Integrating KNX and OPC UA Information Model

Salvatore Cavalieri, Ferdinando Chiacchio and Alberto Di Savia Puglisi
University of Catania, Department of Electrical Electronic and Computer Engineering, Catania, Italy
Email: salvatore.cavalieri@dieei.unict.it, chiacchio@dmi.unict.it, disalberto@gmail.com

Abstract—**Interoperability in all the levels of a communication architecture for automation applications cannot be easily satisfied because control devices are generally based on different technologies and communication protocols. This paper deals with the interoperability of a well-known communication standard for home and building automation, i.e. the KNX. Current literature presents several papers demonstrating that OPC Unified Architecture (OPC UA) can provide a cohesive, secure and reliable cross platform framework for interoperability in automation environment; some approaches propose the use of OPC UA to achieve interoperability of KNX. Starting from the current proposals, in this paper the authors propose further improvements always aimed to integrate KNX and OPC UA information model.**

Index Terms—**OPC UA, KNX, interoperability**

I. Introduction

Home and building automation is rapidly entering in our houses, thanks to a wide variety of commercial offers by both producers and installers. KNX is a worldwide standard for home and building control and represents one of the most important actor of this process of modernization [1][2][3].

KNX is not the only communication standard used for home and building automation and generally at the field level of the automation systems. Typical scenarios of a generic automation system (including home and building environment) feature the presence of different standards, causing problems of interoperability.

The main problems limiting the interoperability are present at the information level. Generally, data under control of dedicated devices are represented in different ways since each technology defines its own interworking model. This concerns how the data are modelled (i.e., the structure of data and the distribution to the devices) as well as the semantics (i.e., encoding, data types and other meta-data like engineering units, quality of data). For this reason interoperability has been always achieved by the adoption of expensive proprietary mapping.

OPC Unified Architecture (OPC UA) [4] can provide a cohesive, secure and reliable cross platform framework for interoperability, resulting a very good solution to perform the integration of different communication systems in automation environment.

Current literature presents some results achieved in this direction; in [5][6] and [7], an OPC UA-based interworking model is defined to make interoperable the KNX with other communication systems.

Starting from these proposals (i.e., [5][6] and [7]), in this paper the authors propose further improvements for the integration of KNX interworking model into OPC UA; they are mainly based on the definition of different ways to represent KNX functions and data structures into OPC UA information model. The main features of the novel proposal will be deeply described, pointing out the difference with the current literature. In order to better understand the proposal here presented, a simple case study will be shown and the relevant implementation of the KNX/OPC UA integration will be proposed. As it will be pointed out, it has been based on OPC UA .Net stack implementation made available by OPC Foundation [8].

II. KNX

KNX is one of the most worldwide used and known standards for home and building automation [9]. KNX communication stack allows exchanging of KNX telegrams through Physical, Data Link [10], Network, Transport and Application [11] layers.

KNX systems are typically distributed and based on Functional Blocks (FBs), representing key components in the KNX interworking model; they are distributed in different devices and, in order to exchange data, are connected each other via the KNX network. Each device can accommodate multiple FBs.

FBs are mainly made up by DataPoints; it is possible to distinguish input, output and parameter DataPoints. Each DataPoint is characterized by a well-defined DataPoint Type (DPT). Any DPT standardizes one combination of format (bit length), encoding, range (upper and lower limits) and unit [12]. A DPT is uniquely identified by a main and a sub number; the main number defines the format and encoding of DPT while the sub number concerns range and unit [12].

III. OPC UA

The OPC UA specifications are currently made up by 11 parts [13][14]. OPC UA allows applications to exchange information on the basis of a client/server

Manuscript received July 29, 2013; revised August 25, 2013; accepted September 25, 2013.
Corresponding author: S. Cavalieri, University of Catania, salvatore.cavalieri@dieei.unict.it

© 2014 ACADEMY PUBLISHER
doi:10.4304/jcp.9.7.1536-1541

model. OPC UA defines information model to represent data and communication services to realize the data exchange.

