OPC® FOUNDATION

Implementation of the OPC UA Information Model for Analyzers

Version 1.00

Mariusz Postol (CAS), Voytek J. Janisz (ABB), Claude M Lafond (ABB), Jim Luth (OPC Foundation)

Abstract

A primary objective of analyzers is to determine the process state/ behaviour by measuring selected physical values that are characteristic for it. Obtained result – process data - is used to control and optimize the process.

To integrate analyzers into the supervisory control systems the process data must be transported and unambiguously represent the process for parties that are to be interoperable. To meet the above requirement it is proposed to employ OPC Unified Architecture technology that is universally accepted, platform-neutral communication standard. To make the process representation unambiguous companion specification has been issued that defines the Information Model for Analyzer Devices. Together with the specification - to prove the concept and the model correctness – a reference implementation has been released.

The ADI model is generic, and to be used must be adapted and implemented. To deal with the model tailoring to particular application needs it must be extended by defining dedicated types and customised by overriding definitions provided by the original ADI model. Any Information Model is only an abstraction describing the information exchanged by the communicating parties. To be utilized must be incorporated into an UA Server providing the Services and establishing the Address Space according to the OPC UA specification.

In this whitepaper selected aspects of the ADI model adoption are discussed. The whitepaper is dedicated for process architects and software developers to help deployment of the ADI ready solutions in real production environment. Very general scope of the discussion makes the paper a case study that can be useful also for the deployment of any OPC UA product.

Introduction

Analyzer is a device comprised of one or more measurement channels which has its own configuration, status and control. There are variety of analysers groups such as light spectrometers, particle size monitoring systems, imaging particle size monitoring systems, acoustic spectrometers, mass spectrometers, chromatographs, imaging systems and nuclear magnetic resonance spectrometers. These groups can be extended and each group can also be further divided.

Main goal of the analyzer device is to provide process data that is generated from scaled data by applying a chemometric model.

Process data is typically represented as a scalar value or a set of scalar values and it is often used for process control. Examples of process data are: concentration, moisture and hardness.

Scaled data is generated from raw data and represents an actual measurement expressed in meaningful units. Scaled data is typically an array of numbers. Examples of scaled data are: absorbance, scatter intensity. To obtain scaled data a mathematical description - analyser model - of the process and associated information to convert raw data into scaled data is used. Raw Data is generated by an analyser representing an actual measurement. Raw data is typically represented as an array of numbers. Examples of raw data are: raw spectrum, chromatogram and particle size bin count.

The analyzer configuration is a set of values of all parameters that when set, put the analyser in a well defined state.

Analyzers contain measurement channels. A channel is subset of an analyser that represents a specific sensing port and associated data, which includes raw and scaled data (e.g. spectrum), configuration, status and control.

To enhance the analyzer behaviour or operation replaceable accessory are used. An accessory is a physical device that can be mounted directly on the analyser or analyser channel. Examples of accessories are: vial holder, filter wheel, auger, and heater. The accessories are attached using accessory slots.

Sampling point is a physical interface point on the process where the process is monitored. To provide mapping between a channel and a process sampling points the concept of stream is used.

Because there are a large variety of analyzers types, from various vendors with many different types of data, including complex arrays and structures a real challenge is integration of the analyzers and control and monitoring systems. Initiatives such as Process Analytical Technology are driving analyzer integration and the best way to accomplish this is via open standards. To address this problem two questions can be distinguished:

- How to get access to (transport) the process data,
- How to represent (model) the process data.

To answer the first question we need a universally accepted, platform-neutral communication standard that allows also addressing the second question, i.e. designing an appropriate information model. OPC Unified Architecture (UA) technology [1], [2], [3] meets all the requirements, because:

- It is a platform neutral standard allowing easy embedded implementation
- It is designed to support complex data types and object models.
- It is designed to achieve high speed data transfers using efficient binary protocols.
- It has broad industry support beyond just process automation and is being used in support of other industry standards such as S95, S88, EDDL, MIMOSA, OAGiS.

One of the main goals of the OPC Unified Architecture is to provide a consistent mechanism for the integration of process control and enterprise management systems using client/server middle-range archetype.

