3 array data type is mapped to three OPC UA properties defined in Table 31.

Table 31 – Array Data Type Property Definition

References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Instance of any *VariableType* or a *DataType* Node.					
HasProperty	Variable	Dimensions	UInt32	PropertyType	Mandatory
HasProperty	Variable	IndexMin	Int32 [ ]	PropertyType	Mandatory
HasProperty	Variable	IndexMax	Int32 [ ]	PropertyType	Mandatory

The Property Dimensions contains the number of dimensions of the array.

The Property IndexMin contains an array of lower bounds, one for each array dimension.

The Property IndexMax contains an array of upper bounds, one for each array dimension.

The IEC 61131-3 array data type is mapped to an OPC UA data type derived from the corresponding elementary data types defined in Table 27. The data type has the two Properties defined in Table 31. The IEC example in this chapter is mapped to an OPC UA data type with the name ANALOG_16_INPUT_DATA which is a subtype of Int16.

The IEC 61131-3 array in a *Ctrl Variable* declaration is mapped to the two Properties defined in Table 31. The Properties are children of the OPC UA Variable representing the *Ctrl Variable*.

9.2.3.4 Mapping of structure data types

IEC 61131-3 structure data types are mapped as subtypes of the OPC UA DataType Structure. OPC UA servers must explicitly describe how structured DataTypes are encoded / decoded and provide this information to the client which is using it while reading / writing structure data.

The following example of an IEC 61131-3 structure data type declaration (using Structured Text) will be used for further illustrations. This structure data type comprises three structure elements of different elementary data types.

```
TYPE ExampleIEC611313Structure:
  STRUCT
    IntStructureElement: INT;
    RealStructureElement: REAL;
    BoolStructureElement: BOOL;
  END_STRUCT;
END_TYPE
```

9.2.3.4.1 Deprecated Mapping of structure data types

The following Figure 22 shows the deprecated mapping of the above example. This mapping is deprecated since it is deprecated in OPC UA V1.04.

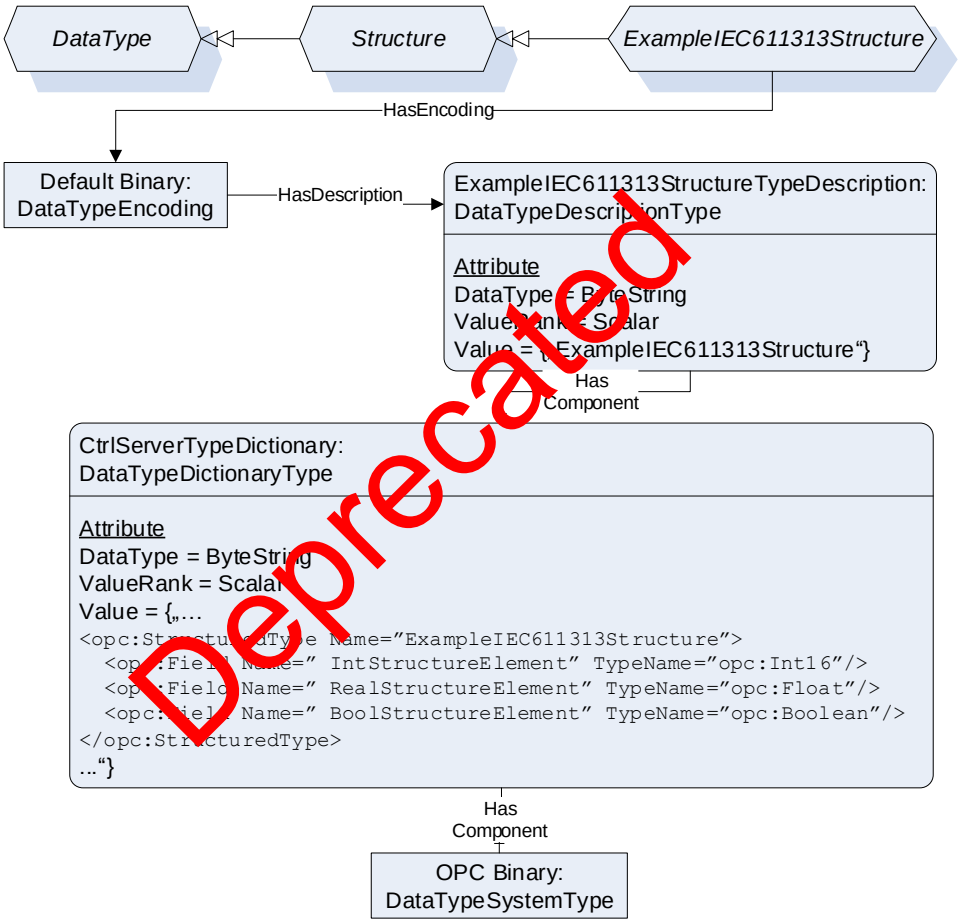

Figure 22 – Deprecated Mapping of structure data types

Ctrl servers must support the binary encoding (“Default Binary”). Additionally, other encodings may be provided (not shown in above figure). A Server may provide, for backward compatibility, the deprecated DataTypeDictionary Variable describing all necessary DataTypes. Each DataType is represented by a DataTypeDescription Variable. Optionally, a Property DictionaryFragment may be available, allowing clients not to read the complete DataTypeDictionary in order to get the information about only a single DataType (not shown in above figure).