Inside an OPC UA Server, particular objects, called Nodes, are used to represent any kind of information, including the representation of data instances and the definition of data types. The set of nodes inside an OPC UA Server is called AddressSpace [13]. Each node belongs to exactly one of the following node classes:

- *Base NodeClass*, representing the base class from which all other node classes are derived.
- *Object NodeClass*, representing real-world entities like system components, hardware and software components, or even whole system.
- *ObjectType NodeClass*, holding type definition for objects. OPC UA defines a type model for objects supporting an object-oriented type hierarchy [13] (OPC UA Part 3 & Part 5). OPC UA standard allows extending the standard *ObjectTypes* [13] with additional components. The *BaseObjectType* is the base *ObjectType* and all other *ObjectTypes* shall either directly or indirectly inherit from it.
- *Variable NodeClass*, modeling values of the system. OPC UA distinguishes between data variables (e.g., representing physical values of the process under control) and properties (e.g., meta-data that further describe nodes).
- *VariableType NodeClass*, used to provide type definition for variables. Like *ObjectType*, OPC UA defines a type model for variables supporting an object-oriented type hierarchy [13] (OPC UA Part 3 & Part 5). The *BaseVariableType* is the abstract base type and all other *VariableTypes* shall inherit from it. The *BaseDataVariableType* is a subtype of the *BaseVariableType*. It is used as the type definition whenever there is a *DataVariable* having no more concrete type definition available.
- *DataType NodeClass*, used to provide type definitions for the values of variables. The *BaseDataType* is the abstract base type and all other Data Types shall inherit from it; among them there is the *Structure* data type.
- *ReferenceType NodeClass*, used to define different reference types.
- *Method NodeClass*, modeling callable functions that initiate actions within a server.
- *View NodeClass*, allowing OPC UA Servers to subset the AddressSpace into Views to simplify Client access.

Each Node has several attributes that describe it. Some attribute are common to all nodes (like *NodeId*, *NodeClass* and *DisplayName*). Other attributes are available only for some node classes; considering variables, examples of this kind of attributes are *Value* (representing the value of the variable) and *DataType* (indicating the data type of the Value attribute). Particular relations may be defined between nodes; they are called *References*. *References* are always defined from one node to another. *References* can be defined in a symmetric or asymmetric way and further be divided into hierarchical or non-hierarchical. Hierarchical references can be used to introduce topologies and hierarchies within the model; typical examples of standard OPC UA hierarchical references are *Organizes* (used to introduce a general hierarchy of nodes) and *HasComponent* (used to reference a complex type to its sub nodes). Example of OPC UA non-hierarchical references are *HasEncoding* (specifying the encoding of a structured data type), *HasDescription* (specifying the description within the encoding object) and *HasTypeDefinition* (allowing referencing an object or variable to its type definition).

To promote interoperability of Clients and Servers, the OPC UA AddressSpace is structured hierarchically with the same top levels for all OPC UA Servers [13] (OPC UA Part 1).

Nodes are accessible by Clients using OPC UA *Services* (interfaces and methods). Client and server applications use OPC UA Client and Server Application Programming Interface (API) to exchange data, respectively. OPC UA Client/Server API is an internal interface that isolates the Client/Server application code from an OPC UA Communication Stack. Implementation of the OPC UA Communication Stack is not linked to any specific technology; this allows OPC UA to be mapped to future technologies as necessary, without negating the basic design.

IV. Integrating KNX into OPC UA

Integration of KNX interworking model within OPC UA information model can be realized mapping the following main KNX elements into the address space of OPC UA: Function Blocks (FBs), DataPoints, KNX Addresses and their corresponding type definitions. Indeed, OPC UA standard offers many elements that can be used to perform such translation. In this section, this mapping will be shown. Although the basic ideas come from [5][6][7], a novel approach to map KNX DataPoint Types (DPTs) (based on main and sub number of the corresponding identifiers) will be presented and described in the following.

A. Function Block Modeling

In order to model FBs it is possible to adopt two solutions, as pointed out in [5] and [6]. The first one is to map a FB with an OPC UA complex variable; in this case, DataPoints can be modeled as a part of the complex variable representing the FB. The second option is to use OPC UA complex objects, as described in [5] and realized in [6]; basically, according to this approach, KNX FBs are modeled extending the OPC object type *BaseObjectType*.

