Towards a Modelica OPC UA Library for Industrial Automation

Bernhard Thiele[1]

[1]Institute of System Dynamics and Control, German Aerospace Center (DLR), Germany,
bernhard.thiele@dlr.de

Abstract

Open Platform Communications Unified Architecture (OPC UA) is often named as a prospective enabler for future automation systems integrations, as for example envisioned in the German Initiative Industrie 4.0. The DLR OPC UA Modelica library connects OPC UA with the Modelica world. There are two main goals: First, OPC UA server capabilities for emulating the communication interface of (physical) hardware components in order to create component simulations, *e.g.*, for virtual commissioning. Second, OPC UA client capabilities for interacting with real-world hardware components, *e.g.*, for process visualization and monitoring or interactive simulation and control purposes. The library works on Windows and Linux platforms. It is tested using the Modelica environments Dymola and OpenModelica.
Keywords: OPC UA, Industry 4.0, Robotics, Modelica

1 Introduction

Open Platform Communications Unified Architecture (OPC UA) is often named as a prospective enabler for future automation systems integrations. It is considered as an existing technology which can cover various communication aspects for flexible production lines as envisioned in the German Initiative Industrie 4.0 and is also prominently mentioned in the Reference Architecture Model Industrie 4.0 (ZVEI 2015).

The significance of OPC UA as a core communication protocol for future automation systems has prompted the development of a Modelica library with the goals of enabling:

- Component simulations: The simulated component has an *OPC UA server* instance which mimics the *OPC UA server* of an actual system component down to some level of acceptable fidelity. Modelica is used for modeling the component's behavior in order to provide realistic values for the simulated process variables. Consumers of the data are OPC UA clients, *e.g.*, in supervisory control and data acquisition (SCADA) sytems. Possible scenarios are Hardware-in-the-Loop (HIL) simulation or virtual commissioning.

- Interactive simulation and control: The application has an *OPC UA client* instance which interacts with physical hardware components. The Modelica-based application uses actual hardware process variables which it queries from an OPC UA server instance on the real hardware component. Possible scenarios include querying quantities from hardware components for process visualization and monitoring and (model-based) high-level control tasks.

The term *component simulation* is used in the sense as defined in (Harrison and Proctor 2015), where emulation is defined as "the production of artificially created signals to represent the physical presence of some part of the manufacturing process" and a simulation of one component with emulation capabilities is termed as a component simulation.

The following paragraphs briefly describe existing approaches of supporting OPC UA connectivity in Modelica environments.

The OpenModelica environment (Fritzson et al. 2020) supports an option for starting an embedded OPC UA server which maps the simulation variables into an address space which can be monitored by OPC UA clients. In addition, a connected client can control the progress of the simulation by setting specific simulation control variables through the OPC UA interface. The feature was added during the OpenCPS project and is briefly described in its deliverable report (Sjölund and Asghar 2018). It is worth noting that the server implementation is at the tool level, *i.e.*, OpenModelica specific. Additionaly, while the goal of our library-based OPC UA server approach is to model the communication interface of (physical) hardware components (in order to create component simulations for HIL simulation and virtual commissioning), the goal of the OpenModelica tool-based OPC UA server is to facilitate debugging and monitoring of (embedded) OpenModelica real-time simulations.

The Modelica OPC UA libraries from Wolfram (Wolfram Research 2021) and ESI (ESI Group 2021) provide OPC UA client functionality. Their goals regarding the Modelica OPC UA client interface are similar to the goals of our own library. The significant difference is that in their present state OPC UA server capabilities are not in the scope of these libraries.

One common trait of all the approaches for connecting OPC UA to Modelica (including our own), is that they rely on the open62541 open-source implementation of OPC UA (*open62541* 2021) as underlying technology stack.

2 OPC UA

Besides being a hardware-independent communication protocol, the interesting capability of OPC UA is the ability of semantic information modeling. This information modeling allows an object-oriented style of modeling devices including hierarchical composition, object types (≈ classes), type hierarchies (≈ inheritance), instantiation, and (customizable) relations between objects. Indeed, if desired, there is the option of using the well-established Unified Modeling Language (UML) as base for OPC UA information model design as described by Pauker et al. (2016).

The base elements of OPC UA's meta model are nodes. These nodes are connected by typed references resulting in an undirected graph forming the *OPC UA Address Space*. Several node classes are predefined by the standard. Each node has a set of *attributes* which depend on the node class. They can be mandatory or optional. An attribute which is mandatory for any node is its *NodeId* for uniquely identifying the node.

The information model can be extended by so-called *companion specifications*. Simply speaking, one could compare those to libraries in conventional programming languages, *e.g.*, they usually define new object types which can be instantiated. Anyone, *e.g.*, device manufactures, can define own extensions. However, companion specifications particularly facilitate domain specific standardization.

A lot of work in our group is centered around robotic applications. Hence, integrating the OPC UA Companion Specification for Robotics (OPC 40010-1 2019) is of particular interest. The corresponding specification work is driven by the VDMA Robotics Initiative with the goal of specifying an OPC UA information model for complete motion device systems (including, but not limited to, conventional industrial robots), split up into several parts (Part 1 to Part n). At the time of this writing, the group has so far completed and released Part 1. It provides a basic description of a motion device system with the aim of pushing condition data vertically into higher level manufacturing systems. Future extension will cover further use cases, *e.g.*, to configure and control a robot. An example exploring interesting possibilities is given by Profanter et al. (2019) who propose an extension which provides a standardized (hardware-agnostic) control interface for robot manipulators.

