Computing and Informatics, Vol. 37, 2018, 946–968, doi: 10.4149/cai_2018_4_946

OBJECT MAPPING IN THE OPC-UA PROTOCOL FOR STATICALLY AND DYNAMICALLY TYPED PROGRAMMING LANGUAGES

Piotr P. Nikiel

CERN
CH-1211 Geneva 23
Switzerland
e-mail: piotr@nikiel.info

Krzysztof Korcyl

Institute of Nuclear Physics PAN
ul. Radzikowskiego 152
31-342 Kraków, Poland
e-mail: Krzysztof.Korcyl@ifj.edu.pl

Abstract. Two or more object-oriented components located in networked computers can form a distributed system to exchange information and execute methods. The most known approaches include object request broker architectures (e.g. CORBA), messaging-service architecture (e.g. based on ZMQ or JMS) or some variant of Service Oriented Architecture (e.g. SOAP). One of new approaches in the field is the OPC-UA protocol. While having common parts with all aforementioned architectures, it brings very rich and extensible information modelling capabilities, versatility and dynamic address space model, among others. This paper proposes a mapping of information model (applicable in the OPC-UA protocol) into class and object structure of an object-oriented programming language. Special attention is paid to whether given programming language is statically or dynamically typed, with examples and applications in C++ for the former case and Python for the latter. The study also covers the cases of using the proposed mapping at both server- and client-side of OPC-UA software.

Keywords: Middle-ware, OPC-UA

Mathematics Subject Classification 2010: 94-A99

1 INTRODUCTION

There are many architectures and software technologies enabling data exchange between software applications. In the software layering stack they are positioned between a software layer devised to send or receive data (e.g. particular end-user software application) and a layer that enables access to the communication medium (e.g. operating system functions providing access to the network protocol stack or a software library implementing HTTP client). Such a glue layer is often termed "middle-ware".

The software technologies, architectures and programming languages evolved over the years and many new concepts became common, for example object orientation, or common usage of programming languages running in virtual machines, to name a few. In parallel there was a shift towards higher-level programming languages, which are less bound to specific execution environment architecture or even completely independent from it. Middle-ware technologies followed the evolution, profiting both from advancements in programming as well as in ubiquity of network-oriented software, thanks to nowadays presence of the Internet almost everywhere.

Today the middle-ware technologies take part in all steps of data exchange, such as:

- supporting varied data types, possibly including nested or otherwise aggregated types, as well as types defined in run-time, and data serialization[1] of these types,
- processing of serialized data such as timestamping, encoding, encryption or checksum verification at the reception,
- ensuring guaranteed, timely and efficient data exchange, possibly notifying when these conditions cannot be met,
- establishing and shutting down communication channels in varied communication patterns (e.g. one-to-one, one-to-many or other).

The consequences of abandoning the middle-ware layer seem to be quite serious. Then some or most of its tasks have to be carried out by the application layer. The obvious consequence is the lack of abstraction which such a layer would provide, so the application layer would have to deal also with e.g. on-the-wire data encoding concerns. Also modularity and portability would be affected, the former one by not being able to rework data exchange mechanisms without touching the application layer while the latter one by potentially being bound to the initially chosen communication medium.

To take a practical example, let us imagine a system of one data publisher and many receivers, for example: a system distributing currency quotes. A simple substitute for middle-ware solution could be e.g. publishing the data in plain

[1] Serialization is a transformation of given data (e.g. a variable in a computer program) to a stream of octets, such that it can be sent over network and restored at a remote program to a representation which is identical to the original data.

text, with currency pairs in the consecutive lines, where each line would have the currency pair identifier and the bid and ask prices, all comma-separated. An interested receiver would connect to the publisher and wait for the delivery of one or more lines of text. This already brings in many open questions starting from the format of numbers (e.g., which decimal separator to use?) and lines (which newline character(s) to use?) up to what happens when a receiver is interested only in one currency pair? Does it have to receive all the pairs and ignore all but the one of the interest?

Even bigger concern arises when the receiver has to automatically process the data. In our example, the plain text is attractive to humans but no so much for automated processing where it is not only redundant but also ambiguous. Assuming some binary format helps to alleviate some issues but not all of them (e.g., when the receiver is interested only in a part of the data). Generally, it is clear that a more generic solution in middle-ware layer would be desired.

An improvement can be achieved when a generic data serialization approach is used. Such an approach typically requires that the data format (often called a protocol) is described beforehand in a supported notation. The description can be then used to govern the behaviour of the data serializer and deserializer (e.g., the serializer knows the offset of a given data field in the serialized message). More importantly, such a description can be used to obtain programming language bindings to data structures which are to be serialized.

There are many examples of such generic data serializers. One of the most known is the Abstract Syntax Notation One (ASN.1) [1], commonly used as a workhorse of many protocols like SNMP [2] and other. The notation used to describe the data format has a well defined syntax covering any data format which bases on primitive data types (Booleans, integers, etc.) or an aggregation of those (sequence, set, etc.). The description written according to the ASN.1 notation can be used to generate bindings (mappings) in many possible programming languages using the ASN.1 compilers, such that a software developer just refers to field names in generated bindings and not to encoding-specific data. Interestingly, the ASN.1 makes a distinction between the notation itself (being a description of exchanged information) and particular encoding types which are specified in different documents [3]. Therefore one notation can be encoded using many possible ways, including binary formats, XML and other.

A second example is a much newer generic data serializer called Google Protocol Buffers [4]. The Google Protocol Buffers defines a message description format called "proto files". The description format lets define message types that an application would use and then generate mappings for a number of supported programming languages. Such a mapping not only provides an entry point for the programming language but also contains everything which is needed to output or input data in a serialized, binary form.

One must mention that some programming languages support serialization as a built-in feature. The Java programming language with its Java Virtual Machine is a notable example [5]. A significant improvement with regard to both examples cited

above (the ASN.1 and the Google Protocol Buffers) is that in order to serialize given object, only the information from its class is needed, without any prior preparation of external description of the data format[2]. In practice it has an advantage of being able to exchange information as objects between systems in most pristine way with no additional cost in encoding and serialization. However the downside is that the mechanism is specific to the Java programming language and the usage of such serialized objects from another programming language clearly falls beyond the intended purpose of the mechanism.

All three examples shown above illustrate how to interface an important part of middle-ware layer – data serialization and encoding – to a programming language of choice. However nothing was said yet on factual transport of serialized data from one system to another.

