Article

Insights into Mapping Solutions Based on OPC UA Information Model Applied to the Industry 4.0 Asset Administration Shell

Salvatore Cavalieri * and Marco Giuseppe Salafia

Department of Electrical Electronic and Computer Engineering, University of Catania, 95125 Catania, Italy; marcogiuseppe.salafia@unict.it

* Correspondence: salvatore.cavalieri@unict.it; Tel.: +39-095-738-2362

Received: 29 March 2020; Accepted: 13 April 2020; Published: 14 April 2020

Abstract: In the context of Industry 4.0, lot of effort is being put to achieve interoperability among industrial applications. As the definition and adoption of communication standards are of paramount importance for the realization of interoperability, during the last few years different organizations have developed reference architectures to align standards in the context of the fourth industrial revolution. One of the main examples is the reference architecture model for Industry 4.0, which defines the asset administration shell as the corner stone of the interoperability between applications managing manufacturing systems. Inside Industry 4.0 there is also so much interest behind the standard open platform communications unified architecture (OPC UA), which is listed as the one recommendation for realizing the communication layer of the reference architecture model. The contribution of this paper is to give some insights behind modelling techniques that should be adopted during the definition of OPC UA Information Model exposing information of the very recent metamodel defined for the asset administration shell. All the general rationales and solutions here provided are compared with the current OPC UA-based existing representation of asset administration shell provided by literature. Specifically, differences will be pointed out giving to the reader advantages and disadvantages behind each solution.

Keywords: interoperability; mapping; industry 4.0; OPC UA; asset administration shell

1. Introduction

The very recent fourth industrial revolution, known with the name of Industry 4.0, aims to create more flexible and innovative products and services leading to new added-value business models [1,2].

In the context of Industry 4.0, lot of effort is being put to achieve full integration of the industrial applications. According to the Industry 4.0 vision, a variety of areas related to manufacturing, security, and machine communication, among others, need to interoperate and align their respective information models. As the definition and adoption of communication standards are of paramount importance for the realization of interoperability, during the last few years, different organizations have developed reference architectures to align standards in the context of the fourth industrial revolution. One of the main examples is the "Reference Architecture Model for Industry 4.0 (RAMI 4.0)" [3].

In order to fulfil the main requirements of Industry 4.0, especially in terms of interoperability, the concept of the asset administration shell (AAS) has been defined in RAMI 4.0. The AAS is intended to provide digital representations of all information being available about and from an asset, which can be a hardware system or a software component. For this reason, in November 2018, Plattform Industrie 4.0 (www.plattform-i40.de) released the first draft of the document "Details of the Asset Administration Shell" [4] defining the so-called AAS metamodel; it is mainly aimed to define internal structure of AAS

in terms of digital information available for an asset. In November 2019, the version 2.0 of the same document has been released [5]. AAS is not just a set of structured information about assets but offers service interfaces to access properties and functions provided by the physical assets [6]. In RAMI 4.0, the conjunction of the physical asset and its AAS is referred as I4.0 Component. These components can exchange information in a uniform manner since all the different aspects of the implementation and communication of the assets are abstracted by means of the AASs.

In the context of interoperability inside Industry 4.0, there is also so much interest behind the standard OPC UA (open platform communications unified architecture) [7]. OPC UA plays an important role in current industry environments [8] and it is considered the most accepted protocol which harmonizes the machine to machine (M2M) interaction [9]. During these last years, OPC UA has proven to be an effective communications middleware mainly in industrial applications [10]. Due to its powerful functionalities, OPC UA is one of the main candidates to lead the standardization and systems integration for present and future frameworks [10]. In particular, OPC UA is listed as the one recommendation for realizing the communication layer of RAMI 4.0 [11]. Furthermore, it has been identified as one candidate to build I4.0 Component interface and an important means to provide information models of assets.

The standard OPC UA is based on both client/server and publish/subscribe communication models and provides a semantically enriched information model in order to represent data. Literature provides several publications taking advantage in terms of interoperability of OPC UA information model to structure and expose information coming from different domains of interest. Several papers describe how OPC UA information models for specific domains can be generated starting from the information models belonging to these domains. Some of the most relevant works include the mapping between IEC 61850 and CIM to OPC UA information models (e.g., [12–16]). There also exist approaches for mapping and transformation of Unified Modeling Language (UML) to OPC UA (e.g., [17,18]). In [19], analogies between IEC 62714 (AutomationML) and OPC UA are examined in order to simplify the creation of OPC UA information model based on already existing AutomationML (AML) data. Reiswich and Fay in [20] discuss how it is possible taking advantage of OPC UA information model properties to improve the work situation of operators and engineers. In [21], the software model of IEC 61131-3 for programmable logic controller (PLC) is translated into an OPC UA information model discussing also the advantage of adopting OPC UA for a secure data exchange. Several companion specifications dealing with the mapping of specific domain into OPC UA information models, are also available. OPC UA/PLC open companion specification [22] deals with modelling PLCs. Another example is the OPC UA/AML companion specification [23], which describes the transformation between AML and OPC UA information models. Another one, which is important for the manufacturing domain, is the OPC UA/MTConnect companion specification [24]. Finally, it is important to recall the existence of OPC UA/IEC61850 companion specification for electrical substation automation systems [25].

On account of what just pointed out and due to the important role played by AAS and OPC UA inside Industry 4.0, an OPC UA information model may be defined to structure and expose the current AAS metamodel [5]. This may led to several advantages, among which the possibility to exchange AAS digital information between industrial applications through the OPC UA communication system. To the best of authors' knowledge, some activities about this issue already exist. Document [5] provides a proposal of mapping the AAS metamodel into several technologies, including OPC UA. Mapping of AAS metamodel into OPC UA information model is also under consideration by the ZVEI, VDMA and OPC Foundation joint working group [26], aimed to the definition of a draft of a new OPC UA Companion Specification for AAS; no outcomes were still produced by this group.

The contribution of this paper is to give some insights behind modelling techniques that should be adopted during the definition of OPC UA information models exposing information relevant to the Industry 4.0 specific domains of interest. The current AAS metamodel [5] is considered in this work. All the general rationales here provided are compared with the only other mapping proposal between

AAS metamodel and OPC UA available in literature (contained in [5] as said before), when needed. Specifically, differences will be pointed out giving to the reader pros and cons behind each solution.

It is important to point out that the work presented in this paper must not be considered an alternative to the document [5] or to the ongoing work carried on by the ZVEI, VDMA and OPC Foundation joint working group or any activities done by other research groups. Since AAS metamodel, and thus its mapping in OPC UA, is continuously developed and improved, authors' aim is only to provide reasoning about mapping choices that may be considered for future versions of such mapping solutions.

This paper is an extended version of the earlier publication by the same authors [27], which reported very preliminary results of their studies and has been published before the document [5]. This last information is given to the reader only to point out that many reasoning exposed in this paper have been conceived before the OPC UA information model presented in [5].

The paper is structured as it follows. In Section 2 a background of the AAS metamodel is provided, describing the fundamental entities specified in [5]. In Section 3 the OPC UA Information Model is presented. Section 4 gives an overview about general techniques to be adopted to map specific domains of interest into OPC UA Information Model. In Section 5, the general mapping techniques shown in the previous section will be exemplified for the AAS metamodel, providing rationales behind mapping decisions and comparisons with actual solutions given in [5]. Section 6 points out how adoption of new OPC UA mechanisms defined in recent amendments may led to enhancements in the mapping process based on OPC UA information model. In Section 7, a case study will be presented in order to help the reader to better understand the reasoning discussed in the paper. Finally, Section 8 will point out the software implementations made by the authors, some of which are open source and available on GitHub.

2. The Asset Administration Shell Metamodel

The internal structure of AAS was described in a very high level of abstraction in [3] and in [28]; its definition takes in account lot of requirements summarized in [28]. As shown by Figure 1, AAS consists of a header and a body; the former contains information about identification of the AAS and the asset it represents, whilst the latter contains a certain number of submodels. Submodels represent different aspects of the concerned asset (e.g., engineering, communication, and drilling). Standardized submodels defining functions and properties are foreseen to represent each aspect [29].

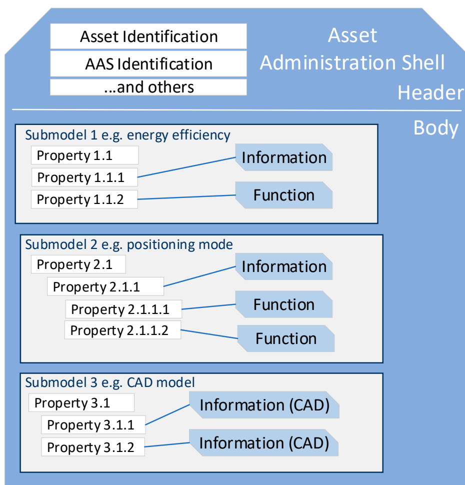

Figure 1. Internal structure of an asset administration shell (AAS).

Every sub model contains a hierarchy of Properties related to the domain of the sub model itself. The format of the property descriptions should follow a standard, like IEC 61360 [30]. Sub models are

composed of Sub model elements, each of which is an abstract class for all the entities in the metamodel that can be aggregated by a sub model.

In [4,5] the AAS metamodel is introduced. These documents describe how information is structured inside an AAS. In the remainder, details about the latest AAS metamodel will be provided giving a description of its fundamental entities [5].

Entities in the AAS metamodel can inherit from more than one common class (multiple inheritance). Common classes are abstract classes used to describe aspects shared by metamodel entities; they are Identifiable, Referable, HasKind, HasSemantics, and HasDataSpecification. Due to lack of space, only the most important attributes of common classes and AAS entities will be described in the following. For a complete description, the reader should refer to [5]. From now on the name of a common class will be used for a particular metamodel entity if it inherits from that common class. For example, "HasSemantics entity" will refer to an entity inheriting from the HasSemantics common class; saying that an asset is either "Identifiable" or an "Identifiable entity" means that the asset entity inherits from the identifiable common class.

2.1. *Identifiable and Referable*

In the digitalization process of an asset, everything should be unambiguously identified: parts, products, people, software and services. But in order to achieve interoperability, relationships between entities shall be identifiable too. For this reason, the AAS metamodel makes a distinction between elements that are identifiable, referable or none of both.

An identifiable entity can be uniquely identified by means of a globally unique identifier. This is a very important feature for an entity, because this makes possible to refer the entity in any context. Table 1 summarizes the attributes of identifiable entities. Identifier is a structured type and is composed of a string field (id), and a second field (idType) of type IdentifierType; according to [5] IdentifierType is an enumerative type made up the following elements: IRDI, URI and Custom. Values of identifier type are used inside the attribute identification of identifiable entities, as shown in Table 1.

Table 1. Attributes of identifiable.

Attribute	Description	Type
administration	Administrative information	AdministrativeInformation
identification (Mandatory)	Global unique identification	Identifier

Table 2 summarizes the attributes of referable entities. A referable entity provides a short identifier (*idShort*) that is unique only in the context of its name space. The name space for a referable entity is defined as its parent element that is either referable or identifiable. Identifiable entities are also referable but the vice versa is not true.

Table 2. Attributes of Referable.

Attribute	Description	Type
idShort	Identifying string of the entity within its name space	String
category	Additional meta information about the class of the element, affecting the expected existence of certain attributes	String
description	Description or comment of the element	String
parent	Reference to the next referable parent element	Reference

2.2. *Reference*

The reference entity is needed in order to establish relationships between entities composing the AAS. References can also be used to refer entities not defined internally the AAS but in some external source. An entity should be at least referable to be pointed by a reference.

The AAS reference entity features the attribute key, which is logically structured as an ordered list of keys where each element refers an entity by means of its identifier. The structure of this key list resembles an URI structure, where the first key refers to the root element and every following key identifies the next element in the hierarchy leading to the referred element, identified by the last key of the list.