3 Overview

Figure 1 gives an overview over the library structure and shows a basic server example.

The package browser at the left side of Figure 1 shows an `OPCUAServer` and `OPCUAClient` block which can be dragged and dropped into the diagram layer. These are the central blocks in the library for creating an OPC UA server or client, respectively. The `LeanLoggerInit` block ensures that messages within the external C code

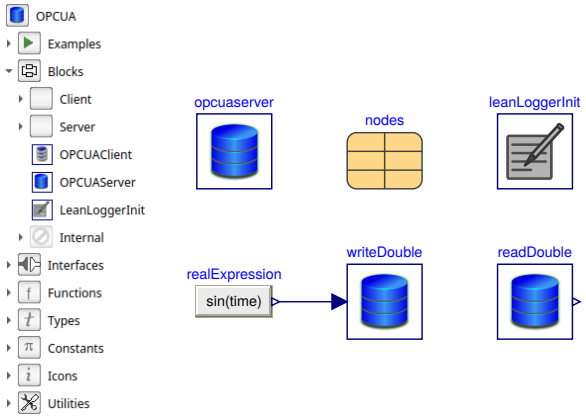

Figure 1. Library structure (left) and basic OPC UA server example (right).

are forwarded to the Modelica environment and also registers the Modelica environment's implementation of the `ModelicaAllocateStringWithErrorReturn()` to the interfaced dynamic link library (DLL, Windows), or shared object library (SO, Linux). The `Functions` package contains the function interface to the external C code.

The right side of Figure 1 shows the diagram layer of a basic server example. The `opcuaserver` component is an instance of the `OPCUAServer` block. It is declared as "`inner`", hence it can be accessed in all deeper levels of the model's instance hierarchy as an "`outer`" element. The `nodes` component is a configuration record. It is passed to the `opcuaserver` component as a parameter. It contains information about OPC UA nodes which shall be created on the server during initialization.

At the bottom are instances of blocks for writing and reading variables of type *Double*. These blocks contain parameters which specify the *NodeIds* of the *variable nodes* which are the target (source) of the write (read) operation.

An OPC UA server is started by simply simulating the model. Usually real-time synchronization is desired which can be either provided by the capabilities of the simulation environment or by external code, *e.g.*, using the Modelica_DeviceDrivers library (Thiele et al. 2017). After starting the simulation, the OPC UA server will listen on a specified port for client connections (default port number is 4840).

A good general purpose OPC UA test client is the freely available UaExpert from Unified Automation (*UaExpert* 2021). It can provide a plethora of information in different configurable views. Figure 2 shows a possible view on the connected basic server example. The "Project" pane (upper left window) shows the connected server(s) ("open62451-based OPC UA Application"). The "Address Space" pane (lower left window) allows browsing through the nodes of the server's information model. Nodes from the "Address Space" pane can be drag-and-dropped into the "Data Access View" (DA View) pane (right window). The DA View creates a subscription and

DOI 10.3384/ecp21181205

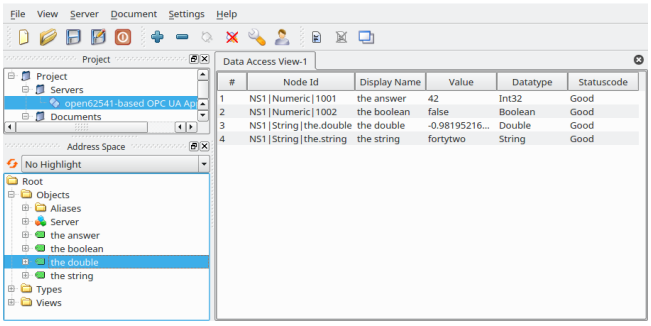

Figure 2. Unified Automation's test client UaExpert connected to the basic OPC UA server example.

allows monitoring (configurable) aspects of the nodes. In this example there are four monitored nodes. Only the node with the display name "the double" changes its value during the simulation. This is the node referenced in the `writeDouble` block of Figure 1. The remaining nodes in the DA View are defined in the `nodes` record (including an initial value), but they are not written to during simulation time.

The DA View also shows the pivotal *NodeId* attribute in the "Node Id" column. *NodeIds* refer to a namespace with an additional identifier value that can be an integer, a string, a guid or a bytestring. In the example two nodes are using an integer identifier ("Numeric") and two are using a string identifier ("String"). All shown nodes are in the namespace index "1" ("NS1"), the namespace reserved for the local server.

4 Server and Client

On the one hand the library provides OPC UA server functionality with the goal of modeling the communication interface of (physical) hardware components. The main task is providing simulated process variables to external devices, *e.g.*, for HIL simulation or virtual commissioning. On the other hand the library provides OPC UA client functionality with the goal of querying actual process variables from hardware components, *e.g.*, for process visualization and monitoring or (model-based) high-level control tasks. It is possible to use server and client blocks within the same Modelica model.