To make systems interoperable, the data transfer mechanism must be associated with a consistent information representation model. OPC UA uses an object as a fundamental notion to represent data and activity of an underlying system. The objects are placeholders of variables, events and methods and are interconnected by references. This concept is similar to well-known object oriented programming (OOP) that is a programming paradigm using "objects" – data structures consisting of fields, events and methods – and their interactions to design computer programs. The OPC UA Information Model [1], [6] provides features such as data abstraction, encapsulation, polymorphism, and inheritance.

The OPC UA object model allows servers to provide type definitions for objects and their components. Type definitions may be abstract, and may be inherited by new types to reflect polymorphism. They may also be common or they may be system-specific. Object types may be defined by standardization organizations, vendors or end-users. Each type must have a globally unique identifier that can be used to provide description of the information meaning from defining body or organization. Using the type definitions to describe the exposed by the server information allows:

- Development against type definition.
- Unambiguous assignment of the semantic to the expected by the client data.

Having defined types in advance, clients may provide dedicated functionality, for example: displaying the information in the context of specific graphics.

The OPC UA information modelling concept is based on layers, which step by step expand the basic model provided by the OPC UA Specification.

The Information Model is a very powerful concept, but it is abstract and hence, in a real environment, it must be implemented in terms of bit streams (to make information transferable) and addresses (to make information selectively available). To meet this requirement, OPC UA introduces a *Node* notion as an atomic addressable entity that consists of attributes (value-holders) and references (address-holders of coupled nodes). The set of *Nodes* that an *OPC UA Server* makes available to clients is referred to as its Address Space [1], [2], [4], which enables representation of both real process environment and real-time process behaviour.

Information Model for analysers

In 2008 the OPC Foundation announced support for Analyzer Devices Integration into the OPC Unified Architecture and created a working group composed of end-users and vendors with main goal to develop a common method for data exchange and an analyzer data model for process and laboratory analyzers. In 2009 the OPC Unified Architecture Companion Specification for Analyser Devices was released [7]. To prove the concept a reference implementation have bee developed containing ADI compliant server and simple client using the Software Development Kid released by the OPC Foundation.

The model described in the specification [7] is intended to provide a unified view of analysers irrespective of the underlying device. This Information Model is also referred to as the ADI Information Model. As it was mentioned, analysers can be further refined into various groups, but the specification defines an Information Model that can be applied to all the groups of analysers.

The ADI Information Model is located above the DI Information Model [8]. It means that the ADI model refers to definitions provided by the DI model, but the reverse is not true. To expand the ADI information model, the next layers shall be provided.

Analysing in detail the whole ADI Information Model is impractical here. Hence, the discussion below will be focused only on selected types defined in this specification to illustrate the design practice of the model adoption.

The object model that describes analysers is separated into a definition of the types representing main parts of the device, namely: *AnalyserDeviceType*, *AnalyserChannelType*, *StreamType*, *AccessoryType* and *AccessorySlotType*.

In general terms *AnalyserDeviceType* represents the instrument as a whole. Each object of the *AnalyserDeviceType* has at least one component of the *AnalyserChannelType* and may have components of the *AccessorySlotType*. Similarly, each object of *AnalyserChannelType* may have *AccessorySlotType* components.

AnalyserDeviceType is an abstract type which shall be subtyped for different types of analyser devices. In the specification [ADI} there are defined the following subtypes of the *AnalyserDeviceType*: *SpectrometerDeviceType*, *AcousticSpectrometerDeviceType*, *MassSpectrometerDeviceType*, *ParticleSizeMonitorDeviceType*, *ChromatographDeviceType*, *NMRDeviceType*. Each of these types may be further subtyped by device vendors to harmonize the Information Model and the underlying process.

ADI Information Model Implementation

Typical implementation architecture consists of OPC UA Clients, which are aware of the ADI Information Model connected to an OPC UA Server, which implements the ADI Information Model (Figure 1).

To illustrate this whitepaper a generic client contained in the OPC Foundation SDK has been used because only Address Space browse service was needed. In production environment the Information Model (types) knowledge may be used to offer additional functions, like customised control panels, dedicated data visualization panes or predefined structure of the database tables. Types knowledge also simplify configuration of the clients, because all of the items composing the complex process information can be accessed automatically.