Each key in the list belongs to a structured type named Key, featuring several attributes. The mandatory attributes are local, type, value and idType. The attribute local specifies whether the referred element is local to the AAS or not. The attribute type specifies the class name of the referenced entity; its value belongs to a custom type named KeyElements, which is an enumeration consisting of all the names of the entities in the metamodel (e.g., property, asset, submodel). The attribute value is a string containing the identifier of the entity referred by the key. The attribute idType describes the kind of identifier used in attribute value; its value is of type KeyType which is an enumeration of all the different kinds allowed for both global and local identifiers, i.e., IRDI, IRI, Custom, idShort and FragmentId (see [5] for the relevant definitions).

2.3. HasKind

The common class HasKind identifies all those entities that can have the double nature of template and instance; the concept of template represents the concept of class in object-oriented programming (OOP), so that an entity instance derives from an entity template in the same way as an object instance derives from a class in OOP.

Templates define common features for all its instances. An entity that can be either a template or an instance is referred in the metamodel as a HasKind entity. It is featured by a unique optional attribute, named kind, which can take either the value “Template” or “Instance”.

2.4. HasSemantics

HasSemantics entity is whatever AAS entity that can be described by means of a concept. A HasSemantics entity owns a reference to another external entity that describes its meaning in a proper manner. To this aim, the HasSemantics element has only one optional attribute, the semanticId, which is a reference to the semantic definition of the element.

2.5. HasDataSpecification and DataSpecification

One of the requirements for the AAS mandates that the definition of additional attributes (e.g., manufacturer specific) for some entities must be possible. An entity that allows its instances to contain additional attributes to those already defined in the entity itself is identified as HasDataSpecification entity. Such an entity contains one or more References to so-called data specification templates (DST), which are used to define the additional attributes. The only attribute inherited by the HasDataSpecification common class is the optional hasDataSpecification, which contains References to the DSTs eventually used.

Even if [5] specifies that DST does not belong explicitly to the metamodel, its internal structure is described using an entity named DataSpecification; it is identifiable, so that its identifier can be used inside references. It consists of an entity named DataSpecificationContent containing the definition of the additional attributes. In other words, if a particular instance of a metamodel class features some additional attributes not defined in the class itself, such instance shall refer a DataSpecification defining such additional attributes in order to declare the presence of these last.

2.6. AssetAdministrationShell

The main element of the entire AAS metamodel is represented by the AssetAdministrationShell entity. This entity is both Identifiable and HasDataSpecification. It provides more attributes based on how an AAS is structured [28]; Table 3 summarizes some of them.

Table 3. Attributes of the AssetAdministrationShell.

Attribute	Description	Type
derivedFrom	A Reference to the AAS the current AAS was derived from	Reference
asset (Mandatory)	A Reference to the Asset entity	Reference
submodel	References to the Submodels	Reference
conceptDictionary	One or more Concept Dictionary entities	ConceptDictionary

The derivedFrom attribute is used to establish a relationship between two AASs that are derived from each other; it contains a Reference. In case of an AAS representing an asset instance, this reference points to the AAS representing the corresponding asset type or another asset instance it was derived from. The same holds for AAS of an asset type as types can also be derived from other types.

The other attributes shown in Table 3 (i.e., asset, submodel and conceptDictionary) refer to the Asset, Submodel and ConceptDictionary entities described in the following subsections, respectively.

2.7. Asset

The Asset entity contains all metadata of an asset represented by an AAS. This entity is Identifiable and HasDataSpecification. Usually it owns a reference to a Submodel entity describing identification aspect of the asset itself, but this is not mandatory.

It features an attribute kind that specifies whether the asset is a type or an instance, in accordance to the asset life cycle as said in [31]. This attribute is used to maintain the relationship between an asset type and its asset instances for their whole lifecycle; in this way, updates on the AAS of the asset type can be reflected on the AASs of the respective asset instances.

2.8. Submodel and SubmodelElement

The Submodel entity defines a specific aspect of the asset represented by the AAS. It is used to structure the AAS into distinguishable parts, organizing related data and functionalities of a domain or subject. Submodels can become standardized, but at the time of writing this paper, no standard Submodels have been released.

This entity is Identifiable, HasKind, HasDataSpecification, and HasSemantics. In case of a Submodel with kind = "Instance", the semanticId attribute may refer to another Submodel entity with kind = "Template".

Submodel aggregates SubmodelElements that are related to the same aspect of the asset identified by the Submodel itself. For this reason, the Submodel entity defines another additional attribute, named submodelElement, which is a composition of zero or more SubmodelElements. A SubmodelElement entity is suitable for the description and differentiation of assets. The SubmodelElement entity is Referable, HasKind, HasDataSpecification and HasSemantics. All the SubmodelElements of a Submodel with kind = "Template" are in turn SubmodelElement templates (i.e., kind = "Template").

2.9. DataElement and Property

DataElement is a SubmodelElement that is no further composed out of other SubmodelElements.

A Property is a DataElement that is made up by the additional attributes shown in Table 4. The attributes value and valueType are the most important; the latter specifies which kind of data value is contained in the former. This information is necessary to decode such a value.

Table 4. Attributes of Property.

Attribute	Description	Type
value	The value of the Property instance	ValueDataType
valueType	Data type of the value	DataTypeDef
valueId	A Reference to the global unique id of a coded value	Reference

2.10. ConceptDictionary and ConceptDescription

One of the core entities of the AAS metamodel to achieve interoperability is ConceptDescription; it is used to define the semantics of entities inside the AAS metamodel. The ConceptDescription entity is Identifiable and HasDataSpecification. Every element in AAS that is HasSemantics should have its semantics described by a ConceptDescription, unless a more specific solution is adopted.

The entity ConceptDictionary represents a collection of ConceptDescription instances. ConceptDictionary is referable and it defines an additional attribute named conceptDescription. Such attribute is a composition of AAS References pointing to ConceptDescription instances.

Typically, a concept description dictionary of an AAS contains only concept descriptions of elements used within submodels of the AAS. In certain scenarios, the concept dictionary may contain copies of property definitions coming from external standards. In this case, a semantic definition to the external standard shall be added; for this reason, ConceptDescription defines the additional optional attribute isCaseOf, which represents a global reference to an external definition the concept is compatible with or was derived from. For instance, if the semantics of a Property in the AAS is defined in eCl@ss [32], a ConceptDescription instance must be created and its attribute isCaseOf must be filled with a reference pointing to the relevant eCl@ss ID [5]. ConceptDescription should follow a standardized template to describe a concept. The only templates available in the metamodel are used to define both semantics of Properties according IEC 61360 [30] and physical unit of measurement.

3. OPC UA Information Model

The OPC UA Information Model provides a standard way for servers to expose information to clients. The set of information is maintained through OPC UA Nodes grouped together to compose the so-called OPC UA AddressSpace [7,33]. The OPC UA Information Model is based on OOP, so that some nodes representing instances inherit from other nodes defining types; multiple inheritance is not recommended in OPC UA even though the specification does not restrict type hierarchies to single inheritance [34].

Each OPC UA Node belongs to a class named NodeClass, some of which will be described in the following. Among the available NodeClasses, there is the variable NodeClass which is used to model data. This NodeClass features an attribute named Value, containing the data, and an attribute named DataType, specifying the type of the content of the attribute Value. DataType may be Built-in, Enumeration or Structured. Arrays of elements belonging to Built-in, Enumeration and Structured DataTypes are also allowed. Two types of Variables are defined: Properties and DataVariables; differences between them and the relevant data they can model, will be discussed in Section 4.

Another NodeClass is the Object. It is a container for other OPC UA Objects and Variables. For example, since the Object Node does not feature an attribute that can be used to contain a data value (e.g., the temperature value of a sensor), an OPC UA DataVariable Node is used as a component of an OPC UA Object Node to represent data associated to that Object.

OPC UA includes NodeClasses defining types. ObjectType NodeClass is used to define types for OPC UA Objects; Objects are instances of ObjectTypes in the sense that they inherit the Nodes beneath their ObjectTypes. OPC UA defines the BaseObjectType which all the ObjectTypes must be extended from. OPC UA already defines several standard ObjectTypes derived from BaseObjectType. An example of ObjectType is FolderType whose instance, named Folder, is an Object organizing the AddressSpace into a hierarchy of OPC UA Nodes; it represents the root node of a subtree. VariableType is another NodeClass used to provide type definition for Variables. OPC UA defines the BaseVariableType which all the VariableTypes must be extended from. Among the standard VariableTypes derived from BaseVariableType, there are the DataVariableType and the PropertyType. The former is used to define a DataVariable Node, whilst the latter defines a Property Node.

Relationships may be defined between OPC UA Nodes; they are called References. The ReferenceType NodeClass is used to define different semantics for References. References may be classified in two different main categories: Hierarchical and NonHierarchical. Among the Hierarchical References, the following

ones will be used in the paper: HasComponent, Organizes and HasProperty. The HasComponent Reference allows to specify that an OPC UA Object contains another OPC UA Object or OPC UA DataVariable. Organizes Reference allows to organize OPC UA Nodes inside a Folder. The HasProperty Reference is used to link a source OPC UA Node to a target OPC UA Property; the semantics is that the source Node features a property described by the target Node.

Among the NonHierarchical References there are the HasTypeDefinition, HasSubtype, and HasModellingRule. The first one is used to bind an OPC UA Object or Variable to its ObjectType or VariableType, respectively. HasSubtype Reference expresses a subtype relationship between types.

For each OPC UA type, the relevant instances may have some mandatory elements (e.g., a particular Object as component), whilst other elements may be optional (e.g., a certain Property). HasModellingRule Reference allows to point out this kind of information for each OPC UA type. For each Variable or Object (henceforward called InstanceDeclaration) referenced by an OPC UA type Node, a HasModellingRule Reference points to a ModellingRule Object as target Node. A ModellingRule associated to an InstanceDeclaration specifies whether a copy of such InstanceDeclaration must be present or not in every instance of an OPC UA type Node. A ModellingRule Mandatory for a specific InstanceDeclaration specifies that instances of the OPC UA type must have that InstanceDeclaration. A ModellingRule Optional, instead, specifies that instances of the OPC UA type may have that InstanceDeclaration, but it is not mandatory.

OPC UA defines standard graphical representation for both Nodes and References [33]. Some of them are summarized by Tables 5 and 6.

Table 5. Graphical representation of some open platform communications unified architecture (OPC UA) nodes.

OPC UA Node	Standard Graphical Representation
DataType	DataType Name
ObjectType (e.g., FolderType)	ObjectType Name
Object The relevant ObjectType is specified on the top	*ObjectType Name* Object Name
Variable (DataVariable/Property) The VariableType is specified on the top	*VariableType Name* Variable Name Attributes
ModellingRule Object/HasModellingRule Reference They are both represented inside the source InstanceDeclaration Node	*ObjectType Name* Object Name [Mandatory] · *ObjectType Name* Object Name [Optional] · *VariableType Name* Variable Name [Mandatory] Attributes · *VariableType Name* Variable Name [Optional] Attributes

Table 6. Graphical representation of some OPC UA References.

OPC UA Reference	Standard Graphical Representation
HasTypeDefinition	
Hierarchical (e.g., Organizes)	
NonHierarchical	
HasComponent	
HasProperty	
HasSubtype	

Very recently, the OPCFoundation released an amendment introducing a new feature in Address Space model called Interface [34]. An Interface is an ObjectType representing a generic feature that can be used by different Objects or ObjectTypes. HasInterface is a new NonHierarchical ReferenceType; an Object may have more HasInterface References connected to different Interfaces. When an Object references an Interface by means of a HasInterface Reference, it inherits all the InstanceDeclarations exposed by the Interface, following the same rules used for an Object that inherits all InstanceDeclarations exposed by its ObjectType. More details may be achieved in [34].

In the remainders of the paper, to avoid confusion between Reference in AAS metamodel and Reference in OPC UA, the suffix AAS or OPC UA is added when the context requires to do so. Furthermore, names of Objects will be written between double quotes, in order to be easier distinguishable.