4.1 Server

Figure 3 gives more details about some server related blocks. The left-hand side robot denotes a placeholder for an arbitrary physical model with process variables which are published by an OPC UA server running on the physical device. The `nodes` record instance is an approach for collecting *nodes* which shall be created on the server in one central data structure. The `opcuaserver` has two main parameters: `portnumber`, for specifying the server's listening port, and `nodes`, a configuration record for defining own *nodes* and *namespaces* on the server. Hence, the declaration in the model is:

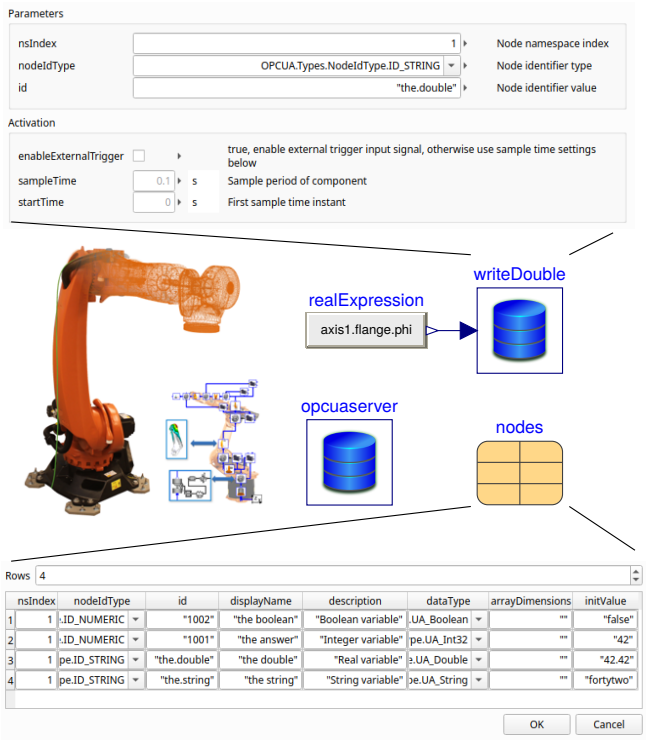

Figure 3. OPC UA server: The `node` record specifies a list of nodes which are created on the server. Other blocks, like `writeDouble`, can use these *NodeIds*.

```
inner Blocks.OPCUAServer opcuaserver(
    portNumber=4840, nodes=nodes);
```

The `writeDouble` block needs to specify a *NodeId* (using parameters `nsIndex`, `nodeIdType`, `id`) which identifies the node to which it periodically writes its input[1]. It is an error if this node does not exist on the server or if it is not compatible.

The record instance `nodes` is an instance of `OPCUA.Types.Nodes`. Its structure is shown in Listing 1. The annotations are hints to editing tools for creating a convenient graphical user interfaces (GUI) for filling the variable sized arrays, *e.g.*, the dialog for the `VariableNode vars[:]` array is the one displayed at the bottom of Figure 3.

Listing 1. Nodes configuration record.

```
record Nodes
  String nsUris[:] = fill("", 0) annotation
      (Dialog(enable=true));
  VariableNode vars[:] = fill(
      OPCUA.Types.VariableNode(), 0)
      annotation (Dialog(enable=true));
end Nodes;
```

There are limits and compromises in this approach. First most, it cannot be used to create arbitrary OPC UA

[1]Parameter `nsIndex` identifies the namespace index ("1" denoting the namespace reserved for the local server), `nodeIdType` the *NodeId* type (here either "Numeric" or "String"), `id` the identifier value, hence these parameters correspond to the attributes displayed in the "Node Id" column of Figure 2.

nodes. Instead, it aims at supporting a subset of *variable node* types which have a rather straightforward mapping to primitive Modelica types (including arrays of these types). Also notice, that it uses strings at places where this seems not to fit in all cases (`id`, `arrayDimension`, `initValue`). This is a compromise so that the columns can encode values of different data types, *e.g.*, `Boolean`, `Integer`, or `Real`.

4.2 Client

Figure 4 gives more details about some client related blocks. The left-hand side denotes the server side to which

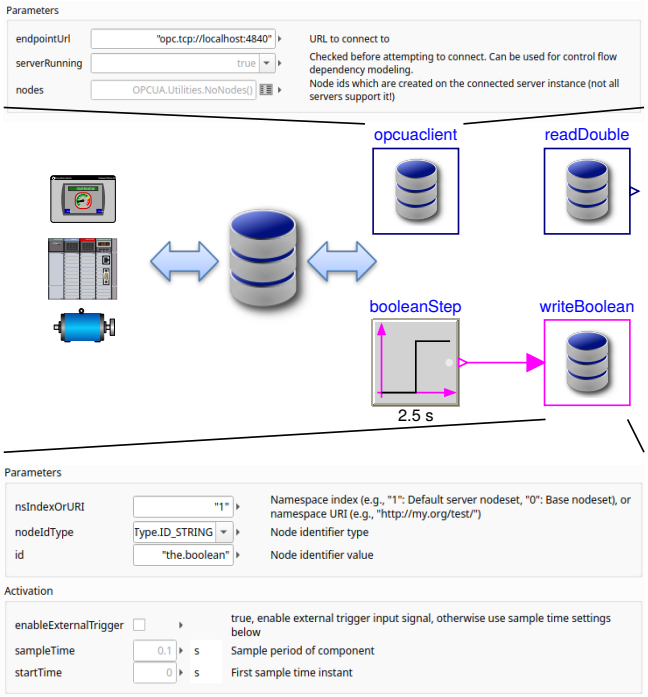

Figure 4. OPC UA Client: The `opcuaclient` block connects to an OPC UA server. Blocks like `writeBoolean` can access nodes on the server by their *NodeId*.

the client connects, which typically provides hardware related process variables.