The proposal here presented will use this second approach, as the first one seems to show several drawbacks, among which the intrinsic limitation of complex variable to include methods and objects.

B. DataPoint Modeling

As described in [5][6] and [7], a DataPoint of a FB can be modeled as simple OPC UA variable, inside the object type of the FB. In this paper, the authors have adopted this way of modeling but introducing a higher convenient level of abstraction as explained in the following.

© 2014 ACADEMY PUBLISHER

In the earlier approaches [5][6] and [7], KNX DPTs specialized directly the *KNXDataPointType* which was an extension of the OPC *BaseDataVariableType*. In this way, any new definition of a DPT would have required to include from scratch all the properties and attributes inside the XML of the OPC UA Information Model. Figure 1 helps to clarify this statement. It shows the UML diagram for the examples of KNX DPTs used in [5]; it is possible to notice that any new subtype of DPT which specializes the *KNXDataPointType* (in the example we have the *RelativeSetValueControlType* DPT, the *InfoOnOffType* DPT and the *SwitchOnOffType* DPT) has to contain all the attributes qualifying the DPT. Now, let's consider *InfoOnOffType* and *SwitchOnOffType* DPT; we can notice that they are characterized by many common attributes (in fact, they possess the same main number) hence a lot of information carried in their own definitions result redundant.

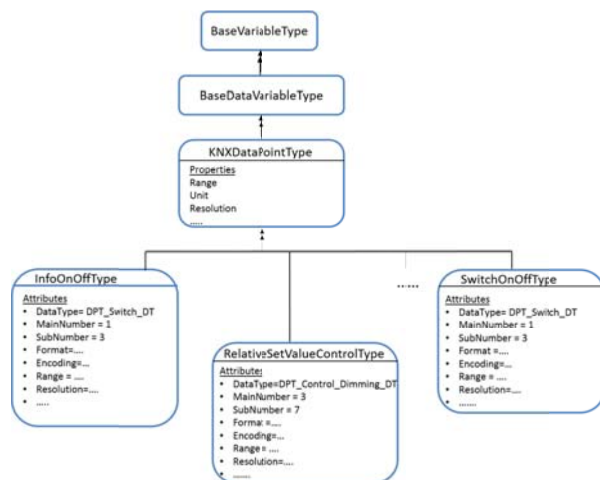

Figure 1. UML Diagram of the KNX DPT presented in [5].

A more efficient way to operate, here proposed by the authors, is possible if we consider that DPTs are characterized (hence uniquely identified) by main and sub numbers, as said in Section II. In particular, the main number defines the format and encoding of DPT while the sub number concerns range and unit [12]. Since DPT with the same main number have the same format and encoding, it is possible to propose a general mapping considering subtypes grouped by main number. Following this approach, the generic *KNXDataPointTypeMNx* variable is defined, where x is the main number of the DPT.

The main advantage brought is that all the information about format and encoding are contained inside this super DPT; hence, there is no need to include such information inside the definition of a new DPT with the same main number which specialize it. In this way, a new DPT (subtype of *KNXDataPointTypeMNx*) will differ only by the sub number that characterizes the information of range and unit.

Thanks to this approach the effort of integrating the KNX interworking model within OPC UA information model results simplified and well structured. In fact, KNX is based on over than 300 DataPoints and, with this approach, they can be grouped within about 60 only super types of *KNXDataPointTypeMNx*. Moreover, further DataPoint types specialization can be postponed and demanded to specific developments, as depending on the need of specific applications.

Figure 2 shows the modified UML diagram of this approach in respect to the one of Figure 1; in particular, the generalized implementation for *KNXDataPointTypeMN1* and *KNXDataPointTypeMN3* is presented. The DPTs *InfoOnOffType*, *SwitchOnOffType* and *RelativeSetValueControlType* shown in Figure 1, are modeled according to our approach in Figure 2.

In particular, *KNXDataPointTypeMN1* is the super type of *InfoOnOffType* and *SwitchOnOffType* while the *KNXDataPointTypeMN3* is the super type of *RelativeSetValueControlType*.