9.2.3.4.2 Mapping of structure data types

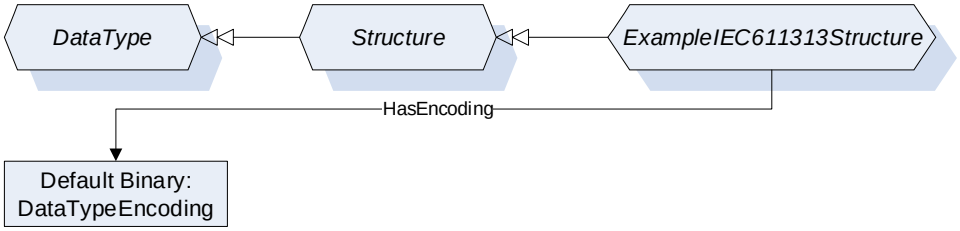

Figure 23 – Mapping of structure data types

Ctrl servers shall support the binary encoding (“Default Binary”). Additionally, other encodings may be provided (not shown in above figure). Since OPC UA V1.04 a structured DataType provides the new attribute DataTypeDefinition. This attribute is defined in OPC 10000-6 – F.12. Implementations shall use this new attribute instead of the deprecated DataTypeDictionary.

A Server provides on a structured DataType Node the DataTypeDefinition attribute describing all elements and their order in this structure.

The Value of the DataTypeDefinition Attribute for a DataType Node describing ExampleIEC611313Structure is shown in Table 32.

Table 32 – Value of the DataTypeDefinition

Name	Type	Description
defaultEncodingId	NodeId	NodeId of the “ExampleIEC611313Structure_Encoding_DefaultBinary” Node.
baseDataType	NodeId	“i=22” [Structure]
structureType	StructureType	Structure_0 [Structure without optional fields]
fields [0]	StructureField	
name	String	“IntStructureElement”
description	LocalizedText	Description of IntStructureElement
dataType	NodeId	“i=4” [Int16]
valueRank	Int32	-1 (Scalar)
isOptional	Boolean	False
fields [1]	StructureField	
Name	String	“RealStructureElement“
Description	LocalizedText	Description of RealStructureElement
dataType	NodeId	“i=10” [Float]
valueRank	Int32	-1 (Scalar)
isOptional	Boolean	false
fields [2]	StructureField	
name	String	“BoolStructureElement“
description	LocalizedText	Description of BoolStructureElement
dataType	NodeId	“i=1” [Boolean]
valueRank	Int32	-1 (Scalar)
isOptional	Boolean	false

9.2.3.4.3 Structure and VariableType

It is strongly recommended for Ctrl servers to provide additionally the structured data as a set of sub variables (components of the variable) providing the structure as several separated values. This allows clients that do not support complex data to access the scalar values. The following Figure 24 shows an example (instances based on the above type descriptions).

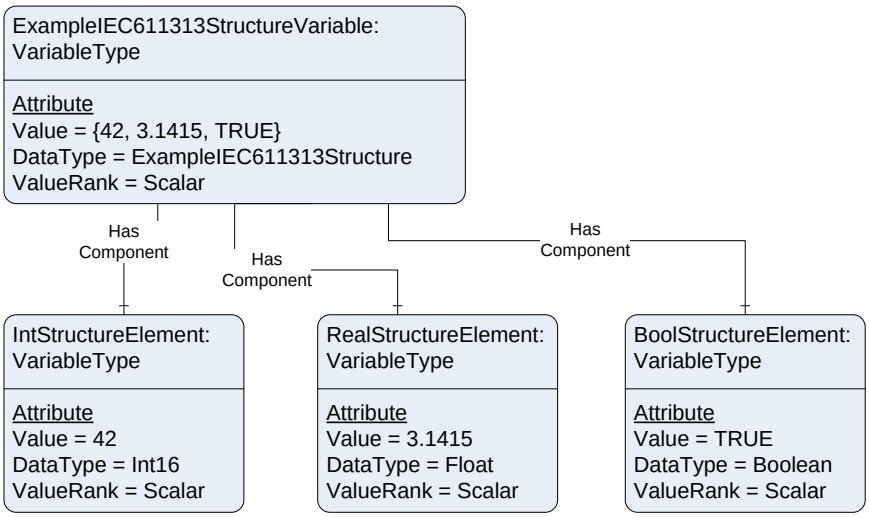

Figure 24 – Mapping of structure data types to Variable components

If a structure element is not an elementary data type, it has to be divided again into sub variables.

It is recommended that Ctrl servers do support complex data. If a server does not support complex data it provides only sub variables for structure variables. The structured variable would be a Folder object in this case.

9.3 Variable specific Node Attributes

9.3.1 General

The *Variable* specific *Attributes* of *OPC UA Address Space Nodes* and their mapping from IEC 61131-3 are defined in Table 33.

Table 33 – Variable Node Attributes

Attribute	Use	DataType	Description
Value	Mandatory	Defined by DataType	The most recent value of the *Variable* that the server has. Its data type is defined by the *DataType, ValueRank* and *ArrayDimension* Attribute.
DataType	Mandatory	NodeId	The *DataType* of the *Variable Value*. It defines the type specific content of the Value together with the *ValueRank* and the *ArrayDimension Attributes*, The mapping is defined in 9.2.
ValueRank	Mandatory	Int32	This *Attribute* indicates whether the *Value* of the *Variable* is an array and how many dimensions the array has. *Ctrl Variables* declared as scalar type have the *ValueRank* -1. *Ctrl Variables* declared with the key word ARRAY…OF have a *ValueRank* that indicates the number of dimension of the array declared for the *Ctrl Variable*.
ArrayDimensions	Optional	UInt32[]	This *Attribute* specifies the length of each dimension for an array value. The *Attribute* is intended to describe the capability of the *Variable*, not the current size. The dimension entries have the length defined with the key word ARRAY…OF.
AccessLevel	Mandatory	Byte	The AccessLevel Attribute is used to indicate how the Value of a *Variable* can be accessed (read/write) and if it contains current and/or historic data. The handling of access to historic data is server specific and is not part of this specification. The mapping of the read and write access part of the *AccessLevel* is defined in 9.3.2
UserAccessLevel	Mandatory	Byte	The user specific settings for the *AccessLevel*.
MinimumSampling Interval	Optional	Duration	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. A MinimumSamplingInterval of 0 indicates that the server is to monitor the item continuously. A MinimumSamplingInterval of -1 means indeterminate. The value of this Attribute is server specific.
Historizing	Mandatory	Boolean	Indicates if the server is currently collecting history for the *Variable Value*. The support of value history is server specific.