The `opcuaclient` needs to specify the endpoint URL of the server instance. In the example a local server listening at port 4840 is expected. Variable `serverRunning` is not a `parameter`, it has continuous-time variability. Its main use is in Modelica models which combine `opcuaclient` and `opcuaserver` blocks in one model. In this case, it can be used to ensure that the server is ready, before the client (in the same model) connects to it. Given the respective access rights, it is also possible to create new nodes on the server. For this purpose a configuration record can be passed as parameter `nodes` (default: no nodes are created). Identical to the server case the record needs to be an instance of `OPCUA.Types.Nodes`.

Often the main interest is in reading process variables, but it is also possible to write data to the server as indicated by the `writeBoolean` block. Comparing Figure 4 with Figure 3 shows that the client and server blocks for accessing variables have a similar interface.

Sometimes *NodeIds* used by the server are known a priori by the client. This is for example the case for standardized information models, including the OPC UA specification itself, as well as companion specifications, or vendor specific information models. However, in practice the client often has no a priori knowledge of the *NodeIds* used by the server for variables of interest. Instead, the client browses the address space of the server programmatically in order to find *NodeIds* corresponding to objects and variables of interest. This is feasible since the address space is represented hierarchically, allowing for simple and complex structures to be discovered and utilized by OPC clients (see the "Address Space" pane in Figure 2).

In particular, nodes in the address space can be discovered by *browse paths*, *i.e.*, by following a sequence of named references (*browse names*) from a start node to hierarchically subordinated nodes. A basic starting node for searching is the root objects folder. Its *NodeId* is known, because it is defined by OPC UA specification. The presented Modelica library browse paths delimited by '/' can be used for discovering *NodeIds*, *e.g.*, starting from the root objects folder the browse path "Server/ServerStatus/State" can be used for retrieving the *NodeId* assigned to the status code variable of the server.

There are different types of references which can model different types of hierarchical composition, *e.g.*, for modeling component composition, folder organization, or instance hierarchies. This allows a fine-grained filtering based on the type of reference. However, at present the DLR OPCUA library does not discern between different types of references when trying to resolve a browse path.

4.3 OPC UA for Robotics

Since a lot of work in our group is centered around robotic applications, integrating the OPC UA for Robotics Companion Specification (OPC 40010-1 2019) is of high interest. The robotics companion specification itself depends on a companion specification featuring an information model for devices (OPC 10000-100 2020). Integrating these companion specifications is a considerable effort and there are different possible approaches.

Accompanying to the textual specification documents there exist XML-based information model definitions according to the OPC UA Nodeset XML schema. These so called *nodeset files* encode OPC UA information models and are understood by respective tools. The open62451 distribution includes an *XML Nodeset Compiler*, a python-based tool, which can generate C code (including C header files) from such XML files. This C code needs to be included in the build process for compiling working server applications. Hence, for supporting the desired companion specifications it is required to modify the build process so that code is generated from the respective nodeset files and this code needs to be included in the compilation pro-

DOI 10.3384/ecp21181205

cess.

The present prototype uses an approach in which a C++ wrapper of the robotics information model encapsulates required function calls to the "low-level" interface of the underlying open62451 library. All code, including dependencies to generated code from the XML Nodeset Compiler and the open62541 library, is assembled in one shared library (see section 5 for more details). The developed C++ classes themselves are encapsulated by a plain external C interface which is compliant to the Modelica external function interface. These C functions are called from respective Modelica functions and are used for creating Modelica external objects.

The Modelica functions can then be used for creating an OPC UA for Robotics compliant information model on the server. Figure 5 shows an example of such an information model as seen by a connected client.

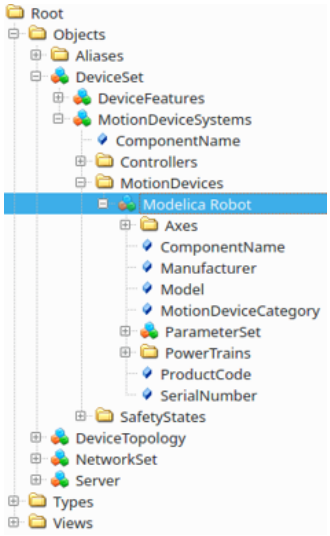

Figure 5. OPC UA for Robotics compliant server example as seen by a connected test client (UaExpert).

5 Function Interface

Figure 6 shows an excerpt from the comprehensive `Functions` package. This package contains definitions for external objects and functions operating on these objects.

Table 1 lists several notable external objects and their underlying (open62451) data structures. The open62451 data structures can be recognized by the library's naming convention of using the prefix "`UA_`" for its exported symbols. Modelica external objects are opaque pointers to some address in memory, so (in principle) the underlying C data structures can be changed or extended without the need of changing the Modelica code. Possible changes may even include to swap out the underlying OPC UA library (though there is at present no intention for such a step).

While `NodeId` and `OPCUAClient` are directly mapped to open62541 data structures, the `OPCUAServer` external object is mapped to a C++ wrapper structure which contains additional information. After a configuration phase, the server loop is started in a dedicated thread by a function called `OPCUAServer.run(...)`. The identifier of the spawned thread is saved in the struct member `threadID` and struct member `running` is set to "true".

The struct member `pubSub` is a composite object which aggregates data structures and logic related to the OPC UA Publish/Subscribe (PubSub) extension. PubSub extends the OPC UA client/server architecture with facilities which (among other things) can enable low-latency communication. Results of an experimental low-latency open62451 PubSub implementation are reported in (Pfrommer et al. 2018). The PubSub extension is usable as an experimental feature within the DLR OPC UA library, but it is not yet intended for real-world use-cases.