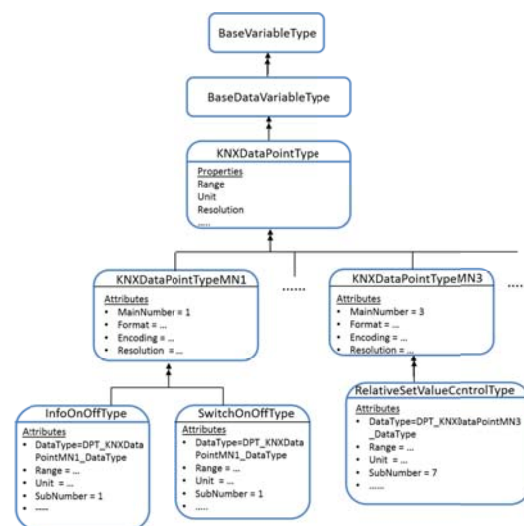

Figure 2. UML Diagram of Figure 1 revised according to the novel approach here presented.

As shown by both Figures 1 and 2, the KNX DPTs feature among attributes, the description of the DataType each DPT refers to; in Figure 2 it's possible to point out the *DPT_KNXDatapointMN1_DataType* (for the DPTs *InfoOnOffType* and *SwitchOnOffType*) and the *DPT_KNXDatapointMN3_DataType* (for the DPT *RelativeSetValueControlType*). This means that in both cases, the definition of those data types must be realized inside OPC UA.

Figure 3 shows the definition of each data type as presented in [5]. *KNXDataPointTypeDataType* is a subtype of *Structure* DataType. According to the previous explanation, DataType of a KNX DPT extends the *KNXDataPointTypeDataType* which, in turn, is a subtype of the OPC Structure.

Figure 4 depicts how the DataTypes of Figure 2 are modeled according to the proposal here presented. Figure 4 shows the specific DataType modeling, for *KNXDataPointTypeMN1* and *KNXDataPointTypeMN3* shown in Figure 2.

Comparing Figures 3 and 4, it's possible to point out the different definition, which is due to the different hypothesis adopted by the authors, as described before.

© 2014 ACADEMY PUBLISHER

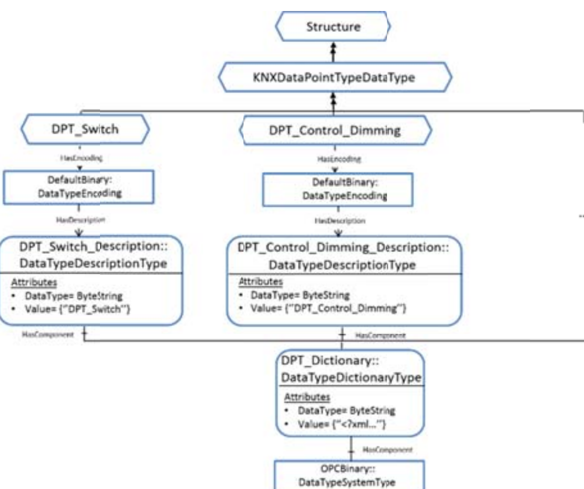

Figure 3. Definition of the Data Type used in [5].

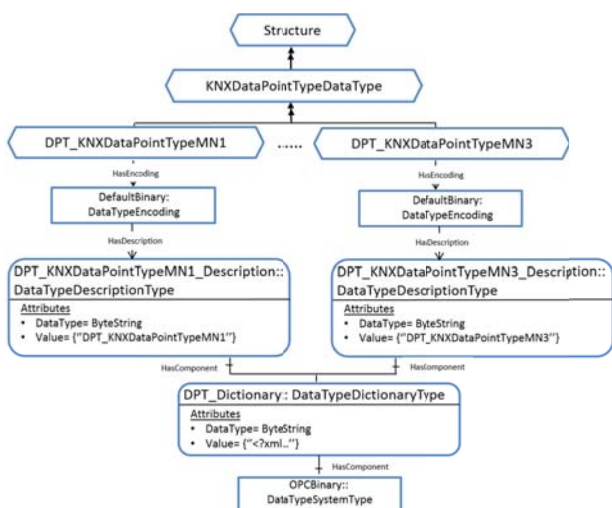

Figure 4. Definition of the Data Type shown in Figure 3, revised according the proposal here presented.