Message oriented middle-ware (often abbreviated as MOM) is one of the most common paradigms used in the data exchange between systems. The bottom line of message orientation is that the visible interface from the application layer is expressed in terms of messages having properties like source identifier, destination identifier(s), validity, priority, persistence settings and among others, payload. The payload is where the actual information is to be placed, and most generally it can be seen as an array of octets(bytes). The payload is where higher level data is supposed to be placed after serialization. Therefore combining the aforementioned data serializers with a message oriented middle-ware form a powerful combination which can transport high level data from one application to another while hiding away details of network technology or data encoding concerns.

A common feature of message oriented middle-ware is that it supports many communication patterns like one-to-one, one-to-many, many-to-many, and others. As a consequence, many data exchange problems can be solved efficiently, e.g. in the aforementioned example of currency quotes distribution, using one-to-many communication pattern, only one message publication would be sufficient to update all interested receivers.

There are many notable examples of message oriented middle-ware. One can imagine using UDP/IP datagrams through the means of socket API (e.g. BSD sockets) with some data serializer as a very basic approach. In the recent years, ZeroMQ [6] gained wide interest as a general purpose distributed messaging library, with a focus on the performance and support for diversified communication patterns. When coupled with the aforementioned Google Protocol Buffers (or another general purpose data serializer), the two offer a powerful message-oriented data exchange solution. In Java-related technologies, the state-of-the-art approach uses the Java Message Service [7], abbreviated as JMS. The JMS itself stays as an abstraction layer on concrete distributed messaging implementation, with support for many open-source and commercial messaging solutions. Taking into account Java's

[2] The class must implement java.io.Serializable interface so that this mechanism could work.

support for built-in serialization, the JMS delivers a straighforward approach for exchanging high level data (Java objects) out-of-the-box.

Message oriented middle-ware is not the only paradigm for data exchange between software systems. Object request broker architectures (abbreviated ORB) take a completely different approach, often termed "distributed object" paradigm. The primary difference is that the ORB approach looks at the problem from the perspective of (remotely placed) objects rather than from the perspective of message circulation. The basic role of an ORB is letting interaction between the objects (e.g. calling their methods) no matter where they are - such a feature is called "location transparency". To illustrate how this paradigm might be used for the data exchange between software systems, we can imagine an object A in a system that wants to share data and an object B in a system that wants to obtain the data. In such a configuration, the object B might ("remotely") execute a method "getData()" on the remote object A, obtaining the data as a value returned from such a call.

One of the most notable examples of ORB architectures is the Common Object Request Broker Architecture [8], often abbreviated "CORBA". CORBA requires that interfaces of the interacting objects are specified in an Interface Definition Language, abbreviated IDL. A source file in IDL language is then passed to IDL compiler which generates stub and skeleton code[3] in a chosen programming language. The generated stub, when called upon, is able to pass the call request to a remote object where the factual implementation gets executed.

One of the primary deficits of CORBA as a data exchange solution is that it is focused around objects interaction (i.e. passing method invocations remotely) and not around data exchange per se. It is easy to notice CORBA advantages when it comes to one-to-one, synchronous communication. However many data exchange problems require one-to-many communication which although possible, does not fit CORBA architecture well. Publish-subscribe communication pattern of course can be implemented in CORBA (e.g. following object-oriented publish-subscribe design pattern, also called observer design pattern [9]), but its asynchronous execution (not to slow down the publisher by clients) requires advanced CORBA mechanisms and complex designs. In these factors it is inferior to message-oriented solutions.

WebServices is a more recent paradigm, covering a big number of specific protocols, technologies and software products, like XML-RPC or SOAP. Their common part is profiting from the design choices that were (and still are) responsible for rapid growth of the internet: HTTP, text-based data encoding like XML or JSON and openness of standards. Using XML-RPC or SOAP with a language binding compiler in fact enables to replicate the distributed-object paradigm, known e.g. from CORBA. The XML and JSON allow data exchange with unlimited structuring features, though the overhead of text based data to its binary counterpart cannot be ignored. It is one of the reasons for which better suited protocols are cho-

[3] In CORBA terminology, stub is the caller's interface while skeleton is the callee's interface.

sen when performance or smaller foot-print (e.g. for embedded applications) matter.

One of the newest additions to the catalogue of the data exchange solutions is the OPC Unified Architecture, abbreviated as OPC-UA [10]. OPC-UA is a standard covering wide span of aspects of data exchange between software applications, including support for extensible and rich information models, notifications, support for rich meta-data and multiple encoding manners (including binary and XML).

Rich information model of OPC-UA, covering well beyond object oriented semantics, is especially attractive for its application in data exchange software written in high-level object-oriented programming language. Therefore it is interesting to study possible mappings and bindings between application's classes and objects and their representations exposed to remote systems communicated using OPC-UA. Some inspiration to this study comes from CORBA architecture as well as from other types of mappings in object-oriented programming languages, e.g. from object-relational mappings for database interfacing or from XSD-CXX project providing XML Schema to C++ data binding compiler [11]. The work presented in this paper has its roots in the previously published research on model-based generation of OPC-UA servers [12, 13].

The primary application of the mappings studied in the paper is for a distributed control system of the ATLAS Experiment, one of experiments at the Large Hadron Collider at CERN in Geneva, Switzerland. The OPC-UA is used in the ATLAS Experiment for the data exchange between software components deployed on nodes of the distributed system. The system is currently composed of about 150 machines running Linux; the majority of them run at least one OPC-UA server delivering data obtained from varied types of hardware. A particular challenge of such a system is that it integrates numerous types of data sources; each data source type has different interface (i.e. schema of published information), different amount of data published per unit of time and might be implemented in a different software technology (many of the integrated components are made by external contributors or external companies). The OPC-UA has been selected as the protocol of choice for component integration because of its high-level object-oriented information model and wide adoption in industry; at the same time it enables to establish much weaker coupling between systems than e.g. CORBA, enabling to easily integrate diversified nodes and providing a lot of flexibility.

The mapping approaches studied in the paper attempt to fill the gap between high-level object-oriented programming languages and middle- ware protocols with rich information modelling capabilities. The study focuses on implementation of mapping known from the ORB-like architecture to the new standard OPC-UA. The additional novelty is that the process of mapping is carried out fully dynamically (if permitted by the programming language), without a priori knowledge of exposed object interfaces and type definitions. Few examples of object-oriented systems are given for considerations of practical aspects of proposed approaches.