A view on the DLR OPC UA library's layered architecture is shown in Figure 7. Only core components are shown, components like the logging facilities or experimental components are suppressed.

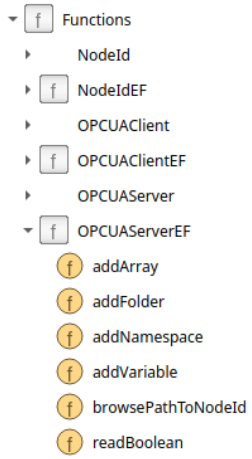

Figure 6. Excerpt of the `Functions` package. It contains external object classes and external functions (EF) operating on these objects.

Table 1. Notable external objects and their opaque pointer mappings.

Modelica	*C/C++ → open62451 (UA_...)*
`NodeId`	→ `UA_NodeId`
`OPCUAClient`	→ `UA_Client`

`OPCUAServer`	→ C++ struct with server related settings:
	`struct uam_server {`
	`  std::thread threadID;`
	`  UA_Server *server;`
	`  UA_Boolean running;`
	`  uam_PubSub *pubSub;`
	`};`
	Particularly, it includes a member of type pointer to `UA_Server`.

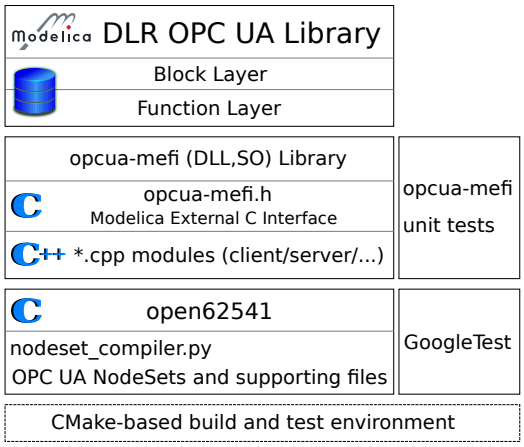

Figure 7. DLR OPC UA library's layered architecture.

The project build, test and packaging automation is managed by the CMake cross-platform family of tools (*CMake* 2021).

The top Modelica OPC UA library uses the Modelica external C functions interface (MEFI) of the underlying *opcua-mefi* library. The *opcua-mefi* library is a dynamic link library (DLL) on Windows, or a shared object library (SO) on Linux. Its Modelica external function interface compliant application programming interface (API) is declared in the header file *opcua-mefi.h*.

While the *opcua-mefi* library declares a Modelica compliant external C function interface, the internal library consists of C++ code. This code wraps and adapts open62451 facilities into structures which can be conveniently used from Modelica. One may say it provides a Modelica-oriented high-level interface to a subset of the open62451 library. Although invisible to a Modelica library user, it is actually the most extensive part of the DLR OPC UA library.

An important part of the *opcua-mefi* library is its unit tests. These tests use the GoogleTest framework (*GoogleTest* 2021) and its CMake integration in order to provide a convenient testing environment on the supported platforms. It integrates nicely with various development environments, *e.g.*, JetBrain's CLion or Microsoft Visual Studio.

The base technology stack is provided by the open62451 open-source library from the open62451 project (*open62541* 2021). It is an impressive open-source C (C99) implementation of OPC UA, licensed under the liberal Mozilla Public License v2.0. Despite its good documentation and a large set of indispensable examples, there is a considerable learning curve for using the library. Though a good part of the learning curve can be attributed to the inherently large and complex OPC UA standard itself.

6 Application Example

The Factory of the Future project (DLR 2021) is a cross-sectoral research project within the German Aerospace Center (DLR). The aim is to develop a wide range of digital production technologies, robotic systems and robotic applications for flexible and networked manufacturing processes, and to demonstrate them in 'lead scenarios'.

One cross-sectoral scenario which is investigated is an assembly process for a motor saw. The scenario includes (physical) robot cells from the Institute of Robotics and Mechatronics (DLR-RM) and the Center for Lightweight-Production-Technology (DLR-ZLP). OPC UA is used as interoperability standard between the different robot cells and involved institutes. The task of our institute, the Institute of System Dynamics and Control (DLR-SR), is the modeling of the assembly process with appropriate fidelity. Our goal with this work is to explore digital twin applications based on physically accurate models.

Modelica is used as modeling language for the physics-based digital twin. There are several challenges for enabling the intended applications, among them:

- Modeling the assembly process requires efficient object manipulation capabilities which can accommodate real-time data updates.
- The Modelica-based simulation model needs to connect and synchronize with the real-world entities and processes.

The first issue lead to the development of a new Modelica library for manipulation tasks, which is outside the scope of this work (Reiser 2021). The present work is concerned with finding a solution to the second issue.

As a first step towards the complete assembly process, one robot cell has been connected at the time of this writing. Figure 8 shows the considered robot cell which has been set up in DLR-RM's lab. The depicted robot is from the recent generation of DLR-RM's light-weight robots and bears the project name SARA (Safe Autonomous

Figure 8. Robot cell from DLR-RM synchronized with Modelica-based real-time simulation model (upper-right screen) from DLR-SR using the DLR OPC UA library for connectivity.

DOI 10.3384/ecp21181205

Robotic Assistant), (Iskandar et al. 2020). In the upper-right corner a screen shows a visualization of DLR-SR's simulation model (*i.e.*, the "digital twin") which is synchronized with real-time data from the SARA robot cell.

