Implementation of Service Orchestrated control procedures in OPC UA for JGrafchart

Johan Hagsund

LUND UNIVERSITY

Department of Automatic Control

MSc Thesis
ISRN LUTFD2/TFRT--5953--SE
ISSN 0280-5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

Abstract

The automation industry is facing many challenges with higher demands on their production process. Technology used today does not allow for fast changes in the production line. This thesis will investigate how services can be modelled using a new standard OPC UA for data exchange. Encapsulation of the mechatronic functions as services will allow for creating control software using a SOA approach. An experimental set-up will investigate how an OPC UA server and client are created.

Acknowledgements

This master thesis has been carried out as a cooperation between the Smart Factory in Kaiserslautern, Germany, and the department of automatic control at Lund University. It has given me the opportunity to work in the front of very important research topic.

I would like to thank Moritz Ohmer and Lisa Ollinger from the Smart Factory for guiding me in my work in Kaiserslautern and for baring with me when the German skills were not enough. I would also like to thank Benjamin Kiepke for assiting me with all kinds of questions that came up during the project while still in Germany.

I would also like to thank Alfred Theorin from the department of automatic control for all the input via our Skype meetings and for all the support with JGrafchart, the analysis of the results and discussions about the report.

Acronyms

DFKI	Deutsches Forschungsentrum für Künstliche Intelligenz
DOS	Disk Operating System
DPWS	Devices Profile for Web Services
HMI	Human Machine Interface
OPC AE	OPC Alarm and Events
OPC DA	OPC Data Access
OPC HDA	OPC Historical Data Access
OPC UA	OPC Unified Architecture
PLC	Programmable Logical Controller
SCADA	Supervisory Control And Data Acquisition
SDK	Software Development Kit
SOA	Service Oriented Architecture
SOA-AT	Service Oriented Architecture in Automation Technologies
URI	Uniform Resource Identifier
WSDL	Web Service Description Language

Contents

1

Introduction

1.1 Introduction

The requirements on modern automation applications are changing, the control systems are getting more and more advanced. At the same time the control systems needs to be developed faster, reducing the set-up time an the time it takes for modifications. Companies are expanding all over the world setting up production cells in multiple countries. Together with an increased threat level for industrial espionage this calls for automation systems with high security features that can connect plants all around the world. This thesis will not provide the solution to all these issues, it will focus on how services can be modelled in OPC Unified Architecture (OPC UA), a new standard for data exchange and information integration, and also to implement a control software in JGrafchart for a device from Firschertechnik using OPC UA for communications.

The first section will focus on how services can be modelled in OPC UA. The second part will focus on how OPC UA has been implemented together with JGrafchart for controlling the device from Fischertechnik. During the time of this project twelve weeks have been spent in Kaiserslautern at the SmartFactory at DFKI, the German Institute for Artificial Knowledge, to learn about OPC UA and to program the implementation of OPC UA. The second part has been spent in Lund for analysis and conclusions of the work.

1.2 Smart Factory

*"...the intelligent factory of the future" [*SmartFactory^{KL}*]*

The SmartFactory is a manufacturer-independent research and demonstration plant established with the goal to transfer new smart technologies into the industrial environment. SmartFactory works in close collaboration with multiple companies to put new ideas into practise in joint projects, the SmartFactory works in the area between research and industry [*SmartFactory*^{KL}].

1.3 Methodology

This project will be structured in two parts where the first part has a more theoretical focus and where the second part focus on implementation. In Kaiserslautern the first weeks was mainly spent on studying OPC UA in order to understand the concept of it as well as getting familiar to other technologies used in the industry today. Once the basic theory was understood the programming started for creating an OPC UA client and server that could be used for the project. A lot of the programming was done in Kaiserslautern since they were more familiar with OPC UA already. OPC UA theory was studied in parallel throughout the entire course of the project since there is always more to be learned. During the last weeks in Germany focus was to create a working set-up where an application in JGrafchart could control a simulated device on the server. Once back in Lund the analysis of the results started as well as the creation of the report.

1.4 Outline

This outline presents an overview of the different parts in this thesis. The thesis is divided into two parts where the first part studies OPC UA and investigates how services, in theory, can be modelled in OPC UA. In the second part a working set up with an OPC UA server and client is to be created.