9.3.2 Access Level

If the IEC attribute CONSTANT is set, the Access Level shall be read only.

The IEC standard does not define a key word to set the Access Level for a Ctrl Variable. The configuration in a programming system is vendor specific but it is recommended to provide a configuration option for the OPC UA Access Level.

When using the PLCopen XML format the AccessLevel shall be provided in PLCopen XML *Additional Data* in the XML element *addData* using the XML element UaAccessLevel as part of the XML element representing the variable. The value is a bit mask where the first bit indicates the read access and the second bit indicates the write access.

9.4 Variable Properties

9.4.1 IEC Ctrl Variable Keywords

The IEC 61131-3 key word mapping to OPC UA *Properties* is formally defined in Table 34.

Table 34 – IEC 61131-3 Variable Key Word Property Definition

References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Instance of any Variable Type.					
HasProperty	Variable	RETAIN	Boolean	PropertyType	Optional
HasProperty	Variable	NON_RETAIN	Boolean	PropertyType	Optional
HasProperty	Variable	CONSTANT	Boolean	PropertyType	Optional
HasProperty	Variable	AT	String	PropertyType	Optional

The *Property* RETAIN indicates if the RETAIN key word is set for the *Ctrl Variable*. It provides an explicit declaration of “warm start” behaviour of the *Ctrl Variable* (and *Ctrl Function Blocks* and *Ctrl Programs*).

The *Property* NON_RETAIN indicates if the NON_RETAIN key word is set for the *Ctrl Variable*. It provides an explicit declaration of “warm start” behaviour of the *Ctrl Variable* (and *Ctrl Function Blocks* and *Ctrl Programs*).

The *Property* CONSTANT indicates if the CONSTANT key word is set for the *Ctrl Variable*. It provides a declaration of a fixed value for the *Ctrl Variable*. The *Ctrl Variable* cannot be modified.

The *Property* AT contains the location assignment to the *Ctrl Variable* as string if the AT key word is set for the *Ctrl Variable*.

9.4.2 Configuration of OPC UA defined Properties

The IEC standard does not define key words to configure information like the value range or the engineering unit for a *Ctrl Variable*. The configuration in a programming system is vendor specific but this specification defines the export format in the PLCopen XML *Additional Data* in the XML element *addData*.

The *InstrumentRange Property* defined in OPC 10000-8 shall be provided in the XML element *UaInstrumentRange* as part of the XML element representing the *Ctrl Variable*. The attributes of the XML element are formally defined in Table 35.

Table 35 – Range XML attributes

Name	Type	Use	Default
Low	double	required	
High	double	required	

The *EURange Property* defined in OPC 10000-8 shall be provided in the XML element *UaEURange* as part of the XML element representing the *Ctrl Variable*. The attributes of the XML element are formally defined in Table 35.

The *EngineeringUnits Property* defined in OPC 10000-8 shall be provided in the XML element *UaEngineeringUnits* as part of the XML element representing the *Ctrl Variable*.

10 Objects used to organise the AddressSpace structure

10.1 DeviceSet as entry point for engineering applications (Mandatory)

The full object component hierarchy based on *Object Types* defined in this specification shall be provided as components of the *DeviceSet Object* defined in OPC 10000-100. Figure 25 provides an example for such a component hierarchy.

The *DeviceSet Object* is typically used as entry point by a UA client in the use cases *Engineering* and *Service*.