The OPC Unified Architecture (UA) is a standard that allows servers to provide real-time process data, environment metadata and even non-process data to clients, in a unique and platform-independent way. To meet this objective, each server instantiates and maintains an Address Space that is a collection of information to be exposed to clients [1], [2], [4]. The OPC Unified Architecture Address Space consists of Nodes and References. The main role of the Nodes is to expose the underlying process state as a selected, well-defined piece of information.

To implement the Address Space concept, two questions must be addressed:

- How to create and maintain it?
- How to couple the nodes with the real-time process data sources?

To create the Address Space the UA server must instantiate all Nodes and interconnects them by means of References.

Using the Nodes instances by means of a well-defined set of services [5], clients get access to data representing the state of a selected part of the underlying process. Nodes are divided into classes. The *Variable* class is used to represent the values – has a *Value Attribute*. To be used as the real-time process state representation, the value of the *Value Attribute* must be bound with a real data source, e.g. an analog signal. The *Method* class represents functions that can be called by the clients connected to the server. In this case the real-time process bindings are responsible for conveying the *Parameters* current values, invoking the represented function and returning the execution result. Both classes are the main building blocks that allow the server to couple the exposed Address Space with the current state and behavior of the underlying process.

Therefore, to maintain this coupling, there must be established a connection to physical plant floor devices used to transfer real-time process data.

The methods of Nodes binding with real-time process data are vendor specific. Nodes management functionality on the client part is standardized by the OPC UA Service Model [5] as a set of services. Access to the values representing the current process state is provided by the Read/Write functions. The client can also be informed about changes of the process state using "data change" notifications. Invoke and event notification functionalities allow clients to use the *Methods*.

To implement the functionality presented above, we need to use three coupled function classes (Figure 1):

- *UA Services*
- *Nodes Management*
- *Process Link*

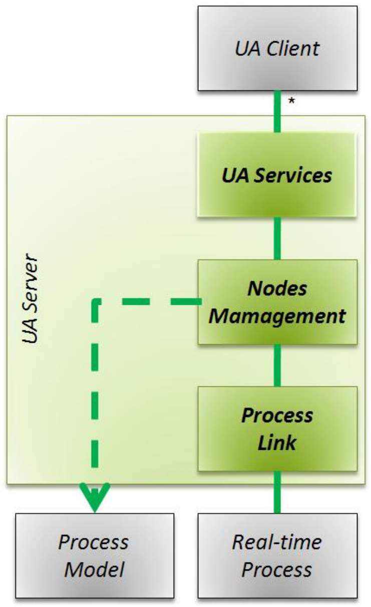

Figure 1 Typical UA archetype

The diagram in Figure 1 shows the dependencies and associations between the function classes mentioned above. In this architecture, the *Process Link* is responsible for transferring real-time process data up and down. The *Nodes Management* function class couples the real-time process data with appropriate Nodes instances representing process metadata and provides a homogenous picture created this way to *UA Services* that finally expose it to all connected clients.

In this case, real-time process data is obtained from the analyzer device as a part of the process. The *Process Link* function class gets access to it using the underlying communication infrastructure and vendor specific protocols. For embedded applications it may use directly internal controller registers of the device.

To create the Address Space - i.e. to instantiate all Nodes and interconnect them by means of References - the *Nodes Management* function class uses a predesigned static *Process Model* (Figure 1) providing a detailed description of all the Nodes, including their *Attributes* and *References*. Static means that the model is predefined for the selected environment, but it does not mean that the exposed Address Space is static. In this approach, Nodes can be instantiated and linked dynamically, however the process must conform to the model definition. Dynamic behaviour of the Address Space can be controlled by the connected clients using services or by the current state of the process.

From the discussion above we learn that before Nodes making up the Address Space can be instantiated by the server, this Address Space must be designed first. Model designing is a process aimed at defining a set of Nodes and their associations and, next, creating this *Process Model* representation in a format appropriate for the implementation of the *Nodes Management* function class.

Depending on the UA server implementation, the Information Model representation and support for the modelling process varies. CAS Unified Architecture Address Space Model Designer (ASMD) is a software tool that is intended to help architects, engineers and developers accomplish *Process Model* preparation. This tool supports all aspects of the model designing process including edition, visualization and, finally, generation of files allowing the server to expose provided real-time process metadata.