Chapter 1 contains a short introduction and presents the outline of the project.
Chapter 2 presents the background of this thesis describing the trends in the industry today.
Chapter 3 goes through the different theories and technologies used during the project such as SOA, DPWS, OPC-UA and Grafchart.
Chapter 4 present the problem description for this thesis.
Chapter 5 discusses OPC UA in more detail and describes the available building blocks.
Chapter 6 gives a short introductions to the different tools used during the process.
Chapter 7 presents different ways services and be modelled in OPC UA.
Chapter 8 is where the second part of the thesis starts with the creation of an OPC UA server and client as well as describing the physical device from Fischertechnik used during the project.
Chapter 9 goes through the results from the project.
Chapter 10 discusses the results and the conclusions are presented.
Chapter 11 is the final chapter containing suggestions for future work.

2

Background

2.1 Background

There is a trend in the manufacturing industry today towards shorter product life cycles, an increased customer demand of individualized goods and a more complex production. In order to meet these demands from more complex production processes the requirements on the automation systems are increasing. A higher degree of reusability must be achieved in order to manage the demands for shorter set-up times [Ollinger et al., 2011] [Ollinger and Zühlke, 2013]. In order for the companies to stay successful they need to be able to adapt and reconfigure their production lines quickly to match the changing demands. The increase complexity in the automation systems calls for a more advanced control system to handle it.

The classic way for controlling such production processes is the use of programmable logical controllers (PLC). Programming of such controllers is done at a very low level and in the very last step of the implementation. There are no tools for combining the information from the planning stage into the implementation on the PLC. The code is very dependent on the hardware and needs to be developed from the beginning for each new project [Ollinger and Zühlke, 2013].

The modern manufacturing industry can not allow for such time consuming engineering processes of developing new code for each new project due to the demands for fast reconfiguration of the production process. The limited possibility of integrating the different stages in the development and planning of a new production line together with the inability to reuse already created code calls for a new technology [Theorin and Johnsson, 2012] [Theorin et al., 2013]. It is estimated that the cost for set-up and installations sums up to almost one third of the total cost during the life time of a plant [Jammes et al., 2005]. Replacing broken parts with new also makes up for a substantial part of the cost. It could be that the exact same part is no longer available and with a hardware specific control system this requires a substantial engineering effort only for a small replacement of a part.

There is a new concept that has shown some success in this field of improving the level of reusability of control systems as well as making the production process more flexible and it is called Service Oriented Architecture (SOA). SOA has the potential to solve many of the issues the industry is facing. It is a new technology for the automation world, an industry that is known for its slow acceptance for new technologies [Theorin and Johnsson, 2012], that need more testing but some studies has shown promising results [Bohn et al., 2006]. Investigating how services can be modelled in OPC UA can be one step in convincing the industry to use this approach. Integrating it into JGrafchart for service orchestration to create control software together with creating a real application showing the results could open their eye for this new way for controlling a plant.

3
Theory

3.1 Introduction

This chapter will go through the main technologies used during this project such as SOA, SOA-AT, DPWS and OPC UA. SOA is a software concept where specific functionality is made available as services to others. SOA-AT deals with how SOA can be implemented in the automation industry so these two makes up the base on how services can be described. DPWS is a technology that implements SOA-AT and is in this project studied in order to make a comparison with how is could be done in OPC-UA which is the last technology. In the end of this chapter Grafchart is described which is used in the second part this thesis for the implementation in JGrafchart.

3.2 Service Oriented Architecture

Service oriented architecture (SOA) is not a technology or standard by it self [Ollinger et al., 2011], it is a concept for high-level software architecture. The basic principle of SOA is that the software is broken down to smaller discrete pieces of code that each describes a small part of the applications functionality. This functionality is modelled and made available to other parts of the application in form of a service. Breaking down large projects into smaller pieces in form of services is called service orientation [Wikipedia, 2014]. Each service contains some meta data describing the functionality it is encapsulating. To create applications using the SOA concept multiple services are combined and used together which is called service orchestration [Theorin, 2013].

Encapsulation of the functionality of the field devices in form of services raises the abstraction level of the application programming allowing for the same services to be used in another application using similar devices. This means a higher degree of reusability, reduced development times [Wikipedia, 2014] as well as a more flexible production [Theorin, 2013].