4. Common Practices in the Definition of OPC UA Information Model

Introduction pointed out that current literature provides several publications taking advantage in terms of interoperability of OPC UA Information Model to structure and expose information coming from different domains of interest. In general, the definition of an OPC UA Information Model requires a phase where all the requirements of the original domain of interest are collected and compared with the standard elements of the OPC UA Information Model in order to find the best mapping between them. Often, this is not an easy task because some concepts from the source domain cannot be directly mapped into OPC UA; in these cases, the definition of new element types extending the original OPC UA elements must be realized. The aim of this section is to point out the common practices adopted when OPC UA Information Model is used to model a generic system; this analysis will be carried on for each of the main OPC UA Information Model elements.

4.1. Variables and DataTypes

Nodes of Variable NodeClass are usually used to represent data. For instance, a Variable may represent the measurement of a temperature sensor or the engineering unit of the measured temperature. As the reader can notice, in both cases we are speaking about data associated to the same device, but the relevant semantics are quite different. In the former case, the Variable represents a value produced by the temperature sensor. In the latter case, instead, the Variable represents a characteristic of the same device. To distinguish these cases, OPC UA defines two main VariableTypes that Variables must inherit from: DataVariableType and PropertyType. As already explained in Section 3, we will refer to variables inheriting to these two VariableTypes as DataVariable and Property, respectively. As pointed out in [7], it is not always so easy to decide when a DataVariable or a Property should be used when modelling data. In general, DataVariable may be chosen to represent data associated to an Object, whereas a Property may be used to represent some characteristic of a Node that usually cannot be described by means of the attributes of the Node itself.

As said in Section 3, the data type of the value contained in a Variable is described by the attribute DataType. Usually variables may contain simple value like integers or strings, but most of the time, during the development of an OPC UA Information Model, the definition of domain-specific data type is required. OPC UA provides the NodeClass DataType for this purpose. If the Information Model requires user defined types, like structures, enumerations and arrays, proper types must be defined in OPC UA information model in order to represent such user-defined types. Section 3 pointed out that Built-in, Enumeration or Structured OPC UA DataTypes are available to this aim. Arrays of elements belonging to Built-in, Enumeration and Structured DataTypes are also allowed.

It is worth noting that, in case of a structured value, the adoption of a Structured DataType is not the only solution. In fact, a structured value can be modelled as a complex Object featuring several Variables Nodes as components (i.e., linked to the Object by HasComponent References), each of which representing a field of the structured value. Both solutions are valid to represent a structured value, however there are pros and cons for each solution. Briefly, using a Structured DataType is possible to easily access all data at once, whereas using an Object with components requires that each variable component is accessed one by one. In other words, structured DataType provides an implicit transaction context during the information access, whereas Object does not and it must be explicitly managed. On the other hand, using Object in order to access individual data of a structured value it is easier than accessing a value belonging to structured DataType. The latter involves an overhead for this kind of operation because all the structured value must be read to retrieve the value of a field. More details can be found in [7].

4.2. Object and ObjectTypes

The OPC UA NodeClass Object is used to represent entities like entire systems, system components, real-world and software objects. For instance, an Object inheriting from an ObjectType modelling a type of device (e.g., engine type, motor type) represents a physical device whose complexity is modelled by all the other nodes connected to it by means of hierarchical references (e.g., HasProperty, HasComponent). In general, the meanings that an Object can assume are unlimited; the important thing is understanding how Object and its ObjectType are the building blocks of a well-organized AddressSpace. As a consequence, for each entity that must be represented in the AddressSpace, a relevant ObjectType should be properly defined; all the objects belonging to this ObjectType represent instances of the particular entity represented by the ObjectType.

When modelling a system, representation of attributes belonging to a particular entity occurs. First of all, the modelling of the attributes must be realized during the definition of ObjectTypes representing the entities. The two most frequent cases consist of attributes containing data values or containing some other complex object. An attribute containing value can be mapped as OPC UA Property (connected with a HasProperty Reference) or DataVariable (connected with a HasComponent Reference) depending on the consideration made in the previous subsection. An attribute containing a complex object, instead, may be mapped as OPC UA Object component, which must be connected to the OPC UA ObjectType by a HasComponent Reference; this is legit as this reference represents a part-of relationship and attributes may be considered parts of an entity.

OPC UA Objects may also be used to organize the AddressSpace [7], as explained in the following subsection.

4.3. AddressSpace Organisation

When OPC UA Information Model is used to model a system made up by several entities, a good practice consists of defining an entry point to all the relevant Nodes. Usually, a Folder Object contained in the standard “Objects” folder [7] is used as an entry point to the subset of the AddressSpace relevant to the system modelled. This Folder Object will contain all the OPC UA Nodes modelling the entities present in the system. All these nodes may be organized in different ways according to the needs to be fulfilled, as explained in the following.

Let us assume to model entities linked by a hierarchical relationship. Hierarchical relationship may occur when entities are organized in a way that resembles the same organization existing between folder and the relevant content in a generic file system. In this case, this relationship may be modelled using a folder object modelling the topmost entity and connecting it to the nodes modelling the other entities, by means of organizes references. If the hierarchical relationship among the entities resembles an aggregation, particular OPC UA hierarchical references, like HasComponent and HasProperty, may be used to connect the OPC UA nodes modelling the original entities.

Let us consider now a system to model where a relationship exists between two entities belonging to different hierarchies. In this case, a common modelling practice in the organization of OPC UA information model involves the use of non-hierarchical OPC UA references. For example, an object belonging to a Folder Object may be linked to the relevant ObjectType (usually belonging to the standard "Types" Folder) by a HasTypeDefinition reference. In general, it is possible to say that non-hierarchical references organizes the AddressSpace from a semantics point of view [7]. Usually ad-hoc non-hierarchical ReferenceTypes must be defined in order to better represent the kind of relationships between entities to be modelled.

5. Mapping AAS Metamodel into OPC UA Information Model

The common practices highlighted in the previous section will be exemplified considering the mapping of the AAS metamodel into the OPC UA information model. The authors will provide reasoning behind the main decisions to be taken in the mapping, pointing out pros and cons when different strategies can be adopted. Furthermore, a comparison with the mapping solutions taken in [5] will be done when different approaches can be adopted for the same scenario.

In Section 5.1, mapping solutions for entities and attributes of the AAS metamodel are discussed. Section 5.2 contains insights about how to structure the AddressSpace of an OPC UA Server in order to expose AASs. In Section 5.3, the referencing mechanism of the AAS metamodel is analyzed and strategies about mapping AAS References in OPC UA are provided and discussed.

5.1. Mapping AAS Entities

One of the first decisions to be taken is deciding which OPC UA NodeClass should be used for each main AAS elements. An AAS metamodel is composed by entities featuring attributes. In turn, entities can be classified in entities structuring the AAS (e.g., AssetAdministrationShell, Submodel, and Asset) and entities defining types for attribute values (e.g., Identifier, Key). Attributes, instead, can be classified as attributes containing values, attributes realizing composition and attributes containing AAS References. In particular, attributes realizing composition may contain also AAS References. In the following, reasoning about choice of the most suitable OPC UA NodeClass to map each of these AAS metamodel elements will be provided.

On the basis of the content of Section 4.2, the use of OPC UA objects for the representation of the main entities in the AAS metamodel that structure the AAS, seems reasonable and feasible.

To semantically distinguish OPC UA Objects mapping AAS entities between each other, declaration of ObjectTypes for each metamodel entity is needed. For instance, an ObjectType AASType may be defined to represent all Object Nodes mapping an AssetAdministrationShell entity; an ObjectType AssetType can be defined for all Object Nodes mapping an Asset entity. In general, for all entities of the metamodel constituting the AAS structure, an ObjectType should be properly defined. Mapping proposed in [5] seems based on the same assumption, as Objects are used to map main entities, providing naming convention for the relevant ObjectTypes.

AAS entities in the metamodel defining new types for attribute values (from now on referred as type in the context of AAS metamodel), often realize structures and enumerations; for this reason they fit to be mapped as OPC UA DataType since they can be both Structured and Enumeration, as said in Section 4.1. For instance, in the context of identifiable entities, two main type entities are introduced in the AAS metamodel: Identifier and IdentifierType. The former is a structured type and the latter is an

enumerative type. Identifier is composed by a string field (id), and a second field (idType) of type IdentifierType. Values of Identifier are used inside the attribute *identification* of Identifiable entities, as shown in Table 1. OPC UA Structured DataType and Enumeration DataType can be used to map Identifier and IdentifierType, respectively. A possible mapping solution is depicted in Figure 2.

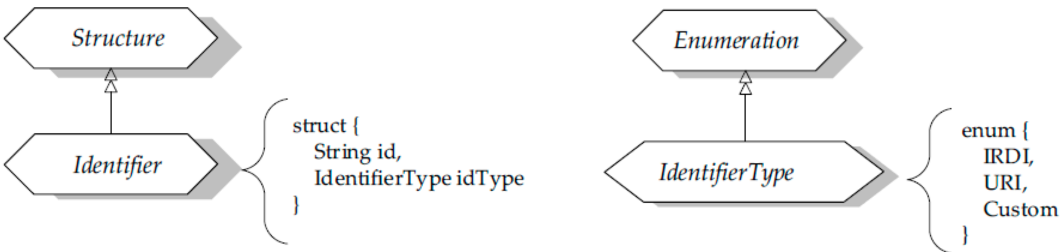

Figure 2. Identifier and IdentifierType mapped as DataTypes.

In [5] different mapping solutions are adopted for type entities depending on the case; no unique rule seems to be adopted. For instance, unlike authors' proposal, identifier is mapped with an ObjectType named AASIdentifierType structured with two Properties: id and idType. The two solutions are quite equivalent; the only considerations to be made at this time is that they led to different implementation strategies for entities featuring attributes containing values, like the *identification* attribute inherited by identifiable (containing Identifier values as said before). More insights about this consideration will be given in the following.

Since it has been assumed to map an entity structuring the AAS as OPC UA Object/ObjectType, reasoning to discover the most suitable solution to map its attributes must be done at this time. Let us start considering attributes containing values (from now on referred as value attributes). In AAS metamodel, value attributes describe features of the AAS entities. For instance, the value attribute description of the referable entities (see Table 2) maintains a brief description of the entity itself. According to what pointed out in Section 4.1, authors believe that attributes containing values can be mapped as OPC UA Properties. This choice is compliant with the decision of mapping type entities with DataTypes; in fact, Properties representing such attributes may contain values encoded using the DataTypes modelling the relevant AAS type entities. For example, resuming the case of the attribute identification, it may be mapped as an OPC UA Property, where the DataTypes shown by Figure 2 are internally used. The solution proposed in [5] based on the mapping of Identifier with the ObjectType named AASIdentifierType, sets the constraint to the kind of NodeClass that shall be adopted to map the identification attribute. According to the solution adopted in [5], identification can be mapped only as a component (thus, as an Object) having the AASIdentifierType ObjectType as type definition. Authors strongly believe that semantics of OPC UA Properties better reflect the meaning of value attributes than semantics of an OPC UA component, which represents a part-of relationship. In fact, the identifier of an entity cannot be considered a part of the entity but an inherent information of the entity itself.

Attributes of AAS entities reflecting composition (from now on referred as composition attributes) do not contain values but contain a collection of other entities. For instance, conceptDictionary attribute of the entity AssetAdministrationShell (see Table 3) contains a list of ConceptDictionary entities related to the AAS. Let us focus on the attribute conceptDictionary to discuss mapping solutions that may be adopted for composition attributes. A first solution is depicted in Figure 3a.