The ASMD implements conceptual containers called solutions and projects (Figure 2) to apply its settings. Any solution contains one or more projects and it manages the way the designer configures, builds, and deploys sets of related projects. Any project includes source files containing the model representation

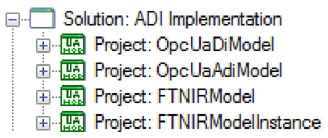
Solution: ADI Implementation
Project: OpcUaDiModel
Project: OpcUaAdiModel
Project: FTNIRModel
Project: FTNIRModelInstance

Figure 2 Solution concept

and related metadata such as namespace, properties and references to other projects. As projects are built output files are produced to be used by the UA Server to expose the designed Address Space.

The OPC UA information modelling concept is based on layers, which step by step expand the basic model provided by the Specification [4], [6]. To follow this concept, ASMD uses projects to implement model layers. Projects are related to each others making up a layered hierarchy and ADI Information Model (*OpcUaAdiModel* in Figure 2) is one of them.

The ADI model is located above the DI Information Model (*OpcUaDiModel* in Figure 2). It means that the ADI model refers to definitions provided by the DI model, but the reverse is not true. To meet the device vendor specific requirements and further expand the ADI model, the next layers shall be provided. In Figure 2 project *FTNIRModel* contains all the type definitions required in this particular example and the project *FTNIRModelInstance* has an object declaration representing the device.

The model representation contained by the projects is a collection of definitions of nodes and their references, which create a domain called namespace. This namespace is identified by global unique URI (Uniform Resource Identifier) that has two roles: avoid ambiguity and define organizations responsible for designing and maintenance of the coupled models.

The Address Space is a collection of Nodes that are instances of node classes. Each node class is defined as an invariable set of named attributes and a collection of references that shall be assigned (given) values when the Node is instantiated at runtime. The Address Space Model Designer allows designers to add freely nodes of any class defined by the specification to the model tree view (Figure 3), i.e. *View*, *Object*, *Variable*, *Method*, *ObjectType*, *DataType*, *ReferenceType*, and *VariableType* [1], [2], [4]. Initial values of the attributes can be provided using the property grid pane.

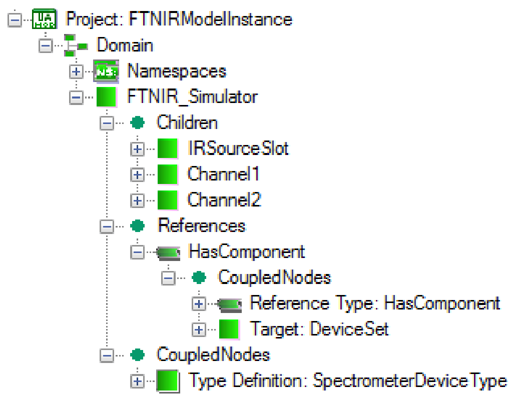
```
Project: FTNIRModelInstance
  Domain
    Namespaces
    FTNIR_Simulator
      Children
        IRSourceSlot
        Channel1
        Channel2
      References
        HasComponent
          CoupledNodes
            Reference Type: HasComponent
            Target: DeviceSet
      CoupledNodes
        Type Definition: SpectrometerDeviceType
```
Figure 3 Example of a user device object

Figure 3 is an example of a graphical representation where the snippet of model definition is presented as nodes on a tree view. To facilitate organization and definition of relationship some tree nodes have special role. The top level *Domain* node is a container of all the definitions belonging to the namespace represented by the project. *Namespaces* collects all namespaces definition that the projects refer to.

Each node added to the tree view has also a few dedicated branches being placeholders of special treatment. The main aim of the *Children* node is to create "part of" relationship. It is an entry to a branch that collects components, i.e. in the Address Space established by a server all nodes in this branch will have been referenced by the parent using a reference of type derived from *HasComponent* or *HasProperty*. For example, *Channel1* (Figure 3) is a component of the *FTNIR_Simulator* object and adding it to this container causes that in the established Address Space it will be referenced by the *FTNIR_Simulator* using *HasComponent* reference. *References* tree node creates a branch that contains all references of the parent node. Automatically created *CoupledNodes* tree node is a container of all nodes coupled with the parent, e.g. type definition of the parent node (*HasTypeDefinition*), target of a reference, etc. This node is used to improve readability of the model and enhance navigation.