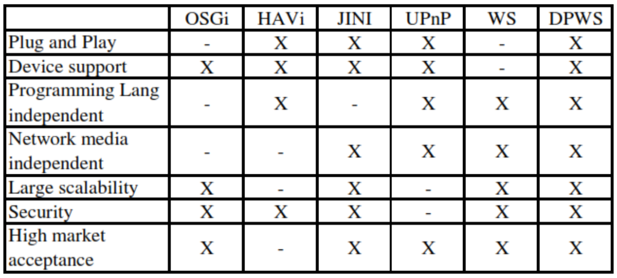
3.3 Service Oriented Architecture in Automation Technologies (SOA-AT)

Applying the concept of SOA within industrial automation has the potential to solve many of the issues the industry is facing with an increased demands on flexibility and shorter set-up times [Theorin et al., 2013]. SOA is originally developed to be used for business processes but the idea of breaking down the process into smaller pieces can be applied in the automation industry as well and is then called Service Oriented Architecture in Automation Technologies (SOA-AT). The difference when applying the SOA paradigm in the automation industry compared to a regular business processes is that the place where the services are executed matters. Instead of executing the services on a large computer as in a regular business processes they are now executed on small embedded devices with limited resources connected with the different field devices [Theorin, 2013]. The physical functionality of the field devices such as actuators and sensors is modelled as basic services that connect the automation systems with the real processes [Theorin et al., 2013]. These basic services are sometimes referred to as mechatronic functions since they have a direct contact with the process [Ollinger and Zühlke, 2013], they represent the actual actions of the real machinery. Multiple basic services can be modelled together and form a composed service. This is done to raise the level of abstraction of the written control application. The control software is created by orchestrating multiple services together [Theorin, 2013]. Each service has a number of operations that can be executed for accessing the functionality of the device.

The use of SOA-AT has not spread so much yet, the industrial automation industry is rather slow on adapting new techniques and prefer to use technologies that have been around for some time and known to work well in practice. There have been two large research projects in this field founded by EU, SIRENA and SOCRADES [Theorin, 2013], that have shown some promising results when it comes to implementing SOA-AT. SIRENA resulted in an implementation of Devices Profile for Web Services (DPWS) meant for embedded devices. DPWS turned out to be the best solution for device integration according to the results in Figure 3.1 from SIRENA [Bohn et al., 2006].

3.4 Devices Profile for Web Services

DPWS is a service technology for implementing SOA-AT. Figure 3.2 illustrates the hierarchy of DPWS where the top-level called device represent the different field devices. Each device has a number of hosted services encapsulating the functionality of the device. It is important to differentiate between a device and a service. A device is representing a real unit such as a motor and a service is representing the functionality of the device. A service is built up from many operations that represent different actions that the service can execute. The different operations are grouped



	OSGi	HAVi	JINI	UPnP	WS	DPWS
Plug and Play	-	X	X	X	-	X
Device support	X	X	X	X	-	X
Programming Lang independent	-	X	-	X	X	X
Network media independent	-	-	X	X	X	X
Large scalability	X	-	X	-	X	X
Security	X	X	X	-	X	X
High market acceptance	X	-	X	X	X	X

Figure 3.1 Results from SIRENA

[Bohn et al., 2006]

in port type where similar operations are grouped together for a better overview [Theorin, 2013].

In DPWS each device must contain meta data describing it self and it must also be discoverable since it is possible for a client to probe a local network for all available devices. The device contains a Web Services Description Language (WSDL) file with the description of all the services and operations offered by the device [Theorin, 2013].

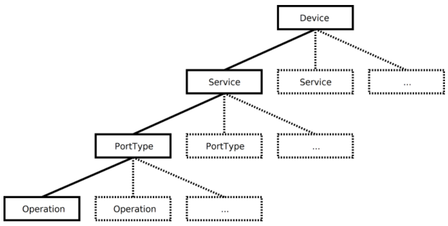


Figure 3.2 DPWS hierarchy

3.5 OPC Unified Architecture

Classic OPC

A first version of OPC entered the market in the mid nineties and it was an immediate success offering a standardized interface on how data was exchanged allowing for a more plug and play set-up. The problem at the time was that many different protocols, bus systems, and interfaces were used so each application needed its own solution for exchanging data. This problem can be compared with the problem using printer drives during the DOS days where each system needed its own solution in form of a printer box for each printer. Windows created a solution for this by integrating the support of printers into the operating system. The same goal was the drive for developing OPC, to create a standardized interface for industrial automation applications [Mahnke et al., 2009].