In OPC UA, each DataType can have several *DataTypeEncoding* (for example "Default", "UA Binary" and "XML" encoding) but it is not permitted for two DataTypes to point to the same *DataTypeEncoding*. As shown in the same Figures 3 and 4, a *DataTypeEncoding* object points to one variable of type *DataTypeDescriptionType*. This variable belongs to a *DataTypeDictionary* variable. *DataTypeDictionary* describes a set of DataTypes in sufficient detail to allow clients to parse/interpret variable values both in receiving and sending to/from the server. The *DataTypeDictionary* is represented as a variable of type *DataTypeDictionaryType* in the Address Space of the server and the description about the DataTypes is contained in its *Value* attribute. *DataTypeDictionaries* are complex variables which expose their *DataTypeDescriptions* as variables using *HasComponent* references. A *DataTypeDescription* provides the information necessary to find the formal description of a DataType within the *DataTypeDictionary* [13] (OPC UA Part 5). On the basis of what said, for each new defined DataType a relevant description has to be inserted in the *DataTypeDictionary* variable, as done by the authors.

C. Assignment of a DataPoint to FB

A very important item to point out in the KNX-OPC UA integration is how DataPoints can be linked to a specific FB. As done in [5][6], the authors realize the assignment of a DataPoint to a FB within the OPC UA information model, with the *Reference*.

In particular, three reference types have been defined as subtypes of *HasComponent* type, a built-in type of OPC UA. They are *HasInputDataPoint* and *HasOutputDataPoint* (connecting FBs respectively to input and output DataPoints), and *HasParameter* used to connect a FB to a parameter DP. Further details can be found in [5] and [6].

D. Data Point Address Modeling

In KNX applications, read and write operations of DataPoints and parameters are performed assigning them an address. For example, if a FB needs to send something to another FB, it uses its output DataPoint. The address of this output DataPoint is the same of the corresponding input DataPoint of the receiver FB. Thus DataPoints are linked by common addresses and FBs are linked in a network.

The same approach proposed in [5][6] and [7] has been adopted to model DataPoint address into OPC UA. In particular, association of an address to a DataPoint is made up by using an ad-hoc Reference called *HasKNXAddress*; it has been defined as a subtype of OPC UA *NonHierarchicalReferences*.

V. Software Implementation

The aim of this section is to present the software implementation of the proposed integration among KNX and OPC UA; description will mainly focus on the implementation of DataPoint model according to the proposal presented in Section IV.B.

Description will be done using a simple case study; it will be assumed to integrate a KNX FB into OPC UA information model. The *KNXActuatorType* FB will be defined and its simple functionalities will be pointed out; among other DataPoints, it will include the *KNXAlarmSensor* DataPoint. Its definition and the definition of the relevant DataPointType (*KNXAlarmSensorType*) will be shown in the following, applying the novel hierarchical methodology proposed in this paper (see Section IV.B).

This work has been completely realized by making use of OPC UA 1.0.2 Stack Source Code .NET developed by OPC Foundation [8], avoiding to use any proprietary development framework as done in [5][6] and [7]; this represents another valuable difference from these solutions. Integration has been performed by editing the XML file of the OPC UA Information Model. The resulting Information Model was integrated into a standard OPC UA Server.

In order to perform the integration of the information model, four reference types, are defined to deal with DataPoints. Among them, three references are subtypes of *HasComponent* type, a built-in type of OPC UA; they constitute the reference types which link DataPoints to