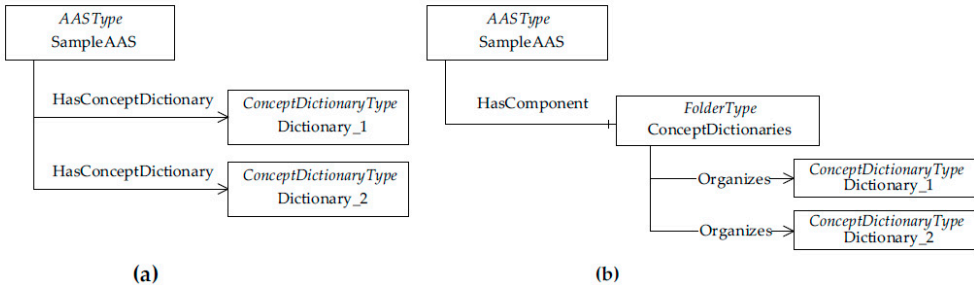

Figure 3. (**a**) basic and (**b**) optimized mapping solutions for attributes defining compositions.

Since ConceptDictionary is an entity structuring the AAS, all the ConceptDictionary entities contained in the conceptDictionary attribute are mapped as OPC UA Objects; furthermore, this mapping may be realized by the definition of a Hierarchical ReferenceType (which could be called "HasConceptDictionary" as shown by Figure 3a) and by the use of References of this type to connect the Object mapping the AssetAdministrationShell ("SampleAAS" Object in the Figure 3) to the Objects mapping the ConceptDictionary entities ("Dictionary_1" and "Dictionary_2" in Figure 3). In other words, the reasoning behind this mapping solution consists of using ad-hoc defined hierarchical references to represent the list of the entities contained in the composition attributes. The solution adopted in [5] uses HasComponent references to map this kind of attributes similarly to what is depicted in Figure 3a. On the other hand, the solution here proposed requires the definition of new Hierarchical ReferenceTypes (that can inherit from HasComponent) to semantically enrich the connection between an object and its components. The use of ad-hoc defined ReferenceType has the advantages to give more clarity about the structure of an Object and provides more filtering options for the Object browsing. Although the solution in Figure 3a can realize the mapping of composition attributes, it has the disadvantage that such attributes disappear in the mapping process (even though its informative content is spread over multiple hierarchical references). Considering the example shown in Figure 3a, no OPC UA elements represent at glance the conceptDictionary attribute. Authors believe that a second solution based on the use of Folder Objects provides a cleaner solution to map this kind of attribute. As said in Section 4.3, composition attributes may be mapped creating a Folder Object containing all the OPC UA objects mapping the entities of the composition; such a Folder collects the objects by means of OPC UA Organizes References. The solution is shown in Figure 3b. In this figure, it has been assumed to name the folder using the plural noun of the mapped attribute (i.e., "ConceptDictionaries" for the attribute conceptDictionary). This is just a suggestion for a naming convention to be adopted for mapping composition attributes.

The last category of attributes is the one concerning those ones containing AAS references to other entities. Since the argument is quite tricky, the discussion for these attributes is postponed in Section 5.3, which is reserved for this topic.

In general, every attribute described for entities in the AAS metamodel is annotated with a cardinality specifying whether the attribute is mandatory or optional for the entity. This behavior shall be maintained when an attribute is mapped either as a property or a component of an OPC UA ObjectType. As discussed in Section 3, properties and components are named InstanceDeclaration in the context of an ObjectType. OPC UA uses ModellingRules to declare an InstanceDeclaration either as mandatory or optional for all the instances of an ObjectType. It follows that, during the mapping process of attributes (of every category), proper ModellingRules must be selected for the InstanceDeclarations realizing the mapping. This reasoning seems adopted also in the approach presented in [5] since the document describes in detail which ModellingRule is applied in the attribute mapping according to its cardinality.

5.2. Structuring the OPC UA AddressSpace

Once defined which OPC UA NodeClasses should be used in the mapping of AAS entities, a decision about how to structure the AddressSpace shall be made. As pointed out in Section 4.3, when structuring an OPC UA AddressSpace, a good practice is using a Folder Object contained in the standard "Objects" folder as an entry point. Since the entity AssetAdministrationShell is the top-most entity in the hierarchy defined by the AAS metamodel, it makes sense defining a Folder Object named "Asset Administration Shells" as a component of the "Objects" Folder, as depicted in Figure 4. Such a Folder will be used to organize all those objects that are instances of AASType. In other words, it organizes all objects mapping AASs. This same solution is adopted in [5] to structure the AddressSpace in an OPC UA Server, where a Folder named "AASROOT" is used to aggregate all Object representing AASs. In the remainder of this section further consideration will be done about extending this folder-based organization to other entities and not for the entity AssetAdministrationShell only.

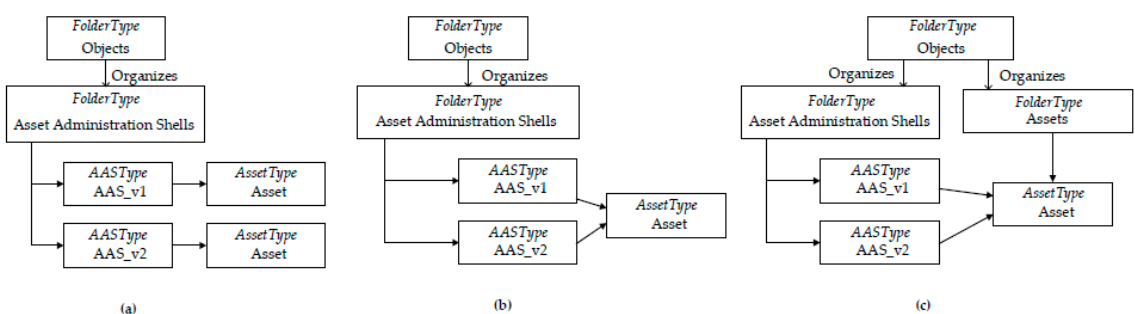

Figure 4. (**a**) basic and (**b**,**c**) optimized strategies to structure objects in the AddressSpace.

Since all the objects representing AAS are located under a single folder (i.e., "Asset Administration Shells"), an OPC UA Client can take advantage of this to select the desired AAS and to browse all the sub-entities it contains. For instance, since an AAS contains a reference to its asset, it seems logic to maintain this kind of connection in OPC UA. So that, an OPC UA Reference can be adopted to connect an AASType object to the AssetType object representing an asset. For the moment we do not consider which kind of ReferenceType may be adopted (e.g., Hierarchical, NonHierarchical) because we just want to focus on connections between nodes representing entities considering all the relationships to structure the information model. In the following, the aforementioned relationship between AAS and Asset will be used as example to show possible mapping strategies in the information model structuring process.

Figure 4a depicts a simple scenario where two AASType Objects represent two different versions of the same AAS (i.e., "AAS_v1" and "AAS_v2"). As the reader can notice, both objects are connected to the AssetType instance mapping the asset they are representing. It seems correct, but since the AASType Objects represents two versions of the same AAS, those two instances of AssetType represents, from a logical point of view, the same Asset. This means that a solution like this led to redundancy of the same entity on multiple Nodes. This situation can happen with other entities and not only for Asset. In general, all the Identifiable entities are the ones that can be shared across different AASs.

To avoid redundant nodes in the AddressSpace representing the same entity, the solution depicted in Figure 4b is here proposed to be used when a single node representing an asset is shared across two objects representing the AASs. This solution solves the problem of redundant data but has the drawback of having very important data nested inside the AddressSpace structure. As previously said, sharable entities are identifiable. Such entities are identifiable because they contain very important information that shall be easy locatable and globally identifiable. With a solution like the one in Figure 4b, in order to know which assets are contained in the AddressSpace of the OPC UA Server, a client should browse the folder "Asset Administration Shells" and repeat the browsing again. This kind of operation leads to a graph traversal which can be very complex in some cases. To cope with this issue,

the solution adopted in Figure 4c involves the use of a folder object named "Assets" to organize all the AssetType objects inside the AddressSpace, in the same manner the folder "Asset Administration Shells" organizes AASType objects. In general, creating a Folder Object as entry point for each kind of identifiable entity seems a good solution. This structures the AddressSpace like a sort of look-up table for identifiable entities, which is an important feature in the context of the AAS environment. The validity of this solution is confirmed by the choices made for the mappings of AAS into XML and JSON provided by [5]; in particular, in this document all identifiable entities are aggregated at root level to reduce redundancy. It is important to point out that these same considerations have not been adopted for OPC UA. Explanation about the choice to limit this mapping solution to XML and JSON only is not provided by [5]. The authors would like to point out that their proposal may be easily included in the current mapping into OPC UA presented in [5].

5.3. Mapping AAS References

The AAS metamodel provides a referencing mechanism deeply discussed in Section 2.2. The entity AAS Reference is a basic mechanism to connect some entities composing an AAS. For instance, an AssetAdministrationShell entity does not contain directly the Asset it is representing but its attribute asset contains an AAS Reference pointing to the asset itself, as said in Section 2.6. AAS Reference is made up by a list of keys (containing in turn entity identifiers) composing an unambiguous path to the pointed entity. The most important thing to consider in the mapping process of AAS References is guaranteeing that the order of keys constituting the path is respected.

A naïve solution could be to use OPC UA References to map AAS References since both create connections between entities and nodes, respectively. The problem with this solution is that AAS References contain inherent attributes (i.e., key) whilst OPC UA References, for definition, contain neither attributes nor properties/components. Furthermore, AAS References can point to an external source, and such behavior cannot be replicated using OPC UA References, which can point only to nodes contained in the AddressSpace.

Since the main aim of AAS References is to connect entities structuring the AAS, it seems reasonable that they may be mapped as OPC UA Objects as done for the entities structuring the AAS. According to this choice, the entity AAS Reference may be mapped with an ObjectType named AASReferenceType, whose structure is depicted by Figure 5. In the following, all the details of the mapping based on this solution will be discussed.

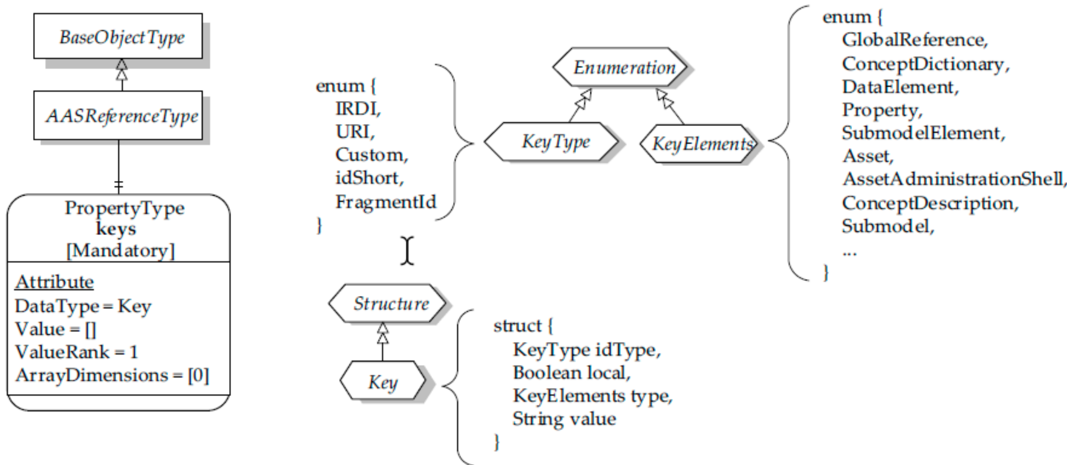

Figure 5. Structure of the AASReferenceType ObjectType and relevant DataTypes.

Attribute key of AAS Reference is mapped as an OPC UA Property according to the solution pointed out in Section 5.1. It has been assumed that this Property contains a value made up by one-dimensional array organizing all the elements modelling the keys (belonging to key type) composing the path to

an AAS entity. For this reason, the name “keys” has been given to the OPC UA Property because it better reflects the presence of multiple key values inside. Furthermore, it has been assumed to map the Key type as an OPC UA Structured DataType because this type entity is a structure, as discussed in Section 5.1. Similar considerations can be done for the type entities KeyType and KeyElements; since they are enumerations, these last can be seamlessly mapped as OPC UA Enumeration DataType. Figure 5 depicts the solution proposed for these types and the reader can notice that, according the proposed approach, it is possible to include both Enumeration DataTypes as fields of the Key Structured DataType. This solution has the great advantage to organize all the keys composing the path to an entity as an ordered array, so that the original order is respected in the mapping. The root of the path is identified by the element at index 0 and the element at last index identifies the entity pointed by the AAS Reference.