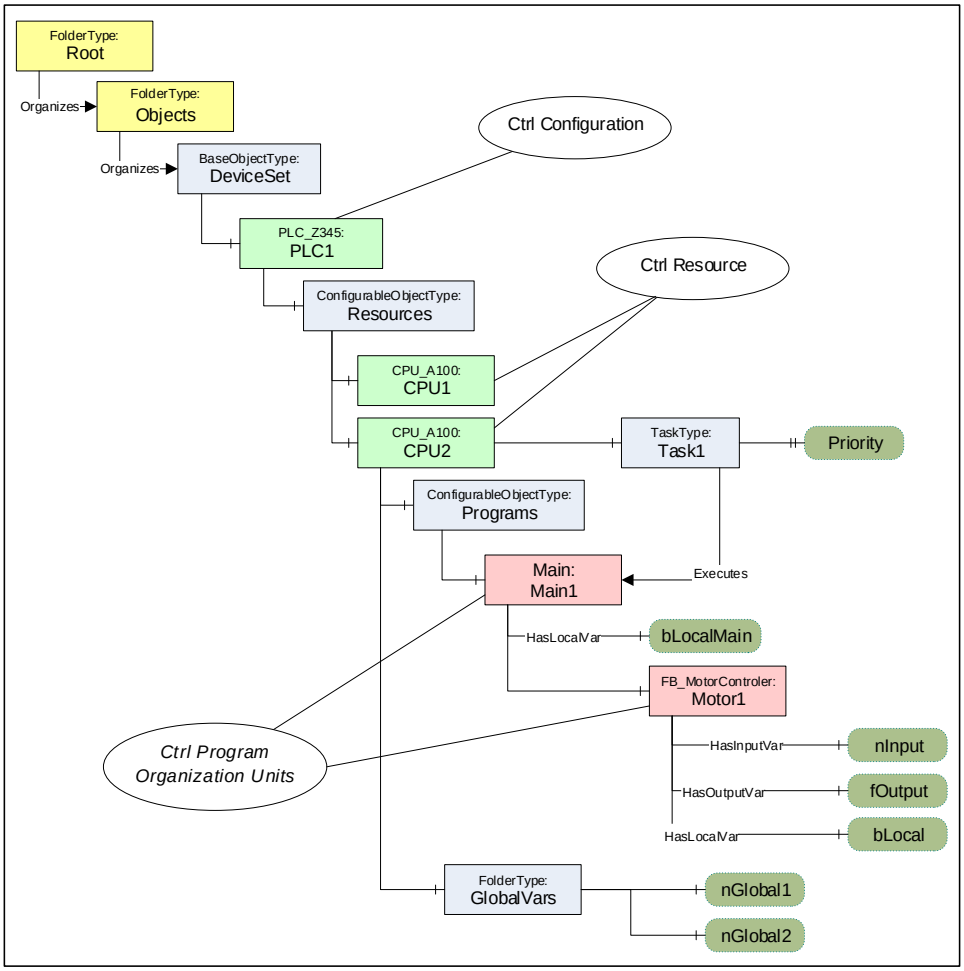

Figure 25 – DeviceSet as entry point for engineering applications

10.2 CtrlTypes Folder for server specific Object Types (Mandatory)

The server specific *ObjectTypes* like vendor specific *Ctrl Configuration* types or user specific *Ctrl Function Block* types can be found by a UA client by following the type hierarchy.

To provide UA clients all relevant server specific types in one place, the *Ctrl Function Block* types shall be referenced directly or indirectly from the *CtrlTypes Folder Object* using *Organizes References*. Other types like *Ctrl Resources* or *Ctrl Program* types may be included in addition. The CtrlTypes node is formally defined in Table 36

Table 36 – CtrlTypes definition

References	Node Class	BrowseName	TypeDefinition	Description
Organized by the ObjectType Folder defined in OPC 10000-5				
HasTypeDefinition	ObjectType	Folder		
Organizes	Object	<Server specific>	FolderType	Optional server specific additional structuring of the type information building to a type catalogue
Organizes	ObjectType	<Server specific>		Server specific Object Types

The server may provide additional Folder objects below the *CtrlTypes Object* to organize the types. This can be used to create a library structure like in the example in Figure 26.

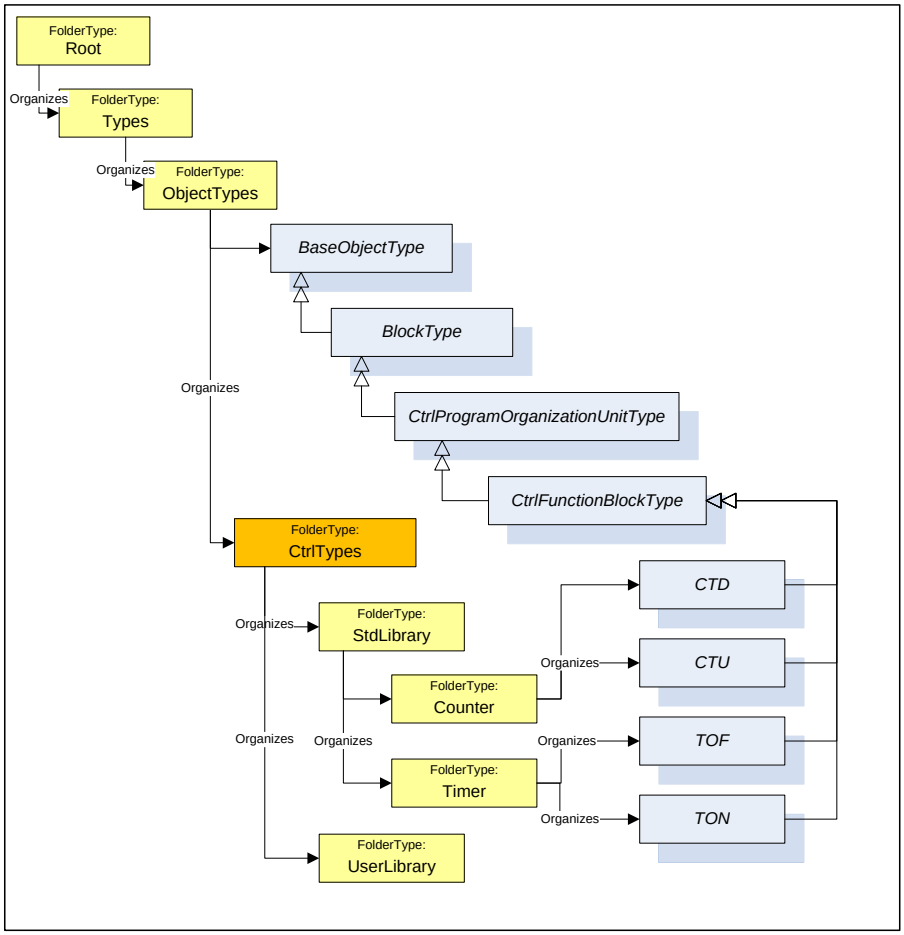

Figure 26 – CtrlTypes Folder used to structure POU types

10.3 Entry point for Observation and Operation (Examples)

The entry point for UA client for the use cases *Observation* and *Operation* is the *Objects Folder*. One typical entry point is a list of *Objects* representing *Ctrl Resources*. Additional *Folders Objects* used to structure the *Ctrl Resources* into a hierarchy are server specific. Such an example is shown in Figure 27.