© 2014 ACADEMY PUBLISHER

KNX FBs. They are *HasInputDataPoint* and *HasOutputDataPoint* (connecting FBs respectively to input and output DataPoints), and *HasParameter* used to connect a FB to a parameter DP; Section IV.C has dealt with this type of references. The last reference type, *HasKNXAddress,* is a subtype of OPC UA *NonHierarchicalReferences* and it is used to associate one address to one DataPoint, as explained in Section IV.D. All these reference types are symmetrical, namely for each of them it is possible to consider the *InverseName* which represents the reference type as seen by the target node. Figure 5 shows the integration of the previous four KNX Reference types inside the XML file; the same figure shows the names assigned for the InverseName (*IsParameterOf*).

```
<opc:ReferenceType SymbolicName="HasInputDataPoint" BaseType="ua:HasComponent">
   <opc:Description>Reference connecting KNX FB with input DP. </opc:Description>
   <opc:InverseName>IsInputDataPointOf</opc:InverseName>
</opc:ReferenceType>

<opc:ReferenceType SymbolicName="HasOutputDataPoint" BaseType="ua:HasComponent">
   <opc:Description> Reference connecting KNX FB with output DP. </opc:Description>
   <opc:InverseName>IsOutputDataPointOf</opc:InverseName>
</opc:ReferenceType>

<opc:ReferenceType SymbolicName="HasParameter" BaseType="ua:HasComponent">
   <opc:Description>Reference connecting a KNX FB with parameter DP. </opc:Description>
   <opc:InverseName> IsParameterOf </opc:InverseName>
</opc:ReferenceType>

<opc:ReferenceType SymbolicName="HasKNXAddress"
                                   BaseType="ua:NonHierarchicalReferences">
   <opc:Description> Reference connecting a DP with its address. </opc:Description>
   <opc:InverseName> IsParameterOf </opc:InverseName>
</opc:ReferenceType>
```

Figure 5. Implementation of Reference Types.

As said in Section IV.A, a KNX function block is modeled extending the OPC object type *BaseObjectType*. Figure 6 shows the implementation of the simple FB used for this case study; it has been called *KNXActuatorType*.

```
<opc:ObjectType SymbolicName="KNXActuatorType"
                                   BaseType="ua:BaseObjectType">
  <opc:Description>A simple KNX FB used in the case study</opc:Description>
  <opc:Children>
    <opc:Variable SymbolicName="KNXAlarmSensor"
                                   TypeDefinition="KNXAlarmSensorType"/>
    <opc:Variable SymbolicName="SwitchOnOff" TypeDefinition="SwitchOnOffType"/>

 </opc:Children>
 <opc:References>
    <opc:Reference>
       <opc:ReferenceType>HasInputDataPoint</opc:ReferenceType>
         <opc:TargetId>GenericKNXSensorType_KNXAlarmSensor </opc:TargetId>
    </opc:Reference>
    <opc:Reference>
       <opc:ReferenceType>HasOutputDataPoint </opc:ReferenceType>
         <opc:TargetId>GenericKNXSensorType_SwitchOnOff </opc:TargetId>
    </opc:Reference>
  </opc:References>
</opc:ObjectType>
```

Figure 6. Implementation of KNXActuatorType FB.

It has been assumed that this FB receives a digital input and according to its state, it sends an on/off command to an actuator. As shown by Figure 6, inside its definition, it is possible to highlight the input and output DataPoints. One of this DataPoint is called *KNXAlarmSensor*, defined as an instance of the *KNXAlarmSensorType* DPT; it represents the digital input. The other DataPoint is called *SwitchOnOff* and realizes the output of the on/off command. On the basis of what said, Figure 6 points out that *KNXAlarmSensor* is defined as an input DataPoint, through the reference type *HasInputDataPoint*; *SwitchOnOff* is defined as an output DataPoint using the reference type *HasOutputDataPoint*. As it can be seen, references are specified by constructing the Symbolic Id for the components by inserting a "_" between each Symbolic Name. In the following, only the implementation of the *KNXAlarmSensorType* DPT will be given.

As already explained in Section IV.B, according to our proposal of integration (see Figure 2), DPTs modeling is based on the definition of *KNXDataPointType*. As already explained *KNXDataPointType* extends *BaseDataVariableType* and contains all the information about the generic DPT: main and sub number, range, unit and resolution. *KNXDataPointTypeMNx* is a subtype, where "x" corresponds to the main number of the KNX DataPoint type. Considering the simple case study here taken into account, only the use of *KNXDataPointType* with main number 1 will be shown in the following for the definition of *KNXAlarmSensor* DataPoint.