2 OPC-UA INFORMATION MODEL AND ITS OBJECT-ORIENTED ASPECTS

Information exposed by an OPC-UA server is organized as a graph [10]. The vertices of the graph are called nodes while the edges are called references. Each node is assigned a type, one of: Object, ObjectType, Method, Variable, VariableType, ReferenceType, View and it also has an address. There are some built-in nodes, e.g. the root node symbolizing an entry point to the OPC-UA address space exposed by the server.

A reference (being graph's edge) may be placed only between two nodes. Each reference has a type, either one of built-in OPC-UA types (e.g. HasComponent, HasTypeDefinition) or a custom reference type.

Let us take a simple example of an OPC-UA address-space exposing just one object called "sensor1" of a class "Sensor", having two fields: "id" and "value" and a method "calibrate". Such an address-space would have graph representation according to the OPC-UA information model as in the Figure 1.

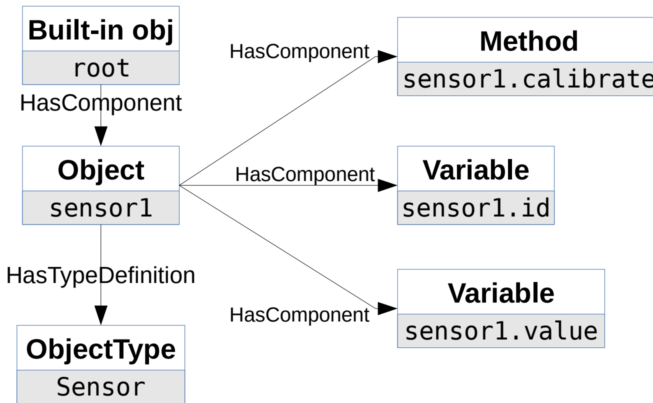

Figure 1. Graph representing OPC-UA address-space described as Example 1

In an OPC-UA server exposing such information model, the graph is stored in the server's OPC-UA address-space. A primary way to obtain the graph by an OPC-UA client is to invoke the "browse request", which takes an address of a node and returns all references originating from the node, including their types and addresses pointed to by them. Therefore, by applying one of the common graph search algorithms (e.g. the breadth-first search or the depth-first search [14]) and starting from the built-in root node, an interested client can discover the whole graph.

However, it is not necessary to know the whole graph to just invoke OPC-UA operations on a remote system: the node address in the address-space is sufficient to uniquely identify any node.

Following operations (also known as transactions) can be invoked on nodes of the address-space[4]:

1. Read and Write of given property of a given node: all nodes support common properties like "description" or "localized name". In addition, variables support "value" property which refers to data stored by a variable.
2. Call: which is a way to invoke methods.
3. Begin, Modify and Stop Monitoring: which implement publish-subscribe functionality in the OPC-UA and let a client be notified about new data.
4. Browse, as explained above.

3 OPC-UA OBJECT MAPPING TO PROGRAMMING LANGUAGE OBJECTS

3.1 OPC-UA Mapping for Distributed-Object Paradigm

The mapping compatible with the distributed-object paradigm is attractive because of its inherent object-oriented properties and proven track record of Object Request Broker architectures (as in e.g. CORBA).

A primary feature of such a mapping should be that an object residing in server application, having a particular interface (methods), could be used through a proxy object residing in client's application. Therefore each invocation of a method should be routed using OPC-UA to a factual remote implementation at server side, as illustrated in Figure 2.

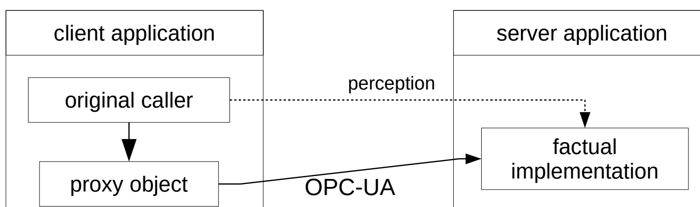

Figure 2. Distributed-object in OPC-UA: factual route of the call operation (solid line) and client-side perception (dotted line)

However, compared to many ORB architectures like CORBA, the OPC-UA information model (including interfaces of objects exposed through the address space) is just a graph stored in the server's memory, and there is no restriction to add or delete new vertices (nodes) or edges (references) in the runtime. The OPC-UA itself does not impose any restriction on requiring the model to be known at compile

[4] The list is not complete, it just enumerates operations attractive in the scope of the study.

time. On the other hand, many commonly used programming languages require type definitions (including interfaces of objects, that is classes) to be known at compile time. Therefore we can speak of many scenarios depending on features of the implementation programming language (separately at client and server side) and chosen mapping approach.

It is worth to emphasize that in this analysis for every high-level class (e.g. "Sensor" from the example in Figure 1) three different type definitions are considered:

- the type definition in the OPC-UA information model, which is stored in the server's address space and can be changed at runtime,
- the type definition for the proxy object at client side, its requirements depend on the chosen programming language,
- the type definition for the factual implementation at server side, its requirements depend on the chosen programming language.

There are three relevant traits to be observed in case of C++ as the implementation language either for client or server side:

- storing a reference to an object (whether achieved by pointers or by C++ reference) does not need a class definition (a forward declaration or opaque-pointer technique can be used),
- calling a method requires the class definition to be known at compile time,
- C++ does not support reflection[5].

Therefore, in C++ implementation at the server side, the type definition for the information model cannot be deduced in run-time (no reflection), and it must be given, along with the source code, with the type definition for factual implementation's class. Similarly, at client side, the proxy object definition must be equivalent to the type definition for the information model at the server side, so both must be given at compile time. C++ implementation brings one more requirement: the type definition in the OPC-UA information model must be constant, otherwise incosistency might arise between all three type definitions.

Java shares many similarities with C++ in the context of the analysis, but it brings an important advantage. Java supports reflection, which in Java not only lets the code to inspect class definitions but also lets invoke the methods discovered using reflection mechanism [15]. Practically it means that a Java object given at runtime could be used as the factual implementation and then mapped as a remote object using OPC-UA. However, the opposite is not straight-forward: creation of a Java class solely on the information from the dynamic information model.[6]

[5] Reflection is a feature of a programming language letting the executed code to inspect itself, e.g. to get a list of methods declared in a given class at run-time.

[6] One could imagine an approach based on bytecode manipulation libraries, however it is not a straight-forward and common solution.