In the following, an example will be provided for a better understanding of the solution just presented, and to give some insights for further improvements. Let us consider the AAS entity named SampleAAS, shown by Figure 6a; it contains the attribute asset whose value is an AAS Reference pointing to the local Asset “3S7PLFDRS35” (the figure points out the relevant value of identification attribute). In Figure 6b an instance of the AASReferenceType ObjectType is applied to map the attribute asset. The AASReferenceType Object “asset” contains in its Property “keys” an array made up by just one key containing, in turn, the identifier of the pointed AssetType Object. Figure 6b also shows the AssetType “SampleAAS” Object modelling the SampleAAS Asset entity. The reader may notice the presence of the “Assets” Folder introduced in the previous subsection.

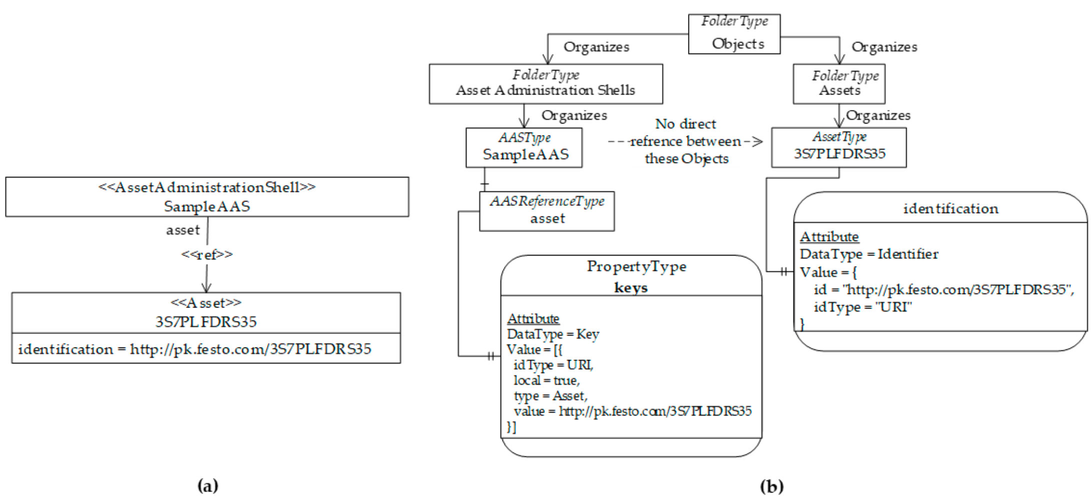

Figure 6. (**a**) AAS entity and (**b**) example of AAS Reference mapping applied to the asset attribute of AAS.

The solution until now presented, respects the structure of the AAS metamodel about referencing mechanism but it presents some limitations from the point of view of an OPC UA Client. A client browsing “SampleAAS” Object and its “asset” component in the AddressSpace doesn’t know at a glance which is the object it is referring to, but it knows just the identifier of such object (i.e., “http://pk.festo.com/3S7PLFDRS35”). Once taken the complete path to the referred object, a client should browse the AddressSpace (in the “Assets” Folder, specifically) looking for the Object associated to that path.

This limitation may be overcome by taking advantage of OPC UA References to connect the “SampleAAS” AASType Object with the “3S7PLFDRS35”Asset Type Object, as shown by Figure 6b. According to the general practices pointed out in Section 4.3, a NonHierarchical ReferenceType may be defined for this purpose for semantic reasons, since most of the time AAS Reference points to entities to define some relationship (like between AAS and the asset it is representing) and not a hierarchical

relationship (AAS does not own the asset). For this specific case, the "HasAsset" NonHierarchical ReferenceType can be defined and used instead of the dotted arrow shown by the Figure 6b.

It is worth noting that with this simple enhancement the number of browsing requests to know which object is pointed by an AASReferenceType instance is drastically reduced. In the case in examination, a client can browse SampleAAS for "HasAsset" Reference to look for the Object in the AddressSpace representing the Asset associated to the AAS.

Analyzing the approach presented in [5] about the mapping of AAS Reference, it is based on similar considerations done before but applied in different manners. In particular, in [5] AAS Reference is mapped as an ObjectType with a property named "keys[]". Furthermore, a NonHierarchical ReferenceType named "AASReference" has been defined and is used to connect objects mapping AAS References to the targeted object in the AddressSpace. There are two substantial differences between the solution proposed in [5] and the one here provided. The first one is that there is no specific mapping for keys, therefore the complete path of the AAS Reference is converted in a string formatted following a specific serialization rule mandated in [5]. The choice to map AAS Reference paths in a unique string as done in [5] requires that clients parse its content to retrieve all the information the path contains, whilst mapping key values by means of DataTypes allows information to be understood at an OPC UA level. The second difference is about the choice made in [5] to use OPC UA References of the same type (named "AASReference") to connect Objects mapping AAS References to the targeted Object in the AddressSpace. According to this solution, an OPC UA Client browsing the AddressSpace cannot distinguish the type of the Object pointed by the OPC UA Reference on the basis of the reference itself, but it will have a clear idea about the object once it is reached through the reference. The solution here provided requires the definition of different ReferenceTypes according to the object to be pointed; for example, in Figure 6b the "HasAsset" Reference is used as the object pointed represents an AAS AssetType. In the case of Objects modelling AAS Submodels, a "HasSubmodel" Reference may be used, instead. According to the solution here proposed, an OPC UA Client browsing the AddressSpace can immediately understand the type of object pointed by the reference, only by the analysis of the type of the reference itself.

6. Exploiting Novel OPC UA Features for the Mapping

In this Section authors want to discuss mapping solutions that can be adopted in case of specific AAS entities. Although the rationales provided in the previous section may also be taken into account, adoption of new OPC UA mechanisms defined in recent amendments may led to enhancements in the mapping process.

6.1. Mapping AAS Common Classes

In Section 5.1 general mapping solutions have been covered for main categories of elements composing the AAS metamodel. In particular, fine-grained mapping solutions about attributes has been provided. In the following, a step further will be done about mapping common classes in OPC UA. Common classes are a very important concept of the AAS metamodel since they feature attributes common to different AAS entities, as discussed in Section 2.

Let us consider the Submodel entity as a use case to introduce possible mapping solutions for common classes. According to the AAS metamodel, Submodel inherits from the common classes Identifiable, HasSemantics, HasKind and HasDataSpecification. For the sake of simplicity, only some of the relevant attributes will be taken into consideration in the following.

As the general rules described in Section 5.1 can be extended to all the metamodel entities inheriting from common classes, using these rules it is possible to achieve the mapping solution depicted in Figure 7a.

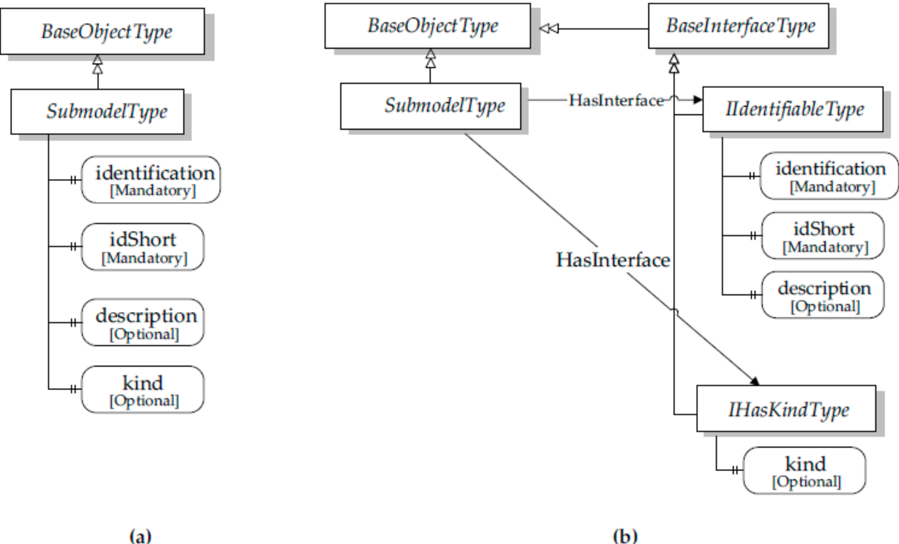

Figure 7. Mapping common Classes using classic OPC UA (**a**) and new OPC UA Interfaces mechanism (**b**).

In particular, the ObjectType named SubmodelType has been defined for all the Objects mapping Submodel entities. The attributes inherited from the common classes are mapped as OPC UA Properties and for each of them a ModellingRule is selected according the cardinality specified by each common class for every attribute.

This solution is quite simple and allows to maintain the same structure of the AAS metamodel. The drawback is that the OPC UA Properties must be redefined for every other ObjectType that map entities inheriting from same common classes. For instance, all other ObjectTypes mapping Identifiable entities shall define again the Properties "identification", "idShort" and "description" as InstanceDeclarations.

Figure 7b shows another solution that leverages on the new OPC UA Interfaces mechanism. Both Identifiable and HasKind common classes are mapped as interfaces called IIdentifiableType and IHasKindType, respectively. Their names are chosen following the guideline given in [34]. This solution shows how every InstanceDeclarations related to attributes coming from common classes are moved from SubmodelType ObjectType to the relevant Interface mapping the common class. SubmodelType now just make use of HasInterface References pointing the two Interfaces to declare that it inherits all their InstanceDeclarations. Compared to the solution shown by Figure 7a, attributes of common classes are mapped only once and different ObjectTypes mapping metamodel entities can point to the relevant interfaces by means of HasInterface References. This interface-based mapping solution is very advantageous but, since interface is a quite new feature of OPC UA, could be complicated finding an OPC UA Software Development Kit (SDK) supporting it for the implementation of an information model. This reason could lead to choosing the first mapping solution instead of the Interface-based one.

It is worth noting that all common classes can be mapped applying one of the two proposed solutions but for HasDataSpecification a different approach should be used, as will be discussed in the following subsection.

In [5] the OPC UA Interface-based mapping solution adopted for common classes is used just for Referable and Identifiable. For all other common classes different solutions are applied case by case; detailed explanation for the mapping of every common class is not provided in [5] but should be guessed from the various examples and mapping rules.

6.2. Mapping HasDataSpecification and DataSpecification

As explained in Section 2, instances of AAS entities inheriting from the HasDataSpecification common class feature more attributes than the ones defined by its original class. In particular, instances of an entity may have different additional attributes depending on values specified in

attributes inherited from HasDataSpecification. The additional attributes are defined in a DST that the HasDataSpecification entity must point to by means of an AAS Reference. Such a behavior is hard to realize using the information model mechanisms of OPC UA, since this requires that instances of a same ObjectType could feature different components and/or Properties depending on which DST they are referring to. Furthermore, a HasDataSpecification entity can point to more than one DST.

Beside Interface, in [34] a new interesting feature named AddIn is specified; it seems useful for the mapping of HasDataSpecification. An AddIn is an Object that associate features (represented by its ObjectType) to the Node it is applied to. OPC UA AddIn model differs from Interface model in that it is based on composition and not on inheritance. An AddIn is applied to a Node by adding a Reference pointing to the AddIn Instance; a HasAddIn Reference or a subtype shall be used [34]. There are no restrictions for AddIn ObjectTypes and there is no special super type for AddIns.

According to this AddIn model, an ObjectType named DataSpecificationType may be created as an AddIn ObjectType to map the entity DataSpecification, which is an abstract entity all the DST entities must inherit from. In order to map a concrete DST entity, an ObjectType inheriting from DataSpecificationType must be created.

Recalling that DST entities are Identifiable, all the ObjectTypes created must expose all the related attributes mapped according one of the two solutions discussed in the previous subsection and depicted by Figure 7; in the following, it will be assumed to use the solution depicted in Figure 7a.