Figure 9 shows the OPC UA related excerpt of the Modelica model used for the demonstration depicted in Figure 8. There are two OPC UA client blocks which

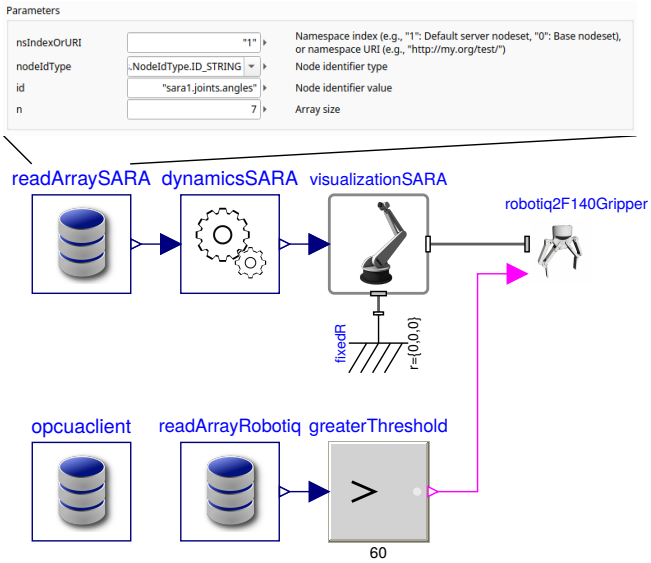

Figure 9. Excerpt of the OPC UA related parts of the Modelica model used for the demonstration depicted in Figure 8.

read server variables stemming from the SARA robot cell: `readArraySARA` reads the joint angles of the robot arm, `readArrayRobotiq` reads the position of the gripper. At present, a simple threshold is used for giving an indication whether the gripper is closed or open. Besides the OPC UA blocks, the figure also shows a composite block with a (simple) dynamics model for the SARA robot, as well as a block which is responsible for the 3D visualization of the robot arm.

A rather complex block shown in this excerpt of the complete model is the gripper block. It implements the manipulation mechanics and interacts with the assembly parts. The assembly parts, as well as the conveyors and tables in the scene are not shown. All those parts are not in the scope of this work, they are part of the aforementioned library for manipulation tasks. In addition, some visualization related blocks are suppressed. The visualization is provided by a prototype of the next generation of the DLR Visualization library (Hellerer, Bellmann, and Schlegel 2014; Kümper, Hellerer, and Bellmann 2021).

Block `readArrayRobotiq` reads an array variable, but the array has only one element (the gripper position, a value in the dimensionless range $[0, 100]$). Essential parameters of the `readArraySARA` block are shown at the top. Notice, that a simple OPC UA approach is used in which the seven joint angles are packed into one array which is identified by a statically fixed string-based *NodeId*. Therefore, neither the OPC UA Robotics extension described in subsection 4.3 is used, nor is there any need for sophisticated node discovery mechanics on the client side.

In summary, the described demonstration was a step towards the envisioned Factory of the Future scenario. In particular, it showed the feasibility of using OPC UA as interoperability standard. Since the DLR OPC UA library also supports OPC UA server functionality (see subsection 4.1), it was possible during development to model the SARA robot cell including its OPC UA server interface and connect it with the client application of Figure 9 within the same model. This simplifies application development, because there is no need that the actual robotic hardware is available. Further work, extending the presented base functionality for exploring more complete digital twin related scenarios is ongoing.

7 Discussion

At the beginning of this library development effort there was the long-term vision of (automagically) generating Modelica models from devices described in established or future automation standards. Since OPC UA is an important standard for the communication aspect, the idea was to explore the generation of Modelica models from OPC UA information models, encoded in *nodeset files* (*i.e.*, *NodeSet2.xml* files), from automation devices and machinery.

After short initial research into available third party library and tools the idea emerged of developing a simple Modelica-oriented C interface on top of the open62451 API. This interface should particularly support the fundamental data types from Modelica (`Boolean`, `Integer`, `Real`, `String` scalars and arrays) and translate between these Modelica types and OPC UA types.

7.1 First Steps

The first steps with the open62451 library were very smooth thanks to good documentation (including working examples) and a polished CMake-based build system. However, striving for more general OPC UA support, including some more advanced constructions, quickly becomes more intricate.

OPC UA defines low-level aspects, like *Int16*, *UInt32*, *Int64*, which cannot always be mapped satisfactorily to Modelica (*e.g.*, the Modelica external function interface defines that `Integer` are mapped to C `int`, hence signed 32-bit integers on common Linux and Windows platforms). For not being overly restrictive on the allowed OPC UA variable types (potentially unsafe) conversions are used at respective places in the *opcua-mefi* library. For mitigation, safe variable value ranges can be checked dynamically in the C code and runtime errors can be risen when violations are detected.

Besides OPC UA built-in types which have a rather straightforward mapping[2], where also exist built-in types

[2]Modelica `Boolean`: OPC UA *Boolean*; Modelica `Integer`:

with no such mapping, e.g., *NodeId* to Modelica. These types require additional design decisions, *e.g.*, *NodeId* is mapped to a Modelica external object. Other OPC UA built-in types, *e.g.*, *XmlElement*, are simply not supported at this state.

7.2 Refactoring

Furthermore, there is a huge flexibility how respective variable nodes can be defined or discovered in OPC UA and often very similar (but not identical) functions and patterns are used for achieving a certain task on either the server or the client. This lead to quite a lot of repetitive code in the *opcua-mefi* library which at some point was addressed by using C++ templates and its code generation facilities for achieving more generic and succinct code.