Figure 7 shows the definition of both *KNXDataPointType* and *KNXDataPointTypeMN1*. In the same figure, the definition of *KNXAlarmSensorType* is also codified. It inherits from *KNXDataPointTypeMN1*; as clearly shown in the figure, it has sub number 5 and is characterized by a property of access (*"AccessLevel"*) specifying what operations are allowed by the server level (in this case both read and write). As it can be seen, the DataType of the *KNXAlarmSensorType* is defined as *KNXDatapointTypeMN1DataType*.

```
<opc:VariableType SymbolicName="KNXDataPointType"
                              BaseType="ua:BaseDataVariableType">
   <opc:Children>
       <opc:Property SymbolicName="MainNumber" DataType="ua:Integer" />
       <opc:Property SymbolicName="SubNumber" DataType="ua:Integer" />
       <opc:Property SymbolicName="Range" DataType="ua:Range" ValueRank="Scalar"/>
       <opc:Property SymbolicName="Unit" DataType="ua:Double" ValueRank="Scalar" />
       <opc:Property SymbolicName="Resolution" DataType="ua:Double"
                                                        ValueRank="Scalar"/>
   </opc:Children>
</opc:VariableType>

<opc:VariableType SymbolicName="KNXDataPointTypeMN1"
                              BaseType="KNXDataPointType" MainNumber = "1"/>

<opc:VariableType SymbolicName="KNXAlarmSensorType"
                              BaseType="KNXDataPointTypeMN1"
                              DataType= "KNXDataPointTypeMN1DataType"
                              SubNumber = "5" AccessLevel="ReadWrite"/>
</opc:VariableType>
```

Figure 7. Implementation of KNXAlarmSensorType DPT.

As explained in Section IV.B, the *Structure* DataType has been chosen to model the type of the present value of a DataPoint. Figure 8 shows an example of such implementation, with the definition of the *KNXDataPointMN1TypeDataType* used in Figure 7 inside the implementation of DataPoint. As shown, *KNXDataPointTypeMN1DataType* derives from the *KNXDataPointTypeDataType* that, in turns, derives from the base type *Structure*. As shown in Figure 8, the encoding of *KNXDataPointTypeMN1DataType* (*KNXDataPointTypeMN1Encoding)* is referred by a reference of the type *HasEncoding*. This encoding includes the variable *KNXDataPointMN1Description* of

© 2014 ACADEMY PUBLISHER

type *DataTypeDescriptionType* whose value field is *KNXDataPointTypeMN1*. It is possible to access the semantics within the dictionary referring to this field. The description is linked to the encoding through the OPC UA reference of type *HasDescription*.

```
<opc:DataType SymbolicName="KNXDataPointTypeDataType" BaseType="ua:Structure" />
<opc:DataType SymbolicName="KNXDataPointTypeMN1DataType"
                                   BaseType="KNXDataPointTypeDataType "/>
  <opc:References>
    <opc:Reference>
      <opc:ReferenceType>ua:HasEncoding</opc:ReferenceType>
        <opc:TargetId>KNXDataPointTypeMN1Encoding </opc:TargetId>
    </opc:Reference>
  <opc:References>
</opc:DataType>

<opc:Object SymbolicName="KNXDataPointTypeMN1Encoding"
                                   TypeDefinition="ua:DataTypeEncodingType">
  <opc:Children>
    <opc:Variable SymbolicName="KNXDataPointTypeMN1Description"
        BaseType="ua:DataTypeDescriptionType" Value="KNXDataPointTypeMN1"/>
  </opc:Children>
  <opc:References>
    <opc:Reference>
      <opc:ReferenceType>ua:HasDescription</opc:ReferenceType>
        <opc:TargetId>
            KNXDataPointTypeMN1Encoding_KNXDataPointTypeMN1Description
        </opc:TargetId>
      </opc:Reference>
  </opc:References>
</opc:Object>
```

Figure 8. Implementation of the KNXDataPointDataType by using the *Structure* Data Type of OPC UA.