The proposed mapping solution will be described using a showcase. In particular, the concrete DST entity DataSpecificationIEC61360 will be considered. As said in Section 2, DSTs can be used to define which attributes (besides those predefined by the metamodel) are used to define a submodel element or a concept description. The DST following the IEC 61360 property definitions (DataSpecificationIEC61360) is explicitly predefined and recommended to be used by the Plattform Industrie 4.0 [5].

As depicted in Figure 8, an ObjectType named DataSpecificationIEC61360Type is created to map the DataSpecificationIEC61360 DST entity as an OPC UA AddIn ObjectType. Such an ObjectType defines Properties and components as InstanceDeclarations mapping all the additional attributes defined by the DST. Due to lack of space, only few attributes are considered in Figure 8 (i.e., preferredName, shortName, unitId and valueFormat). The reader can notice that all the attributes of DataSpecificationIEC61360 inherited by Identifiable are mapped as Properties and components, as shown in the dashed squared area of Figure 8. These last ones contain all the information useful for the identification of the DST in the OPC UA AddressSpace.

In the following the mapping of instance of an AAS entity inheriting from HasDataSpecification common class will be shown, using another example. In particular, it will be considered an instance of a ConceptDescription describing an AAS property using the DataSpecificationIEC61360.

Assuming that the ConceptDescriptionType ObjectType maps the ConceptDescription entity, the "SampleDescription" Object shown in Figure 8 represents an instance of this entity.

An AddIn Object of the DataSpecificationIEC61360Type ObjectType is created (i.e., "IEC61360 Content"), as shown in Figure 8. This Object will be connected to SampleDescription by means of a HasDataSpecification reference.

Authors prefer to define the HasDataSpecification ReferenceType as subtype of HasAddIn so that a generic OPC UA client may easily detect the References involved in the mapping of HasDataSpecification entity. For this reason, Figure 8 shows such a Reference.

Adoption of AddIn to implement the concept of DST in OPC UA has been assumed also in [5]. In this document, AddIn is used to map HasDataSpecification entities in a similar manner. DST are mapped as ObjectType featuring properties and components mapping the additional attributes. The AddIn in this solution uses a pair of elements: one property that is a reference to the DST, and a component that is an instance of the ObjectType mapping the DST. In comparison with the solution here provided by authors, the main difference is the presence of the DST identifier in the adopted AddIn Object, whilst in the authors' solution the identifier of the DST is exposed by the AddIn ObjectType.

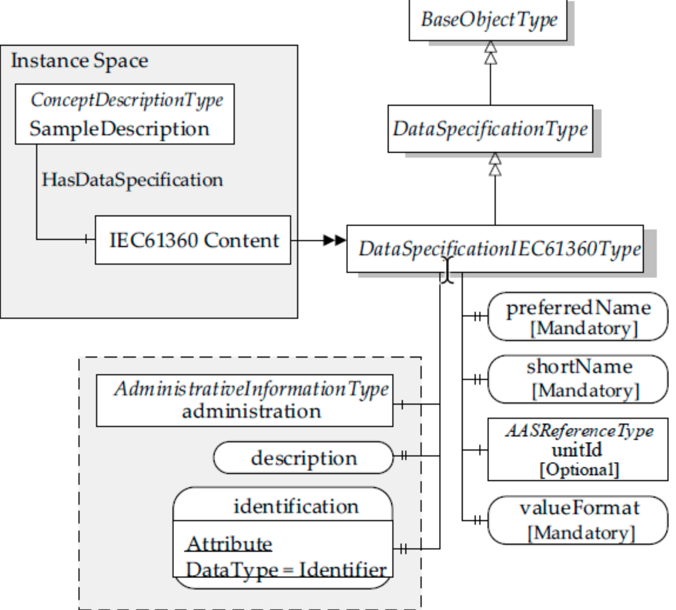

Figure 8. Mapping data specification templates (DST) entities as AddIn ObjectTypes and using them to map HasDataSpecification entities.

6.3. Mapping ConceptDictionary and ConceptDescription

Considering all the rationales provided so far, both ConceptDictionary and ConceptDescription entities can be mapped in an OPC UA Information Model defining proper ObjectTypes. But it is worth noting that, very recently, OPC Foundation released an amendment [35] of the OPC UA Specification defining new ObjectTypes and ReferenceTypes to define classification and additional semantics of a device in terms of an external data dictionary. In other words, such new types can be used to attach semantics to nodes in the AddressSpace referring entries in an external dictionary like IEC CDD or eCl@ss [5,32]. In particular, two main ObjectTypes defined in the amendment [35] are DictionaryFolderType that represents a dictionary, and DictionaryEntryType that represents a pointer to an entry in an external dictionary. Mapping ConceptDictionary and ConceptDescription by using these new ObjectTypes seems feasible and coherent with the strategies proposed in Section 5.1, even though some observations must be done. Both AAS entities inherit from some common classes and they feature attributes that seem not directly representable by properties defined in ObjectTypes proposed in [35]. This leads to the definition of suitable subtypes of these ObjectTypes able to map such AAS entities. Since ConceptDescription represents the entry of an external dictionary, it exposes the same attributes that the dictionary entry provides. These attributes can change case by case depending on the DST the ConceptDescription is adopting to describe semantics (because ConceptDescription is a HasDataSpecification entity, see Section 2.10); this requires the definition of suitable mapping solutions, like those discussed in Section 5.1 involving the creation of new ObjectTypes for AAS entities.

The solution adopted in [5] uses the DictionaryEntryType ObjectType to map the ConceptDescription entity. In particular, it defines specific subtypes of DictionaryEntryType that have at least one AddIn Object to allow the usage of the IEC 61360 DST, as described in the previous subsection.

It is worth mentioning that in the amendment [35] a new ReferenceType named "HasDictionaryEntry" has been defined and used to connect Objects to the relevant DictionaryEntryType; it follows that OPC UA References of this type can be adopted to map HasSemantics entities in case DictionaryEntryType ObjectType is used to map ConceptDescription entity. This solution has been also provided in [5] to map HasSemantics entities.

7. Case Study

In this section a case study is presented to the reader, in order to clarify some of the concepts and the mapping rules discussed so far. The case study takes into account an AAS modelling a motor controller; it is a simplified version of the example provided in [4]. Figure 9 shows the UML of this AAS. In the following, after a brief description of this AAS and the relevant mapping into OPC UA information model, the case study will be clearly defined and discussed.

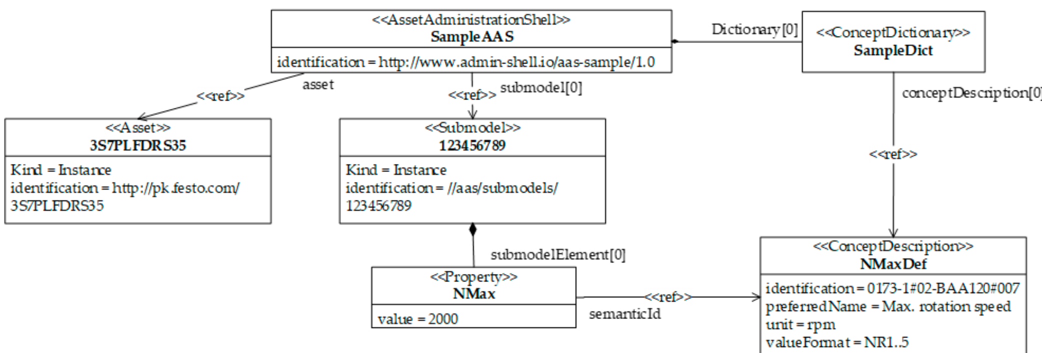

Figure 9. UML class diagram showing the AAS of the case study.

As shown by Figure 9, the AAS (named SampleAAS) contains a Submodel (named 123456789) and an asset (3S7PLFDRS35). The Submodel features only a property (NMax). Furthermore, the AAS has a ConceptDictionary (SampleDict) containing a ConceptDescription (NMaxDef).

All the Identifiable entities feature the attribute identification containing a globally unique identifier. Since NMax is a HasSemantics entity, its attribute semanticId contains the identifier of the ConceptDescription defining its semantics, i.e., NMaxDef. In particular, the semantics specifies that the Property value (2000) represents the maximum rotation speed supported by the motor controller, and it is expressed in rpm (revolutions per minute). Even if Submodel is also a HasSemantics entity, the figure does not show the relevant attributes due to lack of space. In this example, the Property NMax is used to show how semantics is mapped in the OPC UA Information Model.

All the attributes containing an AAS Reference are depicted in Figure 9 with a <<ref>> association. All the names of attributes featuring a composition are depicted using an array notation (e.g., submodel [0], conceptDescription [0]) when pointing to a specific instance of an entity.

In the following, the mapping into OPC UA AddressSpace of the AAS shown in Figure 9 will be given. The AddressSpace may be organized creating a Folder for each kind of identifiable entity, as discussed in Section 5.2. Therefore, the Folders “Asset Administration Shells” and “Assets” (shown in Figure 10) will organize objects mapping the AAS and the asset, respectively; in a similar manner, the folders “Submodels” and “ConceptDescriptions” will organize objects mapping the Submodel and the ConceptDescription, respectively. It is important to point out that these last two Folders are depicted in Figure 10, but for space reason, their contents are shown in Figure 11.

All the identifiable entities in the use case are mapped using instances of OPC UA ObjectTypes as discussed in Section 5.1: AASType for SampleAAS (see Figure 10), AssetType for 3S7PLFDRS35 (see Figure 10), SubmodelType for 123,456,789 (see Figure 11) and ConceptDescriptionType for NMaxDef (see Figure 11). Since all these ObjectTypes represent Identifiable entities of the metamodel, they point to an OPC UA Interface “IIdentifiableType”, as discussed in Section 6.1; this is not depicted in Figures 10 and 11 for space reason. All these instances feature a property “identification” that contains the relevant identifier of the entity represented.

All the attributes consisting in AAS References (depicted with <<ref>> in Figure 9) have been mapped using instances of the AASReferenceType ObjectType according to the solution proposed in Section 5.3. Furthermore, ad-hoc defined Non-Hierarchical ReferenceTypes are used to enhance the representation of AAS References in OPC UA and simplify the browsing of an OPC UA Client,

again said in Section 5.3. The Reference HasAsset in Figure 10, and the References HasSubmodel, HasSemantics, HasConceptDescription in Figure 11 are examples of this concept.

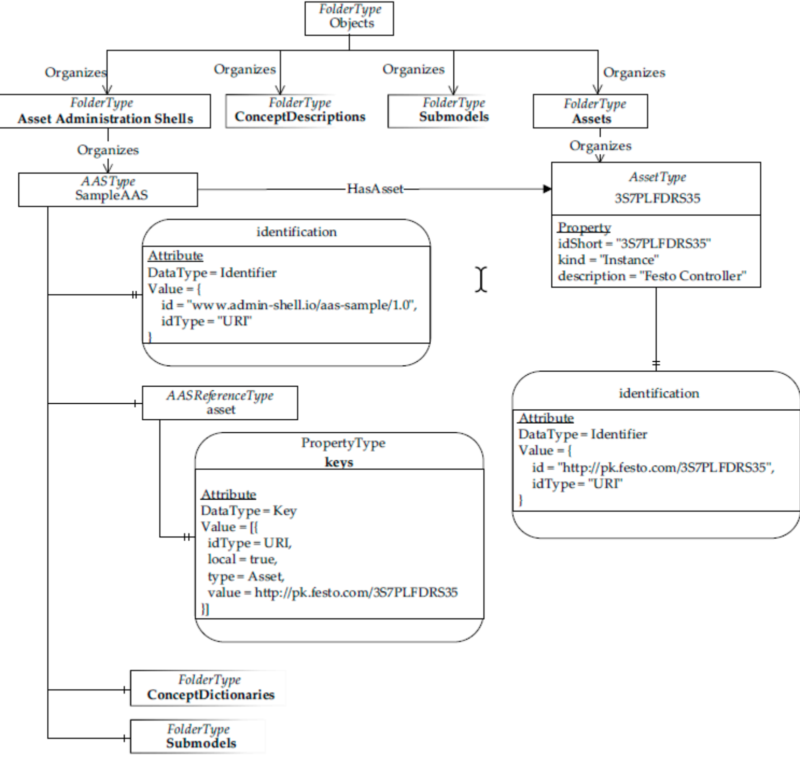

Figure 10. Mapping of the AAS considered in the case study into OPC UA information model.