A big difference is achieved when a dynamically-typed programming language is chosen for the implementation. Let us consider Python as an example of such language.

Python enables a number of very attractive features relevant to the study. The first one is the support for a custom definition for accessor methods for object's fields, which enables to intercept get and set operations on the fields ([16] on operator overloading). The methods have standardized names, `__getattr__` and `__setattr__`, respectively. Though having to be defined before the interpreter or compiler is run, they can use run-time data, effectively imitating that the object's list of fields and their types can be altered at runtime. In addition, the act of accessing field's data can be fully customized and refer to the data external to the object, i.e. imitating field access to a remote object. Such application will be further studied in Section 3.2.

The second feature is the support to intercept invocations to call any callable object through a method called `__call__`. When a custom implementation is provided, the call might redirect to a remote invocation of a method through OPC-UA, among other applications.

Let us illustrate the possibilities of combining usage of `__getattr__` and `__call__` in an example as in the UML diagram in Figure 3.

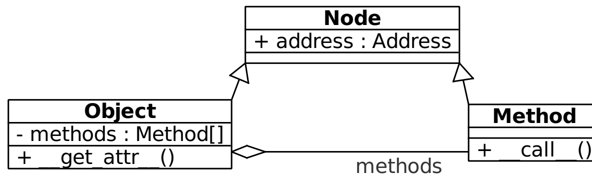

Figure 3. Classes (and their relations) providing remote method invocation capabilities over OPC-UA in Python for distributed-object paradigm

The intention of the example is to illustrate how an object request broking could be facilitated by these features. An example system has an OPC-UA address space with an object called "anObject" with a method called "testMethod". At the client side, in Python program, according to the UML diagram, we construct two objects: "obj" of class "Object" and "method" of class "Method". We insert "method" object into "methods" list in the object. Since both objects are descendants of "Node" class, both have an "address" field. The address of "obj" and "method" point to aforementioned OPC-UA address space nodes "anObject" and "anObject.testMethod".

The aim of the example is to show that invoking "obj.testMethod()" can be given an implementation which performs an OPC-UA transaction invoking respective methods on the server-side.

The invocation of "obj.testMethod()" will first call `__getattr__` method of "obj" object with an argument of "testMethod". A provided custom implementation will intercept the call and identify that requested object's field "testMethod" is in fact

a method (it is in "methods" list) and return its reference, turning it into invocation equivalent to: "testMethod()". In the next step, `__call__` method of "Method" class will be invoked, this time its provided implementation will perform an OPC-UA call transaction on the remote server and return its result. As can be seen, the caller might not even know that the call is effectively handled by a remote method, which is the purpose of the location-transparency feature of distributed-object paradigm.

3.2 OPC-UA Mapping for Data Access

In a similar fashion to mapping object's methods remotely, one could imagine mapping object's fields to OPC-UA variables bound to a particular object.

Let us consider the address-space from example in Figure 1, excluding methods, for C++ programming language. In C++ (or, with minimal changes, Java), we could naively depict an equivalent of "Sensor" class with its instance:

```
class Sensor
{
  public:
  string id;
  double value;
};

Sensor sensor1;
```

However in C++ object's fields are just memory locations and their access (either read or write) cannot do anything else than to access the memory locations. Therefore some additional support is required to synchronize object's fields with their OPC-UA counterpart.

The first approach to consider could be called a snapshot approach, where a copy of data stored in remote object's variables is put on object's user request to local object fields. The implementation synchronizing the contents could be provided either as object's method or as an external function. Such implementation would use OPC-UA synchronous read transaction.

Another possible approach is to use publish-subscribe mechanism in OPC-UA where each notification (carrying new updates) takes care of copying the received data to object's fields. The object's user obtains the data without prior request, behind the scenes. Such configuration would have downsides though, especially because of possible race conditions between two threads – the object's user's thread and the updating thread. Addressing the race condition by a mutex would no longer make it look like behind the scenes update.

A different mapping can be studied for C++, which would map OPC-UA object fields into accessor methods (set, get or both, depending on the information model) rather than to C++ object's fields. Such mapping would drift away from the most

obvious class declaration shown in the listing above, however it would be free from issues of the previous approach:

```
class Sensor
{
  public:
  string getId ();
  double getValue ();
};

Sensor sensor1;
```

It is worth noting that the approach can be applied to server-side and client-side C++ code. Particular examples are detailed in the Section 4.

Similarly to the conclusions drawn in Section 3.1, moving to a more dynamic programming language like Python has benefits also for data-access mapping. Aforementioned `__getattr__` and `__setattr__` methods can be used to intercept access to fields of a Python object, respectively returning or setting the value from/to a remote OPC-UA address-space. To illustrate this, let us again look at the "Sensor" object example (example in Figure 1). When a suitable implementation is provided for the `__getattr__` method, the following statement can invoke a read operation to fetch the remote value and return it like from a plain local object: `sensor1.value`.

When studying mapping for data access, differences in data types between chosen programming language and the OPC-UA types need to be solved. OPC-UA supports many built-in types which have direct equivalents in common programming languages, like integers of varied bit length, floats, Booleans or strings. N-dimensional arrays of those data types are natively supported without need to create custom types. A special "opaque" data type called variant is also supported which can hold any of the built-in primitive types. Therefore no additional work is expected to profit from built-in types in mapped objects in programming language of choice.

However, OPC-UA supports additional ways to define data types which require the type definition to be first available in the address-space. Among them there are so called "simple types" (effectively one of built-in types with a custom name), "enumerations" (resembling "enums" known from C or C++) and "structured data types". A structured data type resembles structures known from the C programming language. In the context of OPC-UA applications, it enables atomic transfer of the whole structure thus guaranteeing that transport layer events like message fragmentation or buffer size restrictions will not deliver an incomplete snapshot of data. This comes at a price however: a structured data type requires that encoding definition (either for binary or XML serialization) is supplied by the application and stored in the address-space. Thus a mapping for data access involving structures requires that in-application serializer is created (using code generation or dynamic

language features or differently).

4 IMPLEMENTATION AND ITS PRACTICAL ASPECTS

So far, the paper explained various conceptual aspects of OPC-UA object mapping for object-request-broker approach and for generic data-access approach, for server side and client side, taking into account relevant features of programming language. This chapter shows how the concepts were implemented.