Finally, having designed the model it must be compiled to provide expected by the UA Server *Process Model* (Figure 1). This operation is partially semi-automatic, but must be accompanied by definition of the bindings between instantiated Nodes in the Address Space and real-time data sources. In the case of the reference application the bindings are added manually by modification of the auto-generated program source code to add behaviour necessary to get access to the data. For generic solutions, Address Space Model Designer can be provided with an external component supporting the selected server configuration in the context of the model. Configuration in the context of the model means that the tool offers possibility to select real-time data source for each instantiated node independently, e.g. select a register of the controller or a tag from an OPC DA client subscription.

ADI Information Model Adoption

The main tasks of the ADI Information Model adoption are as follows:

1) Model extension by definition of vendor specific types.
2) Model customization by overriding components of the existing types.
3) Instantiation of all objects making up the ADI compliant Address Space.

The Information Model defined in the ADI specification [7] is generic, and to expose representative information for a selected analyser device it must be extended further by defining parameters and/or subtypes derived from the base types provided in this specification. These types can be used to create all objects (e.g. *FTNIR_Simulator* - see Figure 3) representing the analyser device in the Address Space exposed by the UA Server. Each analyser device must be represented in the Address Space by an object of a type indirectly derived from an abstract *AnalyserDeviceType*. Additionally, this object must be interconnected to the standard infrastructure of the Address Space. Many instance declarations in the ADI Information Model are optional or have only meta-definition (e.g. components representing channels); therefore they are not created by default as a result of instantiation their parent and must be subject of further definition refining.

Extending the ADI Information Model and refining the definitions provided in this specification should allow designers to adjust the Address Space exposed by the UA Server so as to represent truthfully the underlying process.

An object of a type derived from the *AnalyserDeviceType* representing the device as a whole is the topmost one in the ADI object model. *AnalyserDeviceType* is an abstract type, and therefore to create an object of this type it must be subtyped for different types of analyser devices. A tree view in Figure 3 illustrates an example of the *FTNIR_Simulator* object of the type *SpectrometerDeviceType* that is to represent an example spectrometer in the UA Address Space. The definition of this object causes that the UA Server instantiates it and all the mandatory instance declarations (components) while creating the exposed Address Space.

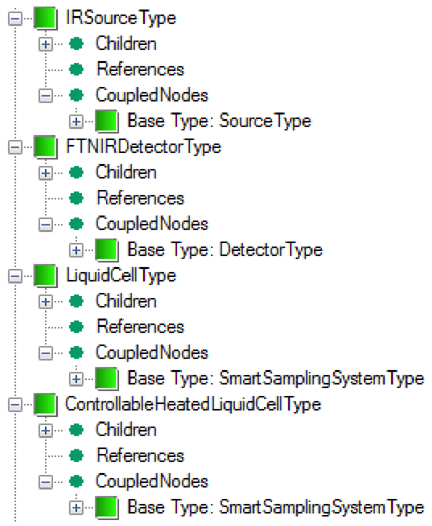

Figure 4 New types definition

To create a vendor specific Information Model, usually additional types must be defined. Figure 4 illustrates a set of new types derived indirectly form the *AccessoryType*. More examples on how to expand the model are described in the specification [7] and in the [2].

The Information Model representing a device is layered (Figure 2), therefore the question how to distribute definitions among layers must be addressed. According to the best practice rules, the vendor specific part of the Information Model shall be layered as follows:

1. Base product type definitions.
2. Product models type definitions.
3. Instance declaration.

In this simple example no product models are recognized, therefore we have no definition on layer 2. According to the above rule the *FTNIR_Simulator* object has been located in the *FTNIRModelInstance* project and all types presented in Figure 4 are provided by the *FTNIRModel* project (Figure 2).