On top of this first interface an experimental Modelica code generator was written which takes a *NodeSet2.xml* file as input and generates a skeleton of Modelica code with the intent of simplifying and accelerating the development of component simulations and physics-based digital twins. While this worked for the very limited number of elements considered for the experiment, it also became apparent that a more complete (industry-relevant) Modelica code generator could hardly be based on the facilities of the present *opcua-mefi* library.

7.3 Another Approach Needed?

The open62541 library itself uses code generation at various places for providing an API which can encompass the comprehensive OPC UA standard. The key here is that OPC UA standard information is not only English text, but partly already encoded in machine processable files, most notable, *NodeSet2.xml* files. One could compare these to a "standard library" in programming which itself is based on more fundamental principles (syntax and semantics of the underlying programming language). Hence, using an appropriate mechanization, C code can be generated from relevant machine processable files.

This could also be key for enabling a more generic and complete Modelica interface. Instead of the high-level oriented API of the *opcua-mefi* library, one could try to interface the lower-level open62451 more directly and use code generation techniques for gaining a Modelica function interface which is closer to the open62451 API.

Although it seems unrealistic to expect that this would magically solve all problems, a clever approach in this direction could push the limits.

7.4 Domain-Specific Extensions

Instead of striving for a level of generality which would allow taking a *NodeSet2.xml* file and generate suitable Modelica code, another option is to manually develop library support for selected (standardized) domain-specific information models of interest. Although it might be a significant initial development effort for supporting a new domain, it can result in well-thought-out reusable library blocks for quickly modeling devices which adhere to the standardized domain-specific interface.

This is the approach used for the integration of the Robotics Companion Specification as described in subsection 4.3. Compared to the generic approach, it is easier to achieve and can be a good alternative if the expected use-cases adhere to such domain-specific information models.

7.5 Outside of Modelica

Another approach with a different angle is using dedicated automation-oriented simulation platforms and rather import Modelica models. Using the Functional Mock-up Interface (FMI) standard for such a purpose suggests itself. For example, Hensel et al. (2016) explore an approach of integrating FMI-based co-simulation with the SIMIT simulation platform from Siemens using OPC UA as a generic middleware technology.

Using a dedicated integration platform can be a practical and flexible alternative if a such a platform is available. The discussed Modelica library approach might be more appealing in Modelica-centric development processes, or if using an additional platform seems too costly or complicated[3].

8 Conclusions and Outlook

Work on the presented Modelica library was started with no prior experience with OPC UA technology. Thanks to available resources, like the open-source open62451 project, or the freely available OPC UA test client from Unified Automation GmbH, the first steps were rather smooth and quick.

However, moving to slightly more advanced concepts it quickly became apparent that the OPC UA standard and related tooling has an intimidating complexity, and it took longer towards the current state of the Modelica library with a more moderate progress than expected.

Indeed, there are plenty of more OPC UA features and aspects which are not yet explored or implemented within the library, or simply not covered for not exceeding the scope of this paper. Among them, supporting the PubSub extension, which has been briefly mentioned in section 5. OPC UA PubSub extends the applicability of OPC UA beyond a strict client/server model and also sketches a direction towards low-latency communication schemes (Pfrommer et al. 2018). These are hot topics with no final conclusion and ongoing discussions within standardization bodies (Bruckner et al. 2019).

A good amount of the motivation for this work is based on the anticipation that OPC UA will play a crucial role for future automation systems. In this respect, exploration of the underlying concepts and technology has been an im-

OPC UA *SByte*, *Byte*, *Int16*, *UInt16*, *Int32*, *UInt32*, *Int64*, *UInt64*; Modelica `Double`: OPC UA *Float*, *Double*; Modelica `String`: OPC UA *String*.

[3]Notice that it is still possible to export a Modelica model with OPC UA interface blocks as Functional Mock-up Unit (FMU) and import it into a co-simulation environment.

DOI 10.3384/ecp21181205

portant driving factor in the development. Consequently, the library has the status of an experimental in-house technology prototype. So far, its runtime stability has been pleasantly reliable (credits to the open62451 project), but the interface, structure, naming, documentation, and the supported feature set is not fixed, yet. Nevertheless, the library can be made available to interested partners.

Future plans with the library include further exploration of more advanced concepts, as well as following (and potentially integrating results of) ongoing standardization efforts with a particular interest for robotic applications and real-time industrial communication.

Acknowledgements

This work was partially funded by the DLR project Factory of the Future. The author would like to thank Robert Reiser and Tobias Bellmann for feedback and support during the development of the library. For the described application example the author would like to thank all who contributed to the demonstration. Particularly, Robert Reiser who implemented the Modelica application example model, Oliver Eiberger and Timo Bachmann who provided CAD data of the robot cell, and Korbinian Nottensteiner and Stefan Schneyer who implemented the OPC UA interface of the robot cell. Finally, the author would like to thank the reviewers for the constructive comments to improve the manuscript.

References

Bruckner, Dietmar, Marius-Petru Stănică, Richard Blair, Sebastian Schriegel, Stephan Kehrer, Maik Seewald, and Thilo Sauter (2019). "An Introduction to OPC UA TSN for Industrial Communication Systems". In: *Proceedings of the IEEE* 107.6, pp. 1121–1131. DOI: 10.1109/JPROC.2018.2888703.