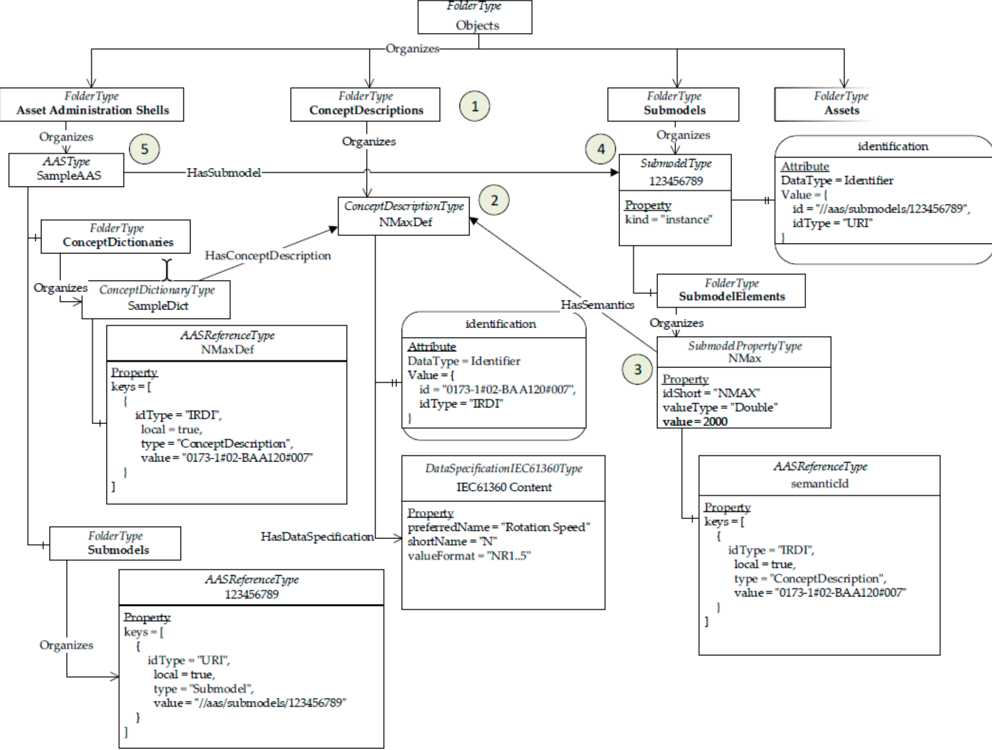

Figure 11. More details of the mapping of the AAS considered in the case study.

All the attributes consisting of composition are mapped as folder objects named using the plural noun of the relevant attribute. Such folders organize objects mapping the entities contained by such composition attributes, as discussed in Section 5.1. The folders “Submodels”, “ConceptDictionaries” and “SubmodelElements” are examples of what was just said. The first two are present in Figure 10, but their contents are highlighted in Figure 11.

The ConceptDictionary SampleDict and the AAS Property NMax shown in Figure 9 are mapped using instances of ad-hoc defined ObjectTypes as said in Section 5.1, i.e., ConceptDictionaryType and SubmodelPropertyType, respectively. As seen in Figure 11, these instances are the Objects “SampleDict” and “NMax”.

Finally, since the ConceptDescription NMaxDef features additional attributes coming from the DST for IEC 61360, an AddIn instance of the DataSpecificationIEC61360Type ObjectType (i.e., “IEC61360 Content” in Figure 11) is created and connected to the “NMaxDef” object by means of a HasDataSpecification Reference, as discussed in Section 6.2. Therefore, all the properties of this AddIn instances are filled accordingly to all the relevant values of the ConceptDescription.

Starting from this example of AAS (named SampleAAS, as said before) and its mapping into OPC UA, let us consider a realistic case study consisting of an assembly system. Several models of a certain product are assembled by human operators in the same flow line. It is assumed that the assembly system provides an operator support system (OSS) for the human operators in the assembly line; in particular, the OSS has the main task to provide information to perform the assembly cycle in the correct way as function of the model to assemble. For more details about Industry 4.0-based assembly systems and OSS, the reader may refer to [36].

In this case study, let us assume that the models of products to be produced are made up by several components to be assembled, among which there is the motor controller considered before in this case study. Different models to be assembled feature a motor controller, but each model requires a motor controller of a given maximum rotation speed. For example, assembly of Model X requires a motor controller with a maximum rotation speed value greater or equal to 2000, whilst the Model Y must be assembled including a motor controller with a maximum rotation speed greater or equal to 3000. For each product arrived to the human operator in the flow line, the OSS must suggest him the right motor controller component to be assembled according to the model of the product received; the OSS must specify an unambiguous id of the product part to be assembled in order to avoid assembling errors by the human operator.

Let us assume the scenario shown by Figure 12. The OSS includes an OPC UA Client which communicates with an OPC UA Server implementing the mapping of the AAS metamodel shown by Figure 9. Different instances of AASType “SampleAAS” are present; in the figure only two instances are depicted for space reason, Motor Controller 1 and Motor Controller 2. These instances differ for the NMAX property, as shown by Figure 12, i.e., the motor controllers represented by these instances differ for the maximum rotation speeds supported.

Let us assume that at a certain moment the OSS has to suggest to an human operator a specific motor controller to be assembled, and let us assume that the model of the product to be assembled requires that that the maximum rotation speed supported must be greater than 1800. The OSS will use the OPC UA Client to realize a search inside the AddressSpace of the OPC UA Server, exploring the available AASs, looking for an AASType “SampleAAS” instance featuring NMAX > 1800.

On the basis of the content of Figure 9, it is clear that the search in the AddressSpace is based on the knowledge that Maximum Rotation Speed has an identification = “0173-1#02-BAA120#007”.

Starting from folder “Concept Descriptions” (see point 1 in Figure 11), the object of ConceptDescriptionType type featuring a property “identification” containing the identification “0173-1#02-BAA120#007” is selected. In this case the object “NMaxDef” is chosen (point 2 in Figure 11). Starting from this Object, the OPC UA HasSemantics Reference is followed in the opposite sense in order to look for Objects of SubmodelPropertyType type. Considering Figure 11, the object “NMAX” is reached (Point 3 in the figure). On the property “value” it is possible to perform the query given in

input. In this case, the condition subject of the query is satisfied. The last step is to give back the id of the AAS, modelling the real motor controller featuring a rotation speed greater than 1800. Starting from object "NMAX" it is possible to reach its container, i.e., the SubmodelType Object "123456789" (point 4 in Figure 11). Finally, following the OPC UA Reference "HasSubmodel" in the opposite sense it is possible to reach the id of the AAS, i.e., Motor Controller 1 (Point 5 of Figure 11). This information will be passed to the human operator, in order to realize the correct assembly. 8. Software Implementation

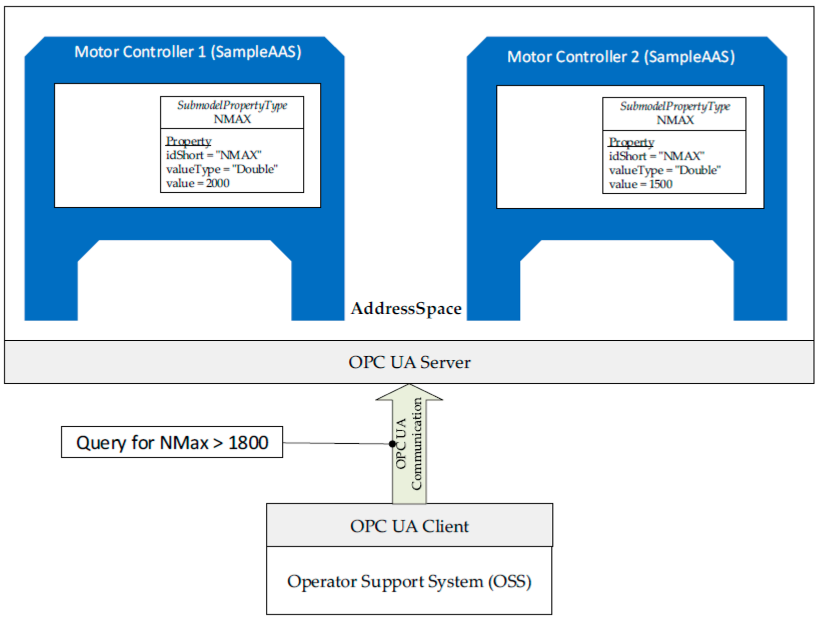

Figure 12. Example of OPC UA Server maintaining instances of AASType "SampleAAS".

On the basis of all the rationales and mapping solutions provided in this work, authors realized an OPC UA information model, called the "AAS Information Model", as a proof of concept; it is freely available at [37]. An extension for the OPC UA SDK in Node.js [38] has been developed providing new functions for developing OPC UA Server implementing the aforementioned AAS Information Model and validating the rationales provided in this work [39]. Descriptions of these implementations are provided on GitHub [37,39] together with installation guides and demos. The authors would like to point out that the implementations available on GitHub at [37,39] are open source and are distributed by the authors under the Apache license version 2.0 [40].

In order to realize an automatic process able to map a particular AAS into an OPC UA AddressSpace, a console application tool has been implemented. It is able to import an XML-serialized AAS and to generate an OPC UA Server exposing mapped from the AAS metamodel. Such a tool applies some of the main strategies discussed in this work to map AAS into an OPC UA AddressSpace; furthermore, it has been developed taking advantage of the authors' OPC UA SDK extension [39]. In the remainder of this section, the same AAS proposed as a use case in Section 7 has been used as evaluation of the console application, thus of the mapping rules. A file representing the AAS of the previous use case is used as a starting point; it contains the same information depicted in Figure 9, but in XML format. Information about the XML serialization is available in [5]. Due to lack of space only a short part of the XML file is depicted in Figure 13, showing the serialization of the 3S7PLFDRS35 Asset information only.

The console application has been developed as a command line application (CLI) running on Node.js. A command specifies which XML file must be imported. The first step realized by the application consists of the validation of the XML serialization file against the XML schema provided in the specification of AAS [5]. If the validation succeeds, the application starts a parsing procedure of the XML file. Considering the XML shown in Figure 13, the application retrieves all the relevant information describing the asset. Such information is used to fill the arguments of a function of the

SDK [39] that creates an instance of the AssetType ObjectType inside the AddressSpace, as depicted in Figure 10 for the 3S7PLFDRS35 Asset.

```
<aas:assets>
  <aas:asset>
    <aas:idShort>3s7plfdrs35</aas:idShort>
    <aas:description>
        <aas:langString lang="EN">Festo Controller</aas:langString>
    </aas:description>
    <aas:identification idType="URI">
          http://pk.festo.com/3s7plfdrs35
    </aas:identification>
    <aas:kind>Instance</aas:kind>
  </aas:asset>
</aas:assets>
```

Figure 13. XML serialization of the 3S7PLFDRS35 Asset.

This procedure is repeated recursively for every other entity contained in the XML file filling the AddressSpace step-by-step. It is worth noting that most of the mapping rules described in this paper are implemented in the code of the CLI application, whereas strategies for information representation are implemented in the information model.

As a result, the information of the XML file is mapped in the AddressSpace of an OPC UA Server and is made accessible to different clients in the network by means of OPC UA communication. The complete AddressSpace produced in this case study is depicted in Figure 14; the graphical visualization of its structure shown by the figure has been achieved using the OPC UA client by unified automation available at [41], connected to the OPC UA Server developed by the authors and maintaining the AddressSpace produced by the mapping process just described.