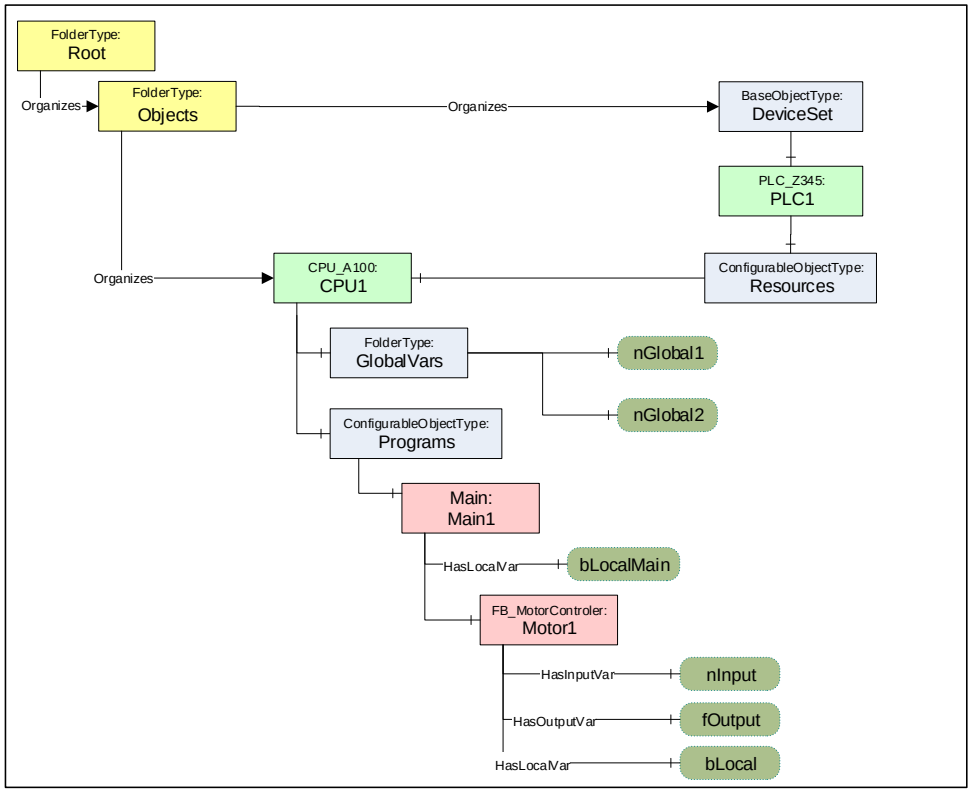

Figure 27 – Browse entry point for Operation with Ctrl Resource

Servers that want to hide some of the components of a *Ctrl Resources* can create a *Folder Object* representing the *Ctrl Resources* and can use *Organizes References* to reference only the components of the *Ctrl Resources* that should be visible in this part of the hierarchy. Such an example is shown in Figure 28.

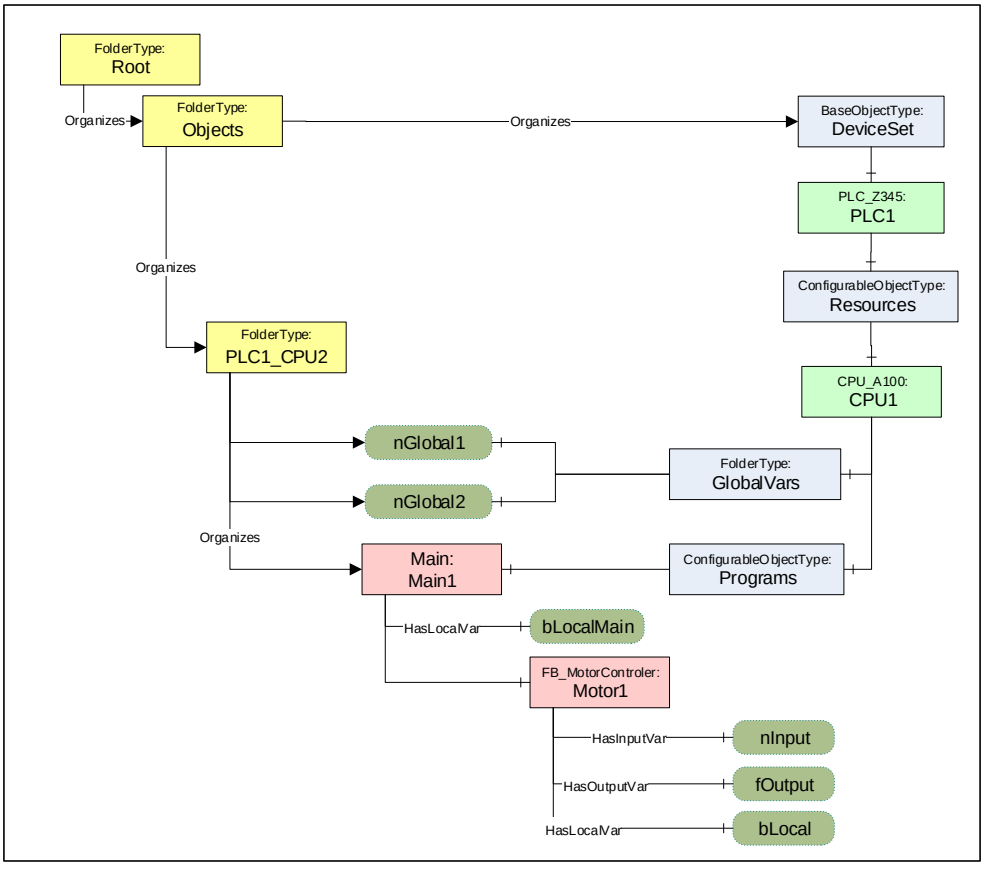

Figure 28 – Browse entry point for Operation with simplified Folder

11 System Architecture

11.1 General

This chapter describes typical system architectures where this specification can be applied. Figure 29 shows a possible configuration where OPC UA based interfaces are involved.