ADI Device Representation Instantiation

Information exposed by the OPC UA Server is composite. Generally speaking, to select a particular target piece of information a client has two options: random access or browsing. Random access requires that any target entity must have been assigned globally unique address and the clients must know it in advance. We call them well-known addresses. It is applicable mostly to entities defined by standardization bodies. The browsing approach means that clients walk down available paths that build up the structure of information. This process is costly, because instead of pointing out the target, we need to discover the structure of information step by step using relative identifiers. The main advantage of this approach is that clients do not need any prior knowledge of the structure – clients of this type are called generic clients. To minimize the cost, after having found the target, every access to it can use random access. Random access is possible since the browsing path is convertible to a globally unique address using the server services [5].

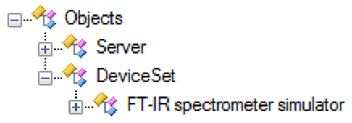

Figure 5 OPC UA Client topmost view

Taking into consideration that the browse mechanism is based on the incremental and relative passage along a Nodes path, we can easily find out that each path must have a defined entry point, so we must address a question where to start. To meet this requirement, the OPC UA Specifications provide a predefined structure [6] containing well defined Nodes that can be used as anchors to start discovering the Address Space by clients.

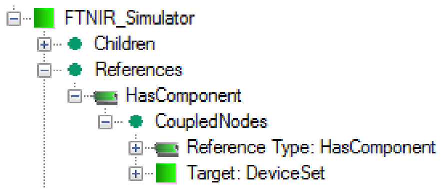

Figure 6 Object locations

For the above example, a typical organization of the UA Server Address Space seen by a client is presented in the Figure 5. Two objects can be distinguished in this hierarchy: *Objects* and *DeviceSet*. The purpose of the *Objects* is that all objects and variables that are not used for type definitions or other organizational purposes (e.g. organizing the *Views*) are accessible through hierarchical references starting from this node. *DeviceSet* is an object containing all the devices according to the DI Information Model specification [8].

To locate an object in the nodes layout presented in Figure 5, a *HasComponent* reference must be added to the object (Figure 6). The *HasComponent* references are used to browse the Address Space from the top toward the bottom, but it is worth noting that in the model this reference is added to the destination node instead (Figure 6). Main reason is to keep the DI model representation invariant.

It should be noted that in Figure 5 the *FTNIR_Simulator* object has a different name (*FT-IR spectrometer simulator*) than in the definition (Figure 6). It is because each node in the Address Space has a *DisplayName* attribute that contains the localized name of the node. Clients shall use this attribute if they want to display the node name to the user. They shall not use the browse name for this purpose.

The Address Space content exposed by the UA Server can change in time reflecting any change of the underlying process. A good example, where the dynamic content of the Address Space is very useful, is hot-swappable device modules, like accessories. In this case the UA Server must be able to discover the current configuration and instantiate/delete Nodes and/or References according to this configuration.

As it was stated above, to create the Address Space, the UA Server needs to instantiate Nodes and interconnect them by References. According to the requirements defined in [6], to create the Address Space, any UA Server must instantiate all mandatory objects that organize the Address Space and can be used as entry points to start browsing and discovering it. One of them is *Objects* (Figure 5) that is the browse entry point for objects.

Having all objects organizing the Address Space, the UA Server creates instance of objects declared by the custom information model. In the above sample model, the server instantiates *FTNIR_Simulator* as a component of the *DeviceSet* defined in the *OpcUaDiModel* (Figure 2).

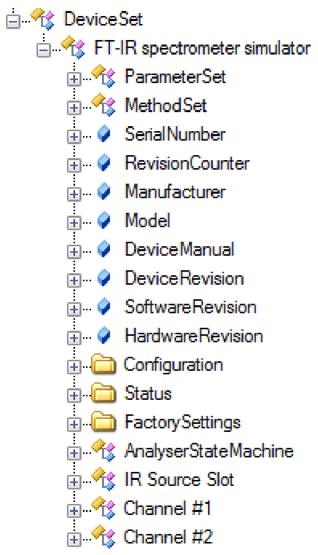

Figure 7 FTNIR Address Space

After parent type instantiation, the server creates also all components of that type and all its base type components called instance declaration. This operation is applied recursively. In other words, in order to get full information about a subtype, the inherited instance declarations shall be collected from all types that can be found by recursively following the inverse *HasSubtype* references from the subtype. For the above example, the Nodes under the *FT-IR spectrometer simulator* (Figure 7) are a collection of all components (coupled by HasComponent reference to the type) traversing the inheritance chain:

1) *SpectrometerDeviceType*
2) *AnalyserDeviceType*
3) *DeviceType*
4) *TopologyElementType*
5) *BaseObjectType*

The new created nodes have the same value of the *BrowseName* attribute as in the type definition. Since *BrowseName* values shall be unique in the context of the parent type definition, new nodes may be created without any fear of breaking the browse path uniqueness rules. This browse path is always unique, because the *BrowseName* of the created main object must be unique in the context it is located in and all instance declarations shall have unique *BrowseName* values in the context of types they are defined by. More detailed discussion on instance declaration concept can be found in [4], [2].

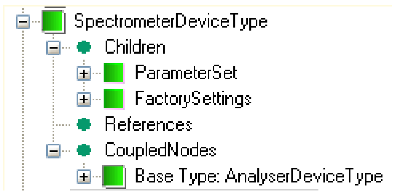

Figure 8 SpectrometerDeviceType

The inheritance mechanism and automatic creation of instance declaration cause that the objects in the Address Space exposed by the UA Server may have more components then they type definition. An example is the *FTNIR_Simulator* object (labelled *FT-IR spectrometer simulator* in Figure 8 after the *DisplayName* attribute) of type *SpectrometerDeviceType.* The type has only two components: *ParameterSet* and *FactorySettings* (Figure 8), but in the exposed Address Space fragment (Figure 7), the other Nodes (except *IRSourceSlot*, *Channel1* and *Channel2*) are created because they are defined as components in one of the base types making up the inheritance hierarchy (Figure 9).

The instantiation process may be modified by overriding the already defined components in the derived types and by adding new components manually.

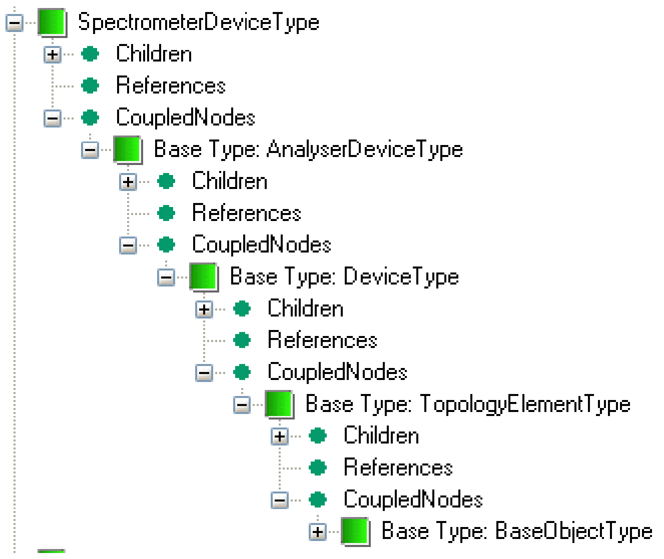

Figure 9 *SpectrometerDeviceType* inheritance hierarchy

IRSourceSlot, *Channel1* and *Channel2* are added to the definition of the *FTNIR_Simulator* object (Figure 3). *IRSourceSlot* is of *AccessorySlotType*, and *Channel1* and *Channel2* are of *AnalyserChannelType*. All are defined as components of the *AnalyserDeviceType*. The definition of the *AnalyserDeviceType* allows designers to add as many components of the *AccessorySlotType* and *AnalyserChannelType* to the created object as it is necessary to represent the structure of an existing analyser. *AnalyserDeviceType* defines cardinality 1..* for the channel meta-definition, therefore it imposes a limitation that at least one channel must be created. Components of the *AnalyserChannelType* are an example where the basic ADI Information Model must be customised to harmonise the Address Space with the represented underlying environment.

ParameterSet and *FactorySettings* (Figure 8) are examples of how to customise the Information Model by extending or modifying the already defined components. *ParameterSet* is defined as component of the *TopologyElementType* and is overridden by the *AnalyserDeviceType* and next by the *SpectrometerDeviceType*.