CMake (2021). URL: https://cmake.org/ (visited on 2021-04-15).

DLR (2021). *DLR Factory of the Future*. URL: https://factory-of-the-future.dlr.de/ (visited on 2021-04-15).

ESI Group (2021). *SimulationX OPC-UA Client Modelica library*. URL: https://doc.simulationx.com/4.2/1033/Content/Libraries / InterfacesGeneral / Communication / OPCUA / open62541.htm (visited on 2021-06-24).

Fritzson, Peter et al. (2020). "The OpenModelica Integrated Environment for Modeling, Simulation, and Model-Based Development". In: *Modeling, Identification and Control* 41.4, pp. 241–295. DOI: 10.4173/mic.2020.4.1.

GoogleTest (2021). URL: https://github.com/google/googletest/ (visited on 2021-04-15).

Harrison, William S. and Frederick Proctor (2015). "Virtual Fusion: State of the Art in Component Simulation/Emulation for Manufacturing". In: *Procedia Manufacturing* 1. 43rd North American Manufacturing Research Conference, NAMRC 43, 8-12 June 2015, UNC Charlotte, North Carolina, United States, pp. 110–121. ISSN: 2351-9789. DOI: 10.1016/j.promfg.2015.09.069.

Hellerer, Matthias, Tobias Bellmann, and Florian Schlegel (2014). "The DLR Visualization Library - Recent development and applications". In: 10th *Int. Modelica Conference*. Ed. by Hubertus Tummescheit and Karl-Erik Årzén. Lund, Sweden. DOI: 10.3384/ECP14096899.

Hensel, Stephan, Markus Graube, Leon Urbas, Till Heinzerling, and Mathias Oppelt (2016). "Co-simulation with OPC UA". In: *2016 IEEE 14th International Conference on Industrial Informatics (INDIN)*. Poitiers, France, pp. 20–25. DOI: 10.1109/INDIN.2016.7819127.

Iskandar, Maged, Christian Ott, Oliver Eiberger, Manuel Keppler, Alin Albu-Schäffer, and Alexander Dietrich (2020). "Joint-Level Control of the DLR Lightweight Robot SARA". In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 8903–8910. URL: https://elib.dlr.de/138637/.

Kümper, Sebastian, Matthias Hellerer, and Tobias Bellmann (2021). "DLR Visualization 2 Library - Real-Time Graphical Environments for Virtual Commissioning". In: 14th *Int. Modelica Conference*. Ed. by Martin Sjölund, Adrian Pop, Lena Buffoni, and Lennart Ochel.

OPC 10000-100 (2020). *OPC Unified Architecture – Part 100: Devices*. Tech. rep. Release 1.02.02. OPC Foundation.

OPC 40010-1 (2019). *OPC UA for Robotics Companion Specification Part 1: Vertical integration*. Tech. rep. OPC Foundation.

open62541 (2021). URL: http://open62541.org (visited on 2021-04-15).

Pauker, Florian, Thomas Frühwirth, Burkhard Kittl, and Wolfgang Kastner (2016). "A Systematic Approach to OPC UA Information Model Design". In: *Procedia CIRP* 57. Factories of the Future in the digital environment - Proceedings of the 49th CIRP Conference on Manufacturing Systems, pp. 321–326. ISSN: 2212-8271. DOI: https://doi.org/10.1016/j.procir.2016.11.056.

Pfrommer, Julius, Andreas Ebner, Siddharth Ravikumar, and Bhagath Karunakaran (2018). "Open Source OPC UA Pub-Sub Over TSN for Realtime Industrial Communication". In: *23rd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE. Turin, Italy, pp. 1087–1090. DOI: 10.1109/ETFA.2018.8502479.

Profanter, Stefan, Ari Breitkreuz, Markus Rickert, and Alois Knoll (2019). "A Hardware-Agnostic OPC UA Skill Model for Robot Manipulators and Tools". In: *24th IEEE International Conference on Emerging Technologies And Factory Automation (ETFA)*. IEEE. Zaragoza, Spain. DOI: 10.1109/ETFA.2019.8869205.

Reiser, Robert (2021). "Object Manipulation and Assembly in Modelica". In: 14th *Int. Modelica Conference*. Ed. by Martin Sjölund, Adrian Pop, Lena Buffoni, and Lennart Ochel.

Sjölund, Martin and Adeel Asghar (2018). *Real-time debugging and monitoring*. Technical Note D4.2 (M36). ITEA3, Project 14018: OPENCPS project.

Thiele, Bernhard, Thomas Beutlich, Volker Waurich, Martin Sjölund, and Tobias Bellmann (2017). "Towards a Standard-Conform, Platform-Generic and Feature-Rich Modelica Device Drivers Library". In: 12th *Int. Modelica Conference*. Ed. by Jiří Kofránek and Francesco Casella. Prague, Czech Republic. DOI: 10.3384/ecp17132713.

UaExpert (2021). URL: https://www.unified-automation.com/products/development-tools/uaexpert.html (visited on 2021-04-15).

Wolfram Research (2021). *Wolfram OPCUA Modelica library*. URL: https://reference.wolfram.com/system-modeler/libraries/OPCUA/OPCUA.html (visited on 2021-06-24).

ZVEI (2015). *The Reference Architectural Model Industrie 4.0 (RAMI 4.0)*. Tech. rep. Zentralverband Elektrotechnik- und Elektronikindustrie e.V. (ZVEI).