Figure 9 shows the last part of the XML implementation related to the description of the portion of dictionary relevant to the case study. The field "*Value*" contains the name given to the dictionary, "DPTXML.xml". The first part of the xml file contains the references to the descriptions defined within the encodings of *KNXDataPointMN1*. The last part is the definition of the OPC Binary type system that contains the *DPTDictionary* (used in the upper part of the same figure) and referred by a *HasComponent* reference.

```
<opc:Variable SymbolicName="DPTDictionary"
          TypeDefinition="ua:DataTypeDictionaryType" DataType="ua:ByteString "
          Value="DPTXML.xml">
 <opc:References>
   <opc:Reference>
    <opc:ReferenceType>ua:HasComponent</opc:ReferenceType>
     <opc:TargetId>KNXDataPointTypeMN1Encoding_KNXDataPointTypeMN1Description
   </opc:TargetId>
  </opc:Reference>
      <opc:ReferenceType>ua:HasComponent</opc:ReferenceType>
 </opc:References>
</opc:Variable>

<opc:Object SymbolicName="OPCBinary" TypeDefinition="ua:DataTypeSystemType">
  <opc:BrowseName>OPCBinary</opc:BrowseName>
    <opc:References>
      <opc:Reference>
        <opc:ReferenceType>ua:HasComponent</opc:ReferenceType>
        <opc:TargetId>DPTDictionary</opc:TargetId>
      </opc:Reference>
    </opc:References>
</opc:Object>
```

Figure 9. Definition of the DPTDictionary and of the OPC Binary Data Type.

FINAL REMARKS

This paper focuses on the problem to improve interoperability in all the levels of a communication architecture in an automation environment; here equipment are based on different technologies and communication protocols avoiding to realize full interoperability, which may be achieved through the adoption of expensive proprietary mapping.

Previous literature has presented some proposals aiming to use OPC Unified Architecture (OPC UA) to provide a cohesive, secure and reliable cross platform framework for interoperability. Some of the available papers apply this approach to the KNX interworking model.

Starting from these proposals, presented in [5][6] and [7], in this paper the authors propose further improvements always considering KNX interworking model integration.

The main difference with the current literature is due to the different approach adopted to model DataPoints of a FB, as described in the paper. This approach may reduce the effort of integration of the KNX interworking model within OPC UA information model.

REFERENCES

[1] Cenelec EN 50090, *Home and Building Electronic System*, 2005.

[2] Cenelec EN 13321–1, *Open Data Communication in Building Automation, Controls and Building Management*, 2012.

[3] ISO/IEC 1454–3, *HSE Architecture*, 2006.

[4] W. Mahnke, S. Leitner, and M. Damm, *OPC Unified Architecture*, Springer, 2009.

[5] W. Granzer, W. Kastner, and P. Furtak, "KNX and OPC UA", *Proceedings of Konnex Scientific Conference*, November 2010.

[6] P. Ruß, *KNX for OPC UA*, Technical Report 183/1-158, A-Lab @ Automation Systems Group, TU Vienna, November 2011, available at https://www.auto.tuwien.ac.at/bib/pdf_TR/TR0158.pdf

[7] W. Granzer, and W. Kastner, "Information modelling in heterogeneous Building Automation Systems", *Proceedings of 9th IEEE International Workshop on Factory Communication Systems (WFCS)*, May 2012.

[8] www.opcfoundation.org

[9] S. Cavalieri, "Modelling and analysing congestion in KNXnet/IP", *Computer Standards & Interfaces*, Volume 34, Issue 3, March 2012, pp. 305–313.

[10] KNX Association, *System Specifications, Communication: Data Link Layer General*, Chapter 3/3/2, 2009.

[11] KNX Association, *System Specifications, Communication: Application Layer*, Chapter 3/3/7, 2010.

[12] KNX Association, *System Specifications, Interworking: Datapoint Types*, Chapter 3/7/2, 2010.

[13] OPC Foundation, *OPC Unified Architecture Specification*, Parts 1-11, 2009.

[14] S. Cavalieri, "Evaluating Overheads Introduced by OPC UA Specification", *Human–Computer Systems Interaction: Backgrounds and Applications 2, Advances in Intelligent and Soft Computing*, Springer, Vol. 98, 2012, pp. 201-221.

© 2014 ACADEMY PUBLISHER