The properties *SerialNumber*, *RevisionCounter*, *Manufacturer*, *Model*, *DeviceManual*, *DeviceRevision*, *SoftwareRevision* and *HardwareRevision* (Figure 7) are inherited from the *DeviceType* defined by the [DI] Information Model. Values of these properties can be defined as default values in the model or provided by the equipment at runtime.

Configuration, *Status* and *FactorySettings* folders are inherited from the *FunctionalGroupType* (Figure 7).

Very important *AnalyserStateMachine* component of the *FT-IR spectrometer simulator* are inherited from *AnalyserDeviceType*. *AnalyserStateMachine* is an object that represents behaviour of the analyser device using the state machine concept. This construct can be used to model discrete object behaviour in terms of the states an object can reside in and transitions that can happen between those states. State machines are built as complex objects using dedicated *ObjectTypes*, *VariableTypes* and *ReferenceTypes*, whose behaviour is governed by the rules that must be strictly observed. A state is a condition in which an object can be at some point during its lifetime, for some finite amount of time. A transition is a change of an object from one state (the source state) to another (the target state). The transition is triggered ("fires") when an event of interest - cause - to a given object occurs. According to the Information Model concept, causes are represented in the form of Methods that shall be called, but a vendor can define other items or have them be internal (i.e. nothing is listed causing the transition). There may also be an action associated with a triggered transition. This action is executed unconditionally before the object enters the target state and effects in the form of *Events* that are generated.

The *AnalyserDeviceType* is also a source of definition of the *Methods* exposed by the UA Server (Figure 7). All the methods are collected as components of the *MethodSet* object.

Summary

There are variety of analysers groups. however, for most of them a primary objective is to determine the process state/behaviour by measuring selected physical values that are characteristic for it. Obtained result – process data - is used to control and optimize the process.

Initiatives such as Process Analytical Technology are driving analyzer integration and the best way to accomplish this is via open standards. To address this problem two questions have been distinguished in this paper:

- How to get access to (transport) the process data,
- How to represent (model) the process data.

To answer these questions the OPC Unified Architecture technology is proposed as the platform to transfer data and define process Information Model.

In the article typical architecture of the UA Server is presented. There are many possibilities to implement ADI compliant server, but there are two common questions that are disused:

- How to design the sever Address Space?
- How to bind the Nodes with the real process data sources?

To increase productivity and avoid mistakes Address Space Model Designer is used to support the Information Model adaptation process. In the presented architecture two approaches to get access to process data are distinguished:

- Hard coded – additional behavior responsible for data access is programmed.
- General bindings – process data binding are configured using Address Space Model Designer.

The ADI Information Model is generic, and therefore before implementing it in a particular application must be expanded by application specific types and customized by overriding the predefined components. In the article a best practice rules is defined that allows designers to systematically distribute modifications to separate layers and finally obtain a tidy and comprehensive process representation.

Information Model is a foundation to instantiate the Address Space. Rules governing this process are also addressed in this paper. The focus is on instance declarations and populating the space with components according to the application needs.

Appropriate Information Model adaptation and implementation is a basic requirement to offer ADI ready and interoperable products. From the experience gained during development of the reference implementation it can be stated that this process can be accomplished engaging very limited resources. Thanks to the reference implementation and supporting tools like Address Space Model Designer only basic knowledge of the Address Space and Information Model concepts are required.

Bibliography

[1] Jürgen Lange, Frank Iwanitz, Thomas J. Burke. Von Data Access bis Unified Architecture. Hüthig Fachverlag, 2009.
[2] Wolfgang Mahnke, Stefan Helmut Leitner, Matthias Damm. OPC Unified Architecture. Berlin: Springer, 2009.
[3] OPC UA Specification: Part 1 – Concepts, Version 1.0 or later. OPC Foundation, 2009.
[4] OPC UA Specification: Part 3 – Address Space Model, Version 1.0 or later. OPC Foundation, 2009.
[5] OPC UA Specification: Part 4 – Services, Version 1.0 or later. OPC Foundation, 2009.
[6] OPC UA Specification: Part 5 – Information Model, Version 1.0 or later. OPC Foundation, 2009.
[7] OPC Unified Architecture Companion Specification for Analyser Devices. OPC Foundation, 2009.
[8] OPC Unified Architecture Companion Specification for Devices. OPC Foundation, 2009.