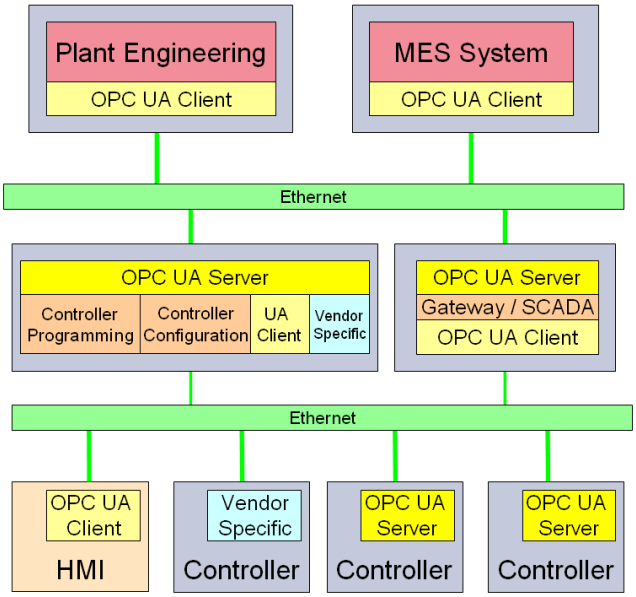

Figure 29 – System Architecture

11.2 Embedded OPC UA Server

Embedded OPC UA servers are directly integrated into a *Controller* providing *Ctrl Program* and *Ctrl Function Block Objects*. Such a server allows direct access to information from a *Controller* using the OPC UA protocol on the wire. Other embedded applications like HMI acting as OPC UA clients can access the information from *Controllers* directly without the need of a PC.

11.3 PC based OPC UA Server

OPC UA servers running on a PC platform are capable of providing access to multiple *Controllers*. They are providing full type information for *Ctrl Resource*, *Ctrl Program and Ctrl Function Block Objects*. The communication to the *Controllers* may use OPC UA or a proprietary protocol on the wire.

11.4 PC based OPC UA Server with engineering capabilities

In addition to PC based OPC UA servers, this type of server includes access to the engineering system for the *Controllers* allowing access to the configuration for the use cases *Engineering* and *Service*.

12 Profiles and Namespaces

12.1 Namespace Metadata

Table 37 defines the namespace metadata for this specification. The *Object* is used to provide version information for the namespace and an indication about static *Nodes*. Static *Nodes* are identical for all *Attributes* in all *Servers*, including the *Value Attribute*. See Part5 for more details.

The information is provided as *Object* of type *NamespaceMetadataType*. This *Object* is a component of the *Namespaces Object* that is part of the *Server Object*. The *NamespaceMetadataType ObjectType* and its *Properties* are defined in OPC 10000-5.

The version information is also provided as part of the ModelTableEntry in the UANodeSet XML file. The UANodeSet XML schema is defined in OPC 10000-6.

Table 37 – NamespaceMetadata Object for this Specification

Attribute	Value		
BrowseName	http://PLCopen.org/OpcUa/IEC61131-3/		
References	**BrowseName**	**DataType**	**Value**
HasProperty	NamespaceUri	String	http://PLCopen.org/OpcUa/IEC61131-3/
HasProperty	NamespaceVersion	String	1.02
HasProperty	NamespacePublicationDate	DateTime	2020-11-25
HasProperty	IsNamespaceSubset	Boolean	Vendor-specific
HasProperty	StaticNodeIdTypes	IdType[]	{Numeric}
HasProperty	StaticNumericNodeIdRange	NumericRange[]	
HasProperty	StaticStringNodeIdPattern	String	

12.2 Conformance Units and Profiles

This chapter defines the corresponding *Profiles* and *Conformance Units* for the OPC UA Information Model for IEC61131-3. *Profiles* are named groupings of *Conformance Units*. *Facets* are *Profiles* that will be combined with other *Profiles* to define the complete functionality of an OPC UA *Server* or *Client.*

12.3 Server Facets

The following tables specify the *Facets* available for *Servers* that implement the IEC 61131-3 Information Model companion specification.

Table 38 defines *Conformance Units* included in the minimum needed facet. It requires the support for profile *BaseDevice Server Facet* defined in OPC 10000-100. It is used together with the *Embedded 2017 UA Server* profile or the *Standard 2017 UA Server* profile defined in OPC 10000-7.

A server supporting all data types including complex data types must support the *ComplexType Server Facet* defined in OPC 10000-7.

Table 38 – *Controller Operation Server Facet* Definition

Conformance Unit	Description	Optional/ Mandatory
Ctrl DeviceSet	Support the full component hirarchy with *Ctrl Configuration*, *Ctrl Resource*, *Ctrl Program* and *Ctrl FunctionBlock* below the DeviceSet Object defined in **OPC 10000-100**.	M
Ctrl Configuration	Support vendor defined *Ctrl Configuration* object types and object instances.	M
Ctrl Resource	Support vendor defined *Ctrl Resource* object types and object instances	M
Ctrl Program	Support user defined *Ctrl Program* object types and object instances.	M
Ctrl FunctionBlock	Support user defined *Ctrl FunctionBlock* object types and object instances.	M
Ctrl Task	Support of *Ctrl Task* objects.	O
Ctrl References	Support of reference types specified in the IEC 61131-3 Information Model companion standard.	O
Profile		
BaseDevice_Server_Facet (defined in **OPC 10000-100**)		M

Table 39 defines a facet for the support of the engineering information defined in the IEC 61131-3 Information Model. The *Controller Engineering Server Facet* requires the *Controller Operation Server Facet*.

Table 39 – *Controller Engineering Server Facet* Definition

Conformance Unit	Description	Optional/ Mandatory
Ctrl Engineering Data	Support to provide all engineering data defined in this specification like properties describing data types.	M
Ctrl Engineering Change	Support of engineering data changes through OPC UA	O
Ctrl Type Creation	Support of type node creation through *NodeManagement* Services to create *Ctrl Program Organization Unit* declarations.	O
Profile		
Controller Operation Server Facet		M

12.4 Client Facets

The following tables specify the *Facets* available for *Clients* that implement the IEC61131-3 Information Model companion specification.