This paper has roots in the Quasar project, which is a model-based development environment for rapid generation of OPC-UA servers in C++ [12, 13]. A distinctive feature of Quasar relevant to this paper is that Quasar uses a server-side object mapping in the C++ programming language for the data access, e.g. to map read, write operations directly into objects as well as to support the publish-subscribe mechanism in the same approach. As shown in the Section 3.1, for such a configuration, the objects' definition must be known at compile-time. Quasar achieves this by generating classes in C++ from the information model stored in an XML file. The same information model is then stored in the exposed OPC-UA address space and protected from being changed at the runtime, for consistency between C++ classes and the information model.

This study extends Quasar's functionality by adding support for methods. The implementation follows Quasar concepts, that is: methods are declared in the Design file (which is a file primarily storing the information model in the XML format); server build process then generates stubs for implementation and required glue logic. As a result, OPC-UA client request to call a method is mapped in 1:1 relation to a method definition provided in C++ at server side.

To evaluate practical aspects of the mapping for clients in C++, authors have studied a stub generation approach re-using much of Quasar's code generation. As an input, a Design file, in Quasar format is required. For a chosen class (out of all classes defined in the Design file), a proxy class can be generated, wrapping fields access and method invocations into OPC-UA transactions. This effectively creates a companion code for any situation where a client in C++ needs to access data from an OPC-UA server created in Quasar. In this mapping, however, methods in OPC-UA have direct equivalent in the generated stubs while variables have accessor methods instead of fields. In authors opinion it is a most robust approach knowing that C++ does not support overloading field access operator.

For Python programming language at client side, authors have created a library called UaObjects. The purpose of the library is to profit from dynamic nature of Python and let instantiate object mappings to OPC-UA at run-time for clients, both for methods and for data access (read, write, monitored items). The UaObjects extensively uses the approach of overloading `__getattr__`, `__setattr__` and `__call__` methods. PyUaf library [17] is used for OPC-UA interfacing.

The UaObjects library exposes a couple of classes as the API:

- Session – represents an open connection to OPC-UA server

- Node – represents a node in the OPC-UA address-space, stores OPC-UA node address.
- Object – a specialization of Node class representing an object in the address-space.
- Variable – a specialization of Node class representing a variable in the address-space.
- Method – a specialization of Node class representing a method.

The programmer's entry point to the UaObjects library is by instantiation of a Session, supplying PyUaf client handle and server's URI:

```
session = uao.Session(client, server_uri)
```

A primary feature of the Session object is that it can be given an OPC-UA address pointing to an OPC-UA address-space object and perform the object mapping. Such feature is achieved by calling a method `get_object`, e.g.:

```
obj = session.get_object('anObject', 2) # 2 is namespace index
```

The mapping is performed recursively by walking the address-space graph using the Depth-First-Search graph algorithm. As an effect, the whole hierarchy of descendant objects becomes accessible from Python. Thanks to this, a hierarchy of objects can be "walked" like nested classes with no further calls to UaObjects functions.

Let us illustrate the process by an example from a building automation domain. Figure 4 shows a graph of address-space information model.

When `get_object` method is invoked with an address of e.g. "airConditioner" object, the following would happen:

- Browse request is executed on "airConditioner" object, returning three references: one method, one object reference and one variable; they get stored in the object being created.
- The process is recursively repeated for all discovered objects.

Few examples of possible uses of objects mapped by UaObjects library are:

- `airConditioner.turnOn()` will invoke the method over OPC-UA,
- `airConditioner.temperatureSetting = Float(22.5)` will invoke a write operation, any error will be thrown,
- `airConditioner.temperatureSensor.temperature` will invoke a read operation over OPC-UA and return the obtained value or None, or throw for an error.

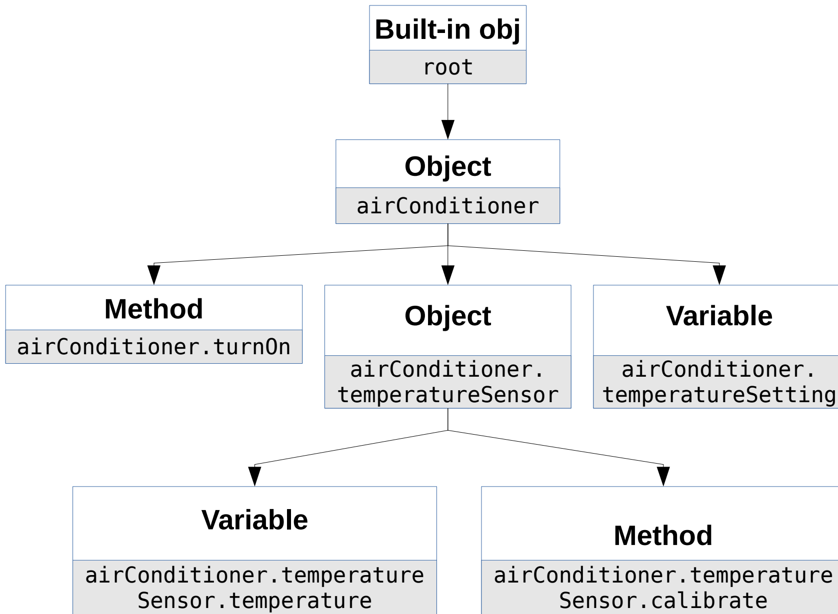

Figure 4. An example OPC-UA information model to illustrate UaObjects library features. The text in greyed fields corresponds to OPC-UA addresses of the nodes (apart from "root" address, which in the OPC-UA is encoded as a numeric identifier, here shown as a string, for simplicity).

5 PERFORMANCE CONSIDERATIONS

Performance is usually an important factor in taking a decision whether to use or to avoid given protocol or solution. Authors have measured the performance of various OPC-UA based approaches studied in this paper and compared it against CORBA and ZMQ/Protocol Buffers under most fair conditions. The primary goal was to measure the time of operation comprising passing payload of given size from either client to server or server to client (both directions were tested separately). Since solely synchronous operations were tested (thus obtained time is the complete duration of the whole operation), obtained operation durations have been used to calculate throughput.

The factual ways of passing data depended on the protocol under test:

- For OPC-UA, write and read operations were under test as well as calling methods either taking arguments (to pass data from a client to a server) or returning values (to pass data from a server to a client). In all situations, a server under test was created with Quasar, with either UASDK or open62541 used as

OPC-UA communication stacks (backends)[7]. As a client, a C++ based client was used (with object mappings generated at compile-time, as described in the paper) or a Python based client with object mappings created in run-time (using UaObjects library, as described in the paper).