The reader can notice that the same Folder structure and all the information represented in Figures 10 and 11 are present in the AddressSpace because they are automatically and seamlessly generated by the SDK using the information model developed by authors.

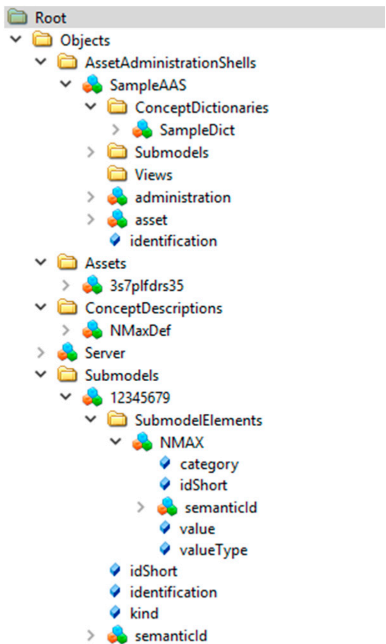

Figure 14. OPC UA AddressSpace generated on the fly by the command line application (CLI) parsing an XML serialized AAS.

8. Conclusions

The main contribution of the paper was that to provide insights and reasoning behind modelling techniques that should be adopted during the definition of an OPC UA information model exposing information coming from Asset Administration Shell. Introduction pointed out that this solution allows to improve interoperability of AAS, as relevant information can be exchanged between industrial applications using OPC UA which is currently considered a reference standard for the communications inside Industry 4.0.

Current literature presents only two other activities on the same subject. The first one is a mapping proposal of the latest release of the AAS metamodel into OPC UA [5]. The other one is an ongoing activity performed by a joint working group made up by ZVEI, VDMA and OPC Foundation aimed to the definition of a draft of a new OPC UA Companion Specification for AAS; to the best authors' knowledge no outcomes were still produced by this group. For this reason, all the general rationales and solutions provided in this paper about the mapping of AAS into OPC UA have been compared only with the proposal defined in [5]. Differences have been pointed out, when occurring, giving to the reader pros and cons behind each solution. Since AAS metamodel and its mapping in OPC UA is continuously developed and improved, authors believe that the work here presented contains reasoning that can be considered for future versions.

Mapping solutions presented in this work have been implemented by the authors; some of them are freely available on GitHub. Among these implementations, there is a tool able to realize the mapping of a particular AAS into an OPC UA AddressSpace. The main advantages of this tool is that this process is totally automated without requiring human intervention. Evaluation of this tool has been carried out by the authors, allowing to point out that no computational requirements are involved in the proposed mapping.

Author Contributions: Conceptualization, S.C. and M.G.S.; methodology, S.C.; software, M.G.S.; validation, S.C.; formal analysis, S.C.; writing—original draft preparation, S.C. and M.G.S.; funding acquisition, S.C. All authors have read and agreed to the published version of the manuscript.

Funding: This research was partially funded by "Piano per la Ricerca 2016/2018", DIEEI University of Catania.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Xu, L.D.; Xu, E.L.; Li, L. Industry 4.0: State of the art and future trends. *Int. J. Prod. Res.* **2018**, *56*, 2941–2962. [CrossRef]
2. Liao, Y.; Deschamps, F.; Loures, E.R.; Ramos, L.F. Past, present and future of Industry 4.0—A systematic literature review and research agenda proposal. *Int. J. Prod. Res.* **2017**, *55*, 3609–3629. [CrossRef]
3. German Institute for Standardization. *Reference Architecture Model Industrie 4.0 (RAMI4.0)*; DIN SPEC 91345; Deutsches Institut für Normung e.V.: Berlin, Germany, 2016.
4. Plattform Industrie 4.0, ZVEI. *Details of the Asset Administration Shell—Part 1: The Exchange of Information between Partners in the Value Chain of Industrie 4.0 (Version 1.0)*; Federal Ministry for Economic Affairs and Energy: Berlin, Germany, November 2018.
5. Plattform Industrie 4.0, ZVEI. *Details of the Asset Administration Shell—Part 1: The Exchange of Information between Partners in the Value Chain of Industrie 4.0 (Version 2.0)*; Federal Ministry for Economic Affairs and Energy: Berlin, Germany, November 2019.
6. VDI/VDE. Industrie 4.0 Service Architecture—Basic Concepts for Interoperability. 2016. Available online: https://www.vdi.de/ueber-uns/presse/publikationen/details/industrie-40-service-architecture-basic-concepts-for-interoperability (accessed on 23 March 2020).
7. Mahnke, W.; Leitner, S.-H.; Damm, M. *OPC Unified Architecture*; Springer: Berlin/Heidelberg, Germany, 2009.
8. Gutierrez-Guerrero, J.M.; Holgado-Terriza, J.A. Automatic Configuration of OPC UA for Industrial Internet of Things Environments. *Electronics* **2019**, *8*, 600. [CrossRef]
9. Ferrari, P.; Flammini, A.; Rinaldi, S.; Sisinni, E.; Maffei, D.; Malara, M. Impact of Quality of Service on Cloud Based Industrial IoT Applications with OPC UA. *Electronics* **2018**, *7*, 109. [CrossRef]

10. González, I.; Calderón, A.J.; Figueiredo, J.; Sousa, J.M.C. A Literature Survey on Open Platform Communications (OPC) Applied to Advanced Industrial Environments. *Electronics* **2019**, *8*, 510. [CrossRef]
11. Diedrich, C.; Belyaev, A.; Schröder, T.; Vialkowitsch, J.; Willmann, A.; Usländer, T.; Koziolek, H.; Wende, J.; Pethig, F.; Niggemann, O. Semantic interoperability for asset communication within smart factories. In Proceedings of the 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), Limassol, Cyprus, 12–15 September 2017.
12. Rohjans, S.; Uslar, M.; Appelrath, H. OPC UA and CIM: Semantics for the smart grid. In Proceedings of the IEEE PES Transmission and Distribution Conference and Exposition, Sao Paulo, Brazil, 8–10 November 2010; pp. 1–8.
13. Rohjans, S.; Piech, K.; Lehnhoff, S. UML-based modeling of OPC UA address spaces for power systems. In Proceedings of the IEEE International Workshop on Intelligent Energy Systems (IWIES 2013), Vienna, Austria, 14 November 2013; pp. 209–214.
14. Lehnhoff, S.; Mahnke, W.; Rohjans, S.; Uslar, S. IEC 61850 based OPC UA communication—The future of smart grid automation. In Proceedings of the Power Systems Computation Conference, Stockholm, Sweden, 22–26 August 2011.
15. Cavalieri, S.; Regalbuto, A. Integration of IEC 61850 SCL and OPC UA to improve interoperability in Smart Grid environment. *Comput. Stand. Interfaces* **2016**, *47*, 77–99. [CrossRef]
16. Shin, I.J.; Song, B.K.; Eom, D.S. Auto-Mapping and Configuration Method of IEC 61850 Information Model Based on OPC UA. *Energies* **2016**, *9*, 901. [CrossRef]
17. Lee, B.; Kim, D.K.; Yang, H.; Oh, S. Model transformation between OPC UA and UML. *Comput. Stand. Interfaces* **2017**, *50*, 236–250. [CrossRef]
18. Pauker, F.; Wolny, S.; Fallah, S.M.; Wimmer, M. UML2OPC-UA—Transforming UML class diagrams to OPC UA information models. In Proceedings of the 11th CIRP Conference on Intelligent Computation in Manufacturing Engineering (CIRP ICME'17), Gulf of Naples, Italy, 19–21 July 2017.
19. Henßen, R.; Schleipen, M. Interoperability between OPC UA and AutomationML. *Procedia Cirp* **2014**, *25*, 297–304. [CrossRef]
20. Reiswich, E.; Fay, A. Strategy for the amendment of plant information models by means of OPC UA. In Proceedings of the 10th IEEE International Conference on Industrial Informatics, Beijing, China, 25–27 July 2012; pp. 495–501.
21. Miyazawa, I.; Murakami, M.; Matsukuma, T.; Fukushima, K.; Maruyama, Y.; Matsumoto, M.; Kawamoto, J.; Yamashita, E. OPC UA information model, data exchange, safety and security for IEC 61131–3. In Proceedings of the SICE Annual Conference, Tokyo, Japan, 13–18 September 2011; pp. 1556–1559.
22. OPC Foundation; PLCopen. *OPC UA Information Model for IEC 61131-3-Release 1.00*; OPC Foundation: Scottsdale, AZ, USA, 2010.
23. OPC Foundation; AutomationML. *OPC UA for AutomationML, OPC UA Companion Specifications*; OPC Foundation: Scottsdale, AZ, USA, 2016.
24. MTConnect-Institute; OPC-Foundation. *MTConnect-OPC UA Companion Specification*; OPC Foundation: Scottsdale, AZ, USA, 2012.
25. OPC Foundation. *OPC Unified Architecture for IEC 61850, OPC 10040, Release Candidate 1.0*; OPC Foundation: Scottsdale, AZ, USA, 2018.
26. I4AAS-Industrie 4.0 Asset Administration Shell. Available online: https://opcfoundation.org/markets-collaboration/i4aas/ (accessed on 23 March 2020).
27. Cavalieri, S.; Mule', S.; Salafia, M.G. OPC UA-based Asset Administration Shell. In Proceedings of the 45th Annual Conference of the IEEE Industrial Electronics Society (IECON 2019), Lisbon, Portugal, 14–17 October 2019.
28. ZVEI; VDI. *Structure of the Administration Shell—Continuation of the Development of the Reference Model for the Industrie 4.0 Component*; Federal Ministry for Economic Affairs and Energy (BMWi): Berlin, Germany, 2016.
29. ZVEI. *Examples of the Asset Administration Shell for Industrie 4.0 Components—Basic Part*; German Electrical and Electronic Manufacturers' Association: Berlin, Germany, 2017.
30. IEC 61360-1:2017. Standard data element types with associated classification scheme. In *Part 1: Definitions—Principles and Methods*; International Electrotechnical Commission: Geneva, Switzerland, 2017.

31. Wagner, C.; Grothoff, J.; Epple, U.; Drath, R.; Malakuti, S.; Grüner, S.; Hoffmeister, M.; Zimermann, P. The role of the Industry 4.0 asset administration shell and the digital twin during the life cycle of a plant. In Proceedings of the 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), Limassol, Cyprus, 12–15 September 2017.
32. eCl@ss. Available online: https://www.eclass.eu/en/standard.html (accessed on 23 March 2020).
33. OPC Foundation. *OPC UA Part 3—Address Space Model*; OPC Foundation: Scottsdale, AZ, USA, 2017.
34. OPC Foundation. *Specification Amendment 7: Interfaces and AddIns*; Release 1.04; OPC Foundation: Scottsdale, AZ, USA, 2019.
35. OPC Foundation. *Specification Amendment 5: Dictionary Reference*; OPC Foundation: Scottsdale, AZ, USA, 2019.
36. Cohen, Y.; Faccio, M.; Galizia, F.G.; Mora, C.; Pilati, F. Assembly system configuration through Industry 4.0 principles: The expected change in the actual paradigms. *IFAC PapersOnLine* **2017**, *50*, 14958. [CrossRef]
37. CoreAAS. Available online: https://github.com/OPCUAUniCT/coreAAS (accessed on 23 March 2020).
38. Node-Opcua. Available online: http://node-opcua.github.io/ (accessed on 23 March 2020).
39. Node-Opcua-Coreaas. Available online: https://github.com/OPCUAUniCT/node-opcua-coreaas (accessed on 23 March 2020).
40. Apache Licence, Version 2.0. Available online: https://www.apache.org/licenses/LICENSE-2.0 (accessed on 6 April 2020).
41. UaExpert—A Full-Featured OPC UA Client. Available online: https://www.unified-automation.com/products/development-tools/uaexpert.html (accessed on 6 April 2020).

© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).