Table 40 defines a facet available for *Clients* that implement the IEC 61131-3 Information Model standard. Servers implementing the *Controller Engineering Server Facet* may use this facet to restrict the engineering features to clients supporting this *Client* facet.

Table 40 – *Controller Engineering Client Facet* Definition

Conformance Unit	Description	Optional/ Mandatory
Ctrl Client Information Model	Consume objects that conform to the types specified in the IEC 61131-3 Information Model companion standard.	M
Ctrl Client Engineering Data	Consume engineering data defined in the IEC 61131-3 Information Model companion standard like properties describing data types.	M
Ctrl Client Engineering Change	Use engineering data changes through OPC UA	O
Ctrl Type Creation	Use type node creation through *NodeManagement* Services to create *Ctrl Program Organization Unit* declarations.	O
Profile		

12.5 Handling of OPC UA Namespaces

Namespaces are used by OPC UA to create unique identifiers across different naming authorities. The *Attributes NodeId* and *BrowseName* are identifiers. A *Node* in the UA *AddressSpace* is unambiguously identified using a *NodeId*. Unlike *NodeIds*, the *BrowseName* cannot be used to unambiguously identify a *Node*. Different *Nodes* may have the same *BrowseName*. They are used to build a browse path between two *Nodes* or to define a standard *Property*.

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 *EngineeringUnits Property*. Another example shown in Figure 30 is the ParameterSet and the GlobalVars object components of a Ctrl Resource instance. The ParameterSet node BrowseName shall use the OPC DI namespace and the GlobalVars node BrowseName shall use the namespace defined by this specification. All *NodeIds* of *Nodes* not defined in this specification shall not use the standard namespaces and are typically using the same namespace like the *Ctrl Resource* object instance, for example local server.

Table 41 provides a list of mandatory and optional namespaces used in a *Controller Server*.

Table 41 – Namespaces used in a Controller Server

NamespaceURI	Description	Use
http://opcfoundation.org/UA/	Namespace for *NodeIds* and *BrowseNames* defined in the OPC UA specification. This namespace shall have namespace index 0.	Mandatory
Local Server URI	Namespace for nodes defined in the local server. This may include types and instances used in a *Ctrl Resource* represented by the server. This namespace shall have namespace index 1.	Mandatory
http://opcfoundation.org/UA/DI/	Namespace for *NodeIds* and *BrowseNames* defined in **OPC 10000-100**. The namespace index is server specific.	Mandatory
http://PLCopen.org/OpcUa/IEC61131-3/	Namespace for *NodeIds* and *BrowseNames* defined in this specification. The namespace index is server specific.	Mandatory
http://PLCopen.org/OpcUa/IEC61131-3/FB/	A server may provide IEC or PLCopen defined *Ctrl Function Block* libraries.	Optional
User defined types and instances in a Ctrl Resource	A server that provides access to different *Ctrl Resources* may provide a separate namespace for each *Ctrl Resources* if it is required to create unique identifiers across *Ctrl Resources*.	Optional
Vendor specific types	A server may provide vendor specific types like types derived from *Ctrl Configuration* or *Ctrl Resource* in a vendor specific namespace.	Optional
Global user defined library	A server may provide global user defined *Ctrl Function Block* libraries in a user specific namespace.	Optional

Figure 30 shows an example for the use of namespaces in NodeIds and BrowseNames.

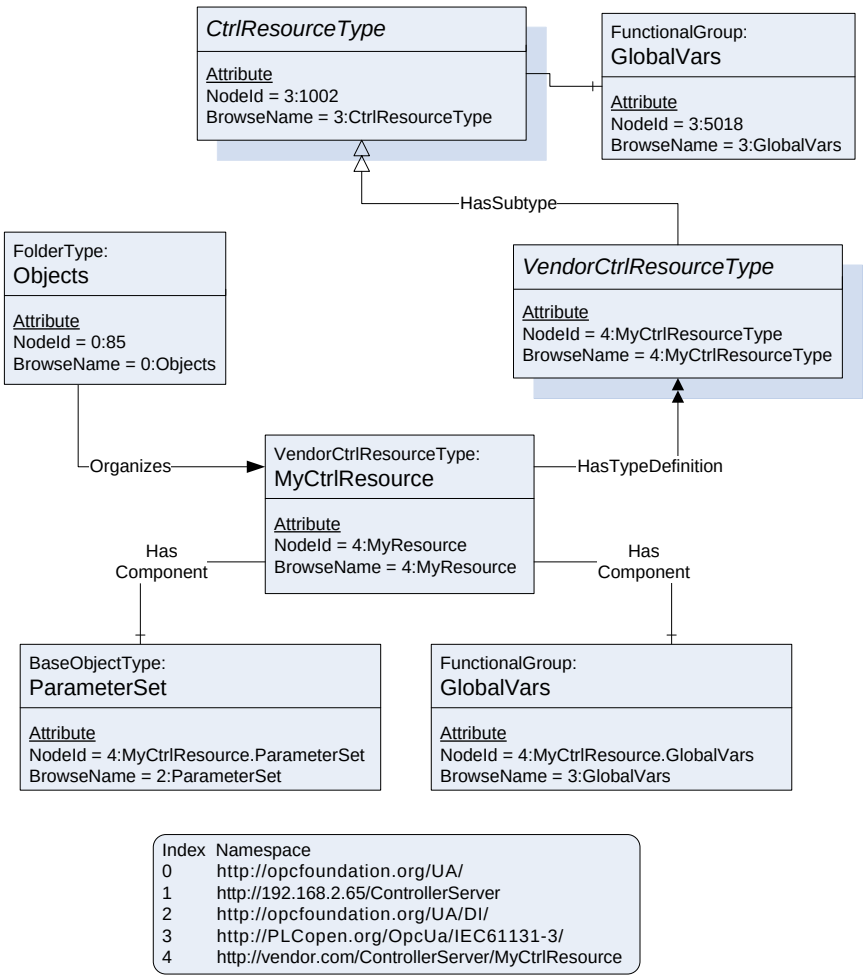

Figure 30 – Example for the use of namespaces in NodeIds and BrowseNames

Table 42 provides a list of namespaces and their index used for *BrowseNames* in this specification. The default namespace of this specification is not listed since all *BrowseNames* without prefix use this default namespace.