- For ZMQ+Google Protocol Buffers, a protocol description file has been prepared and then compiled with the "proto" compiler for C++. The protocol description file containted just one message type consisting of an operation selector (i.e. put data, get data) and the data. For client to server data transfer, a message with "put data" operation with a payload would be sent, and a confirmation message with no payload would be returned. For server to client data transfer, a message with "get data" operation with no payload would be sent, and a confirmation message with payload would be returned. For the transport mechanism ZMQ's request-reply was chosen on top of TCP/IP.
- For CORBA, an interface description file has been prepared and then compiled with the IDL-C++ compiler. OmniORB 4.1.6-2 has been used as the CORBA provider at both client and server side with GIOP as the protocol. Just one interface has been used, taking a string as an argument and returning a string, depending on chosen operation ("get data" or "put data"), a payload would be put in either arguments (for client to server data transfer) or in the return values (for server to client data transfer).

The common configuration of all tests were:

- 1 server and 1 client computer; the server was an 8-core Intel i7 machine at 3.6 GHz clock with 24 GB of RAM, the client was an 4-core Intel i7 machine at 2.4 GHz clock with 8 GB of RAM. Both machines ran Linux operating system;
- physical network connection: Gigabit Ethernet, direct cable connection between both machines. There were no other active network connections;
- request-reply communication pattern was used;
- end-user payload size was considered as the sweeped parameter;
- all test programs were compiled at best optimization levels available;
- all test programs verified size of data at reception with the value configured for given test; an inconsistency would have aborted the test.

The durations of operations are presented in the Figure 5.
The following conclusions could be made from both figures:

- For very small payload sizes (up to 32 B), three clusters of results are visible: ZMQ-based results as clear winners with operations durations of about 100 μs; remaining solutions based on C++ (including CORBA and OPC-UA with C++ clients) with operations durations of about 300 μs; OPC-UA results with client in Python with durations of about 600 μs. Since the only difference between the

[7] At the time of writing, methods support was available only with the UASDK backend.

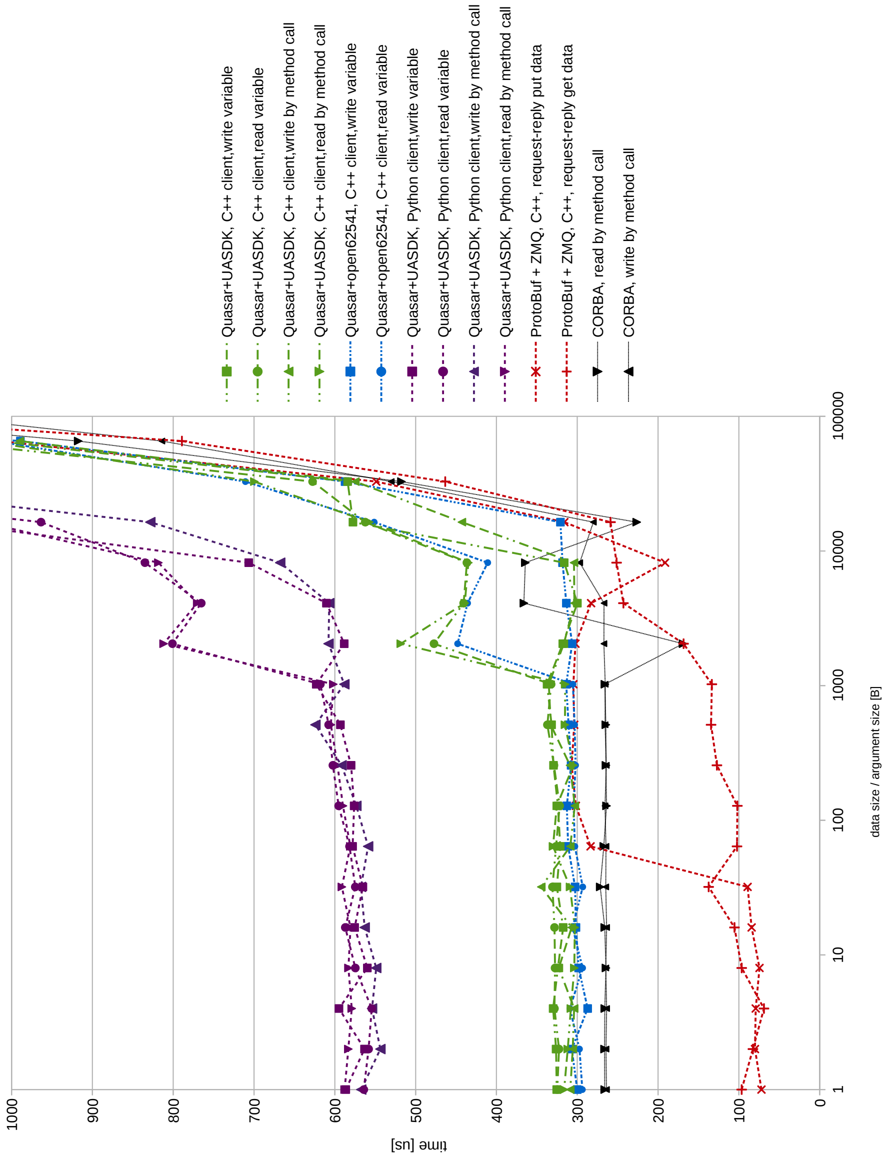

Figure 5. Time required to complete one operation of a given type vs size of payload data. Note X axis is logarithmic and within the range of 1 B–100 kB to emphasize the lower end of the measurements.

second and the third group of results is the programming language (C++ vs. Python), we assume it is the primary factor contributing to the results and not the protocol itself.

- For payload sizes from 64B to 1024B, ZMQ's "put data" implementation joins the second group of results while "get data" implementation still stays ahead by at least twice shorter operation time. The origin of the asymmetry is not clear. Apart from this, observations from the point above still hold true.
- For payload sizes bigger than 1 024 B network related delay becomes the primary contributor. It is worth noticing that apart from Python implementation, all other results converge, however ZMQ-based implementation to transfer data from server to client is still the fastest one.

Based on the results, throughput figures were charted in the Figures 6 and 7. Please note that for chosen communication medium (Gigabit Ethernet) the maximum throughput of the medium itself is about 125 MB/s[8] and in fact, for big payload sizes, the throughput figures converge to about 80–85 % of the medium throughput for all C++ implementations of all 3 chosen protocols.

The following overall conclusions can be made:

- For a data exchange application based on "distributed object" principles (calling remote methods to send/receive data), CORBA and OPC-UA have similar performance. Significant differences apply to both protocols in aspects different from the performance.
- For applications where latency matters most ZMQ surpasses both CORBA and OPC-UA by a very significant factor.

Please note that measurements described in the chapter focused on synchronous request-reply communication patterns. Some of the measured solutions (e.g. ZMQ) offer truly asynchronous data transfers (e.g. ZMQ's Pub-Sub pattern) which, however, have completely different principle of operation and different properties. For example, a subscriber has no control of the rate at which it is receiving data; also error handling is at a different level, e.g., a subscriber might not be getting any data because of either publisher does not have any new data or there is a network issue. Therefore ZMQ's asynchronous communication patterns have not been included in the comparison.

6 CONCLUSIONS

In the paper, authors have studied application of rich information modelling features of OPC-UA to mapping classes and objects defined in OPC-UA address-space into

[8] This, of course, is a rough simplification: on top of the medium, Ethernet frame headers, IP packet headers and TCP headers contribute to what amount of bandwidth is left for payload data.

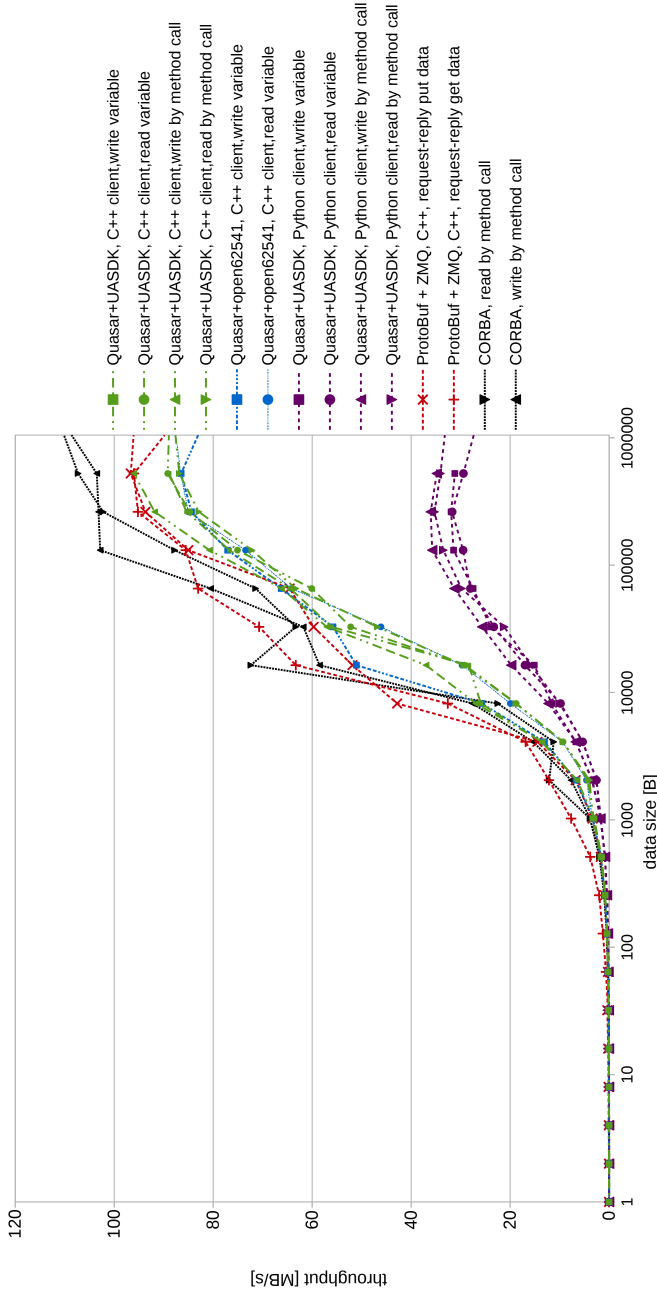

Figure 6. Throughput expressed in payload data size per second as a function of the payload size. The X axis is logarithmic.

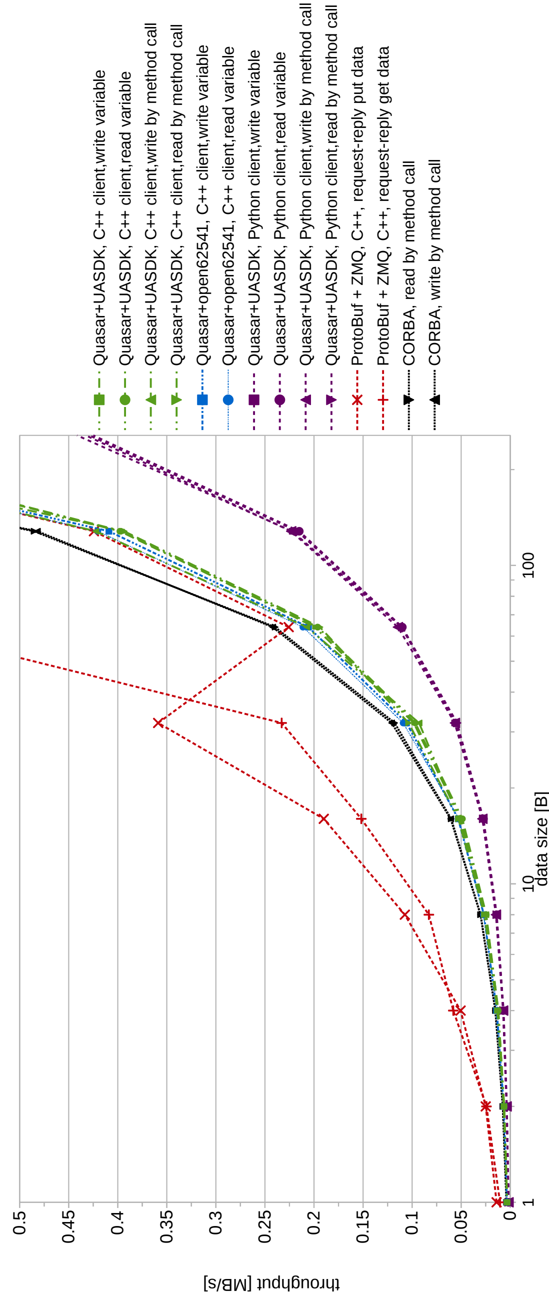

Figure 7. Throughput expressed in payload data size per second as a function of the payload size. The X axis is logarithmic and within range of 1 B–256 B to illustrate the lower range of payloads. The source data is the same as in the Figure 6.

classes and objects of chosen programming language. A number of conclusions have been identified:

- Strongly-typed programming languages (as C++ in the study) enable a direct method mapping to OPC-UA methods, however direct fields mapping has a number of practical down-sides. Authors identify that indirect mapping using accessor methods is a more robust solution.
- Weakly-typed and dynamic programming languages (Python in the study) enable direct mapping of both methods and fields to their OPC-UA counterparts. In addition, a number of "convenience" features is possible like run-time type conversion and possibility to walk the OPC-UA address-space graph by just refering to object fields.
- Languages with support for reflection enable to create the corresponding OPC-UA information model based on programming language class.