Table 42 – Namespaces used in this specification

NamespaceURI	Namespace Index	Example
http://opcfoundation.org/UA/	0	0:EngineeringUnits
http://opcfoundation.org/UA/DI/	2	2:DeviceRevision

Annex A (normative): IEC 61131-3 Namespace and mappings

A.1 Namespace and identifiers for IEC 61131-3 Information Model

This appendix defines the numeric identifiers for all of the numeric *NodeIds* defined in this specification. The identifiers are specified in a CSV file with the following syntax:

```
<SymbolName>, <Identifier>, <NodeClass>
```

Where the *SymbolName* is either the *BrowseName* of a *Type Node* or the *BrowsePath* for an *Instance Node* that appears in the specification and the *Identifier* is the numeric value for the *NodeId*.

The *BrowsePath* for an *Instance Node* is constructed by appending the *BrowseName* of the instance *Node* to the *BrowseName* for the containing instance or type. An underscore character is used to separate each *BrowseName* in the path. Let's take for example, the *CtrlConfigurationType ObjectType Node* which has the *ParameterSet Object*. The **Name** for the *ParameterSet InstanceDeclaration* within the *CtrlConfigurationType* declaration is: *CtrlConfigurationType_ParameterSet*.

The *NamespaceUri* for all *NodeIds* defined here is http://PLCopen.org/OpcUa/IEC61131-3/

A computer processible version of the complete Information Model defined in this specification is also provided. It follows the XML Information Model schema syntax defined in OPC 10000-6.

The Information Model Schema released with this version of the specification can be found here:

http://www.opcfoundation.org/UA/schemas/PLCOpen/1.02/Opc.Ua.PLCopen.NodeSet2_V1.02.xml

A.2 Profile URIs for IEC 61131-3 Information Model

Table 43 defines the Profile URIs for the IEC 61131-3 Information Model companion specification.

Table 43 – Profile URIs

Profile	Profile URI
Controller Operation Server Facet	http://PLCopen.org/OpcUa/IEC61131-3/Profile/Server/ControllerOperation
Controller Engineering Server Facet	http://PLCopen.org/OpcUa/IEC61131-3/Profile/Server/ControllerEngineering
Controller Engineering Client Facet	http://PLCopen.org/OpcUa/IEC61131-3/Profile/Client/ControllerEngineering

A.3 Namespace for IEC61131-3 Function Blocks

The namespace for all *Ctrl Function Block Type* nodes defined in other PLCopen documents like *Motion Ctrl Function Blocks* is "http://PLCopen.org/OpcUa/IEC61131-3/FB/".

The CSV file containing the numeric identifiers for this namespace can be found here:

http://www.PLCopen.org/OpcUa/IEC61131-3/FB/NodeIds.csv

The *NodeIds* for the defined nodes are composed of this namespace and the numeric identifier for the defined node.

Annex B (informative): PLCopen XML Additional Data Schema

B.1 XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:pra="http://www.plcopen.org/xml/tc6_0200/OpcUa"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.plcopen.org/xml/tc6_0200/OpcUa">

  <xsd:simpleType name="AccessLevel">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="Read" />
      <xsd:enumeration value="Write" />
      <xsd:enumeration value="ReadWrite" />
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:simpleType name="Visible">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="Yes" />
      <xsd:enumeration value="No" />
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:complexType name="InstrumentRange">
    <xsd:attribute name="Low" type="xsd:double" use="required" />
    <xsd:attribute name="High" type="xsd:double" use="required" />
  </xsd:complexType>

  <xsd:complexType name="EuRange">
    <xsd:attribute name="Low" type="xsd:double" use="required" />
    <xsd:attribute name="High" type="xsd:double" use="required" />
  </xsd:complexType>

  <xsd:complexType name="EUInformation">
    <xsd:sequence>
      <xsd:element name="DisplayName" type="LocalizedText" minOccurs="1" maxOccurs="1"/>
      <xsd:element name="Description" type="LocalizedText" minOccurs="0" maxOccurs="1"/>
    </xsd:sequence>
    <xsd:attribute name="NamespaceUri" type="xsd:string" use="optional" />
    <xsd:attribute name="UnitId" type="xsd:int" use="optional" />
  </xsd:complexType>

  <xsd:complexType name="InstanceInformation">
    <xsd:sequence>
      <xsd:element name="NodeId" type="xsd:string" minOccurs="1" maxOccurs="1"/>
      <xsd:element name="InstanceNamespaceUri" type="xsd:string" minOccurs="1" maxOccurs="1"/>
      <xsd:element name="Delimiter" type="xsd:string" minOccurs="0" maxOccurs="1"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="LocalizedText">
    <xsd:simpleContent>
      <xsd:extension base="xsd:string">
        <xsd:attribute name="Key" type="xsd:string" use="optional" default="" />
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>

  <xsd:complexType name="NodeId">
    <xsd:sequence>
      <xsd:element name="Identifier" type="xsd:string" minOccurs="0"
maxOccurs="1" nillable="true" />
    </xsd:sequence>
  </xsd:complexType>

</xsd:schema>
```