The following deliverables are additional effects of the study:

- methods support for Quasar environment,
- UaObjects library for Python,
- code generator enabling client-side mappings to OPC-UA for C++.

7 FURTHER STUDY

Authors would like to point out that the OPC-UA gives a possibility to modify the address-space by client requests[9]. Such feature opens further possible improvements:

- Adding or deleting object instances in run-time by a client might be attractive in many scenarios. For example, it might be more practical to configure the address-space contents by a client than by changing server's configuration. Such scenario easily fits to all cases studied in the paper because it does not change (or add or delete) types definitions.
- Adding, deleting or altering type definitions in run-time by clients would not be compatible at least with C++ mappings studied in the paper, however, there is no obvious obstacle for such application in Python.

One could also imagine an extension of the UaObjects library towards Python's meta-classes concept. In such approach, creation of Python's classes in run-time based solely on information from OPC-UA information model could be attempted.

[9] In a multi-user networked environment, it seems natural that such a feature should be restricted only to the clients having certain (i.e. elevated) privileges. In the OPC-UA this restriction is implemented by requesting authentication when a session (i.e. an OPC-UA connection) is created, for example by using the username-password authentication. Then the server will allow only certain users to modify the address-space, for example by checking whether the username bound to the session through which such request comes is in the set of users with elevated privileges.

REFERENCES

[1] International Telecommunication Union: Information Technology – Abstract Syntax Notation One (ASN.1): Specification of Basic Notation, Recommendation ITU-T X.680, also known as ISO 8824-1.

[2] Douglas, M.—Schmidt, K.: Essential SNMP. 2nd ed., O'Reilly, 2009. ISBN 978-0-596-00840-6.

[3] International Telecommunication Union: Information Technology – ASN.1 Encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER), Recommendation ITU-T X.690, also known as ISO 8825-1.

[4] Google: Protocol Buffers, Website, `https://developers.google.com/protocol-buffers/`, accessed 27-Dec-2016.

[5] Java Object Serialization Specification, Java SE v.8, `https://docs.oracle.com/javase/8/docs/technotes/guides/serialization/index.html`.

[6] Hintjens, P.: ZeroMQ: Messaging for Many Applications. O'Reilly, 2013. ISBN 978-1-449-33406-2.

[7] Monson-Haefel, R.—Chappell, D. A.: Java Message Service. O'Reilly, 2001. ISBN 978-0-596-00068-5.

[8] Henning, M.—Vinoski, S.: Advanced CORBA Programming with C++. Addison-Wesley Professional Computing Series, 1999. ISBN 978-0-201-37927-9.

[9] Gamma, E.—Helm, R.—Johnson, R.—Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994. ISBN 978-0-201-63361-2.

[10] Mahnke, W.—Leitner, S.-H.—Damm, M.: OPC Unified Architecture. Springer-Verlag, 2009. ISBN 978-3-540-68898-3, doi: 10.1007/978-3-540-68899-0.

[11] Code Synthesis Tools CC: C++/Tree Mapping Getting Started Guide, available at: `http://www.codesynthesis.com/projects/xsd/documentation/cxx/tree/guide/`.

[12] Nikiel, P. P.—Farnham, B.—Filimonov, V.—Schlenker, S.: Generic OPC-UA Server Framework. Proceedings of 21st International Conference on Computing in High Energy and Nuclear Physics (CHEP2015), Okinawa, Japan, 2015. Journal of Physics: Conference Series, Vol. 664, 2015, Art. No. 082039.

[13] Nikiel, P. P.—Farnham, B.—Schlenker, S.—Soare, C.-V.—Filimonov, V.—Abalo Miron, D.: Quasar – A Generic Framework for Rapid Development of OPC-UA Servers. Proceedings of ICALEPCS 2015, Melbourne, Australia, 2015.

[14] Cormen, T. H.—Leiserson, C. E.—Rivest, R. L.—Stein, C.: Introduction to Algorithms. MIT Press, 2009. ISBN 978-0-262-03384-8.

[15] Horstmann, C. S.—Cornell, G.: Core Java^{TM}: Volume I, Fundamentals. 9th ed., Prentice Hall, 2012. ISBN 978-0-13-708234-6.

[16] Lutz, M.: Python: Pocket Reference. 5th ed., O'Reilly, 2014. ISBN 978-1-449-35701-6.

[17] Pessemier, W.—Deconinck, G.—Raskin, G.—Saey, P.—Van Winckel, H.: UAF: A Generic OPC Unified Architecture Framework. Software and Cyber-infrastructure for Astronomy II., Proceedings of the SPIE, Vol. 8451, 2012, Art. No. 84510P, 10 pp.

Krzysztof Korcyl is Adjunct in Institute of Teleinformatics of Cracow University of Technology, Poland and in Institute of Nuclear Physics PAN, Cracow, Poland where he received his habilitation in physics. He worked for the third level trigger of Delphi experiment at LEP and subsequently for TDAQ system of ATLAS experiment at LHC at CERN, Geneva. His research interests include modeling of large scale real time systems and applicability of FPGA and GPGPU technologies for improving performance in data acquisition and filtering systems.

Piotr P. Nikiel is Software Engineer in the Detector Control System of the ATLAS experiment at LHC at CERN, Geneva. He received his M.Sc. in computing from Cracow University of Technology, Poland and his M.Sc. in electronic engineering from AGH University of Science and Technology, Cracow, Poland. He is software architect of the Quasar project and author of many OPC-UA based applications used at CERN. His research interests include model-based software engineering, embedded systems and programming languages.