The OPC Foundation developed this standard for making information from the process level available for the HMIs and SCADA systems. OPC is based on the COM and DCOM technologies from Windows and is mainly used within industrial automation. The OPC Foundation has developed a specification targeting three application areas where the first and main one is OPC Data Access (OPC DA) that standardize the way data is made available for reading and writing. The other two are OPC Alarm and Events (OPC AE) and OPC Historical Data Access (OPC HDA). The information from the devices is always modelled on the server and made available to the client [Mahnke et al., 2009] that could be the HMIs or the SCADA systems. All computers running Windows OS had built in support for the COM and DCOM technologies and this was a big factor for the early success of OPC [Mahnke et al., 2009].

OPC DA OPC DA is the most used one out of the three OPC specifications. it makes up almost 99% of the products using OPC. As the name states OPC DA is all about making data available for the client to access. It allows real-time monitoring of data values from different devices as in Figure 3.3.

OPC AE OPC AE offers and interface that can receive event notification and alarms as the name suggests shown in Figure 3.3. It could be for example that an alarm occurs because the temperature is too high or that the level of a tank is too low [Mahnke et al., 2009]. It also allows for sending an acknowledgement when an alarm is received. The OPC AE in usually implemented as an extra feature next to OPC DA.

OPC HDA In comparison to OPC DA, OPC HDA gives access to stored data instead of real-time data in Figure 3.3. There are three different options that OPC HDA makes available to the client. The first one is only to offer access to stored raw data. The second is to allow the client to read data from multiple variables at a specific time stamp and the last option is aggregating values from the stored data for one or more variables [Mahnke et al., 2009].

OPC UA

Classic OPC was a great success when entering the market, offering a standardized interface between different application in the automation industry. OPC is implemented on thousands of products with great results being used in more fields than first planned to. Lately many vendors felt left out due to the fact that OPC only ran on Windows OS using the ageing technologies COM and DCOM. They are no longer in development which means that OPC will soon become an outdated standard. These two needs together with the desire to model more complex sys-

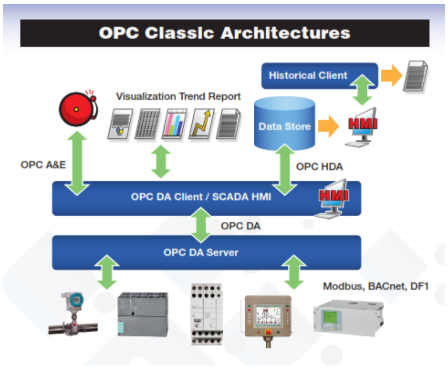


Figure 3.3 OPC structure [*Live webinar*]

tems lead to the development of the OPC Unified Architecture which is a platform-independent standard for data exchange. One key feature during the development phase was to keep all the functionality of classic OPC. This was important so that all devices running OPC would also be compatible with the new standard. More advanced modelling possibilities were desired as well as a higher focus on the security features [Mahnke et al., 2009]. Together the OPC Foundation came up with a set of specifications, a total of thirteen listed in Figure 3.4, describing all the functionality of OPC UA. The goal was to create a new standard that was not going to be dependent on a specific technology nor a specific platform. Devices should be able to run in all operating system with applications written in different languages as shown in Figure 3.5.

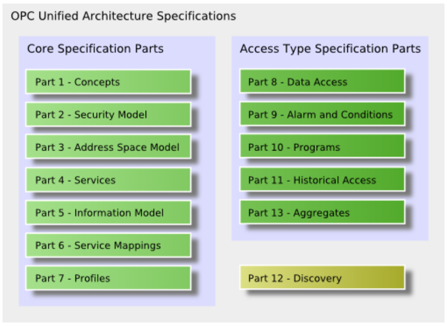

OPC Unified Architecture Specifications
Core Specification Parts
Part 1 - Concepts
Part 2 - Security Model
Part 3 - Address Space Model
Part 4 - Services
Part 5 - Information Model
Part 6 - Service Mappings
Part 7 - Profiles
Access Type Specification Parts
Part 8 - Data Access
Part 9 - Alarm and Conditions
Part 10 - Programs
Part 11 - Historical Access
Part 13 - Aggregates
Part 12 - Discovery

Figure 3.4 OPC UA Specifications [*OPC UA Specifications*]

Classic OPC has shown great success connecting control systems in the local networks but with an expanding market with companies having factories around the world a need for connecting the different sites were developed. The many security features of OPC UA, described in the specifications [*OPC UA Specifications*], makes it possible to connect with other systems over the internet. OPC UA allows for a more integrated system connecting the small embedded devices in the factory with the servers on enterprise level. Tom Burke, the President of the OPC Foundation expresses it [*Live webinar*]:

*"...from the shop floor to the top floor" [*Live webinar*]*

3.6 Grafchart

Grafchart is a graphical programming language that has been developed by the department of Automatic Control at Lund University. It works as a modelling tool based on the theories from Grafcet, statecharts, Petri nets and also some ideas from other object oriented programming languages [Johnsson and Årzén, 1998]. Grafchart is used for sequential procedural application and implements the state-transition paradigm. It was originally developed to be used for batch control but it has proved to work in other automation applications as well [Theorin, 2013]. In the beginning of the development process that started in 1991 there were two implementation of Grafchart where the first one was using the same name Grafchart and was built using Gensym's expert system and the second implementation was done in Java and given the name JGrafchart [Theorin, 2013] The first implementation using the system from Gemsym is no longer in use since it was desirable to use an open platform for the development. JGrafchart is the implementation that is currently used and improved.

JGrafchart

JGrafchart is the Java implementation of Grafchart that is used in this thesis for controlling the process. The two main building blocks in JGrafchart are steps and transitions. A step is representing a state in the process where actions can be executed or services can be called. Each step can either be active or inactive depending on the position of the token. There are four different ways for actions or services in a step to be called. It can happen when the step is activated, which means when the token enters the step, or it can happen when the step is deactivated, which means when the token exits the step. The actions can also be executed on a periodic basis

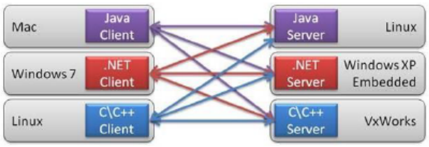


Figure 3.5 OPC UA platform independence [*Live webinar*]

while the step is active. The last option is for the actions to execute when the step is aborted, which happens when an exception transition is fired [*Grafchart*]. It is also possible to measure how long a step has been active using two different methods depending on if the number of seconds or the number of scan cycles the step has been active for is needed. The syntax for finding out the number of seconds a step has been active for is < step name > .s and in a similar way <step name> .t returns the number of scan cycles. A step is illustrated in figure 3.6

The second building block in JGrafchart are the transitions. It represent the change from one state to another and it is associated with a boolean condition. When the step(s) preceding the transition is active the transition is enabled and when the boolean condition becomes true the transition fires. A transition is illustrated in Figure 3.7

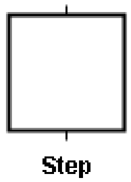


Figure 3.6 Step

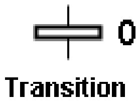


Figure 3.7 Transition

JGrafchart has support for alternative as well as parallel paths shown in Figure 3.8. Parallel paths are created by using the special building block in JGrafchart called Parallel Split which is illustrated in Figure 3.8. Parallel paths are executed in parallel which means that they both contain steps that are active and at the end of the parallel paths the different paths are joined with a building block called Parallel join also illustrated in Figure 3.8. Before the transition following the Parallel join can fire all the preceding steps above the Parallel join must be active and contain a token.

Alternative paths, as shown in Figure 3.8, in JGrafchart means that the token does as the name indicates and chooses one of the paths depending on which transition that fires first. If there are multiple transitions that could fire it is up to the user to make sure that only one fires. This can by either assigning a priority to the transition or making sure that they are mutually excluded.

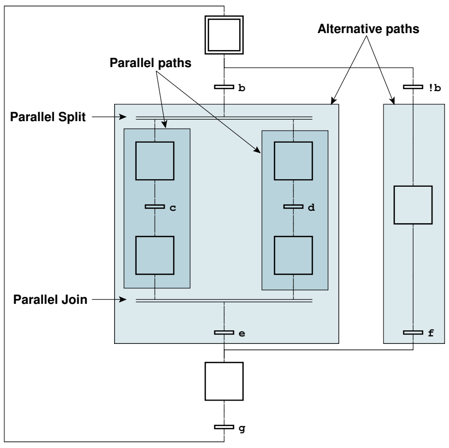


Figure 3.8 Parallel and Alternative paths [*Grafchart*]

In section 3.3 the benefits of composed services are mentioned as a way to increase the level of abstraction. JGrafchart offers a way to implement this modelling where the basic services are orchestrated to form some functionality and then made available as a composed service. This feature is called procedure step, in Figure 3.9, in JGrafchart and allows the user to organize some steps together and make them available as one larger step that model more functionality than the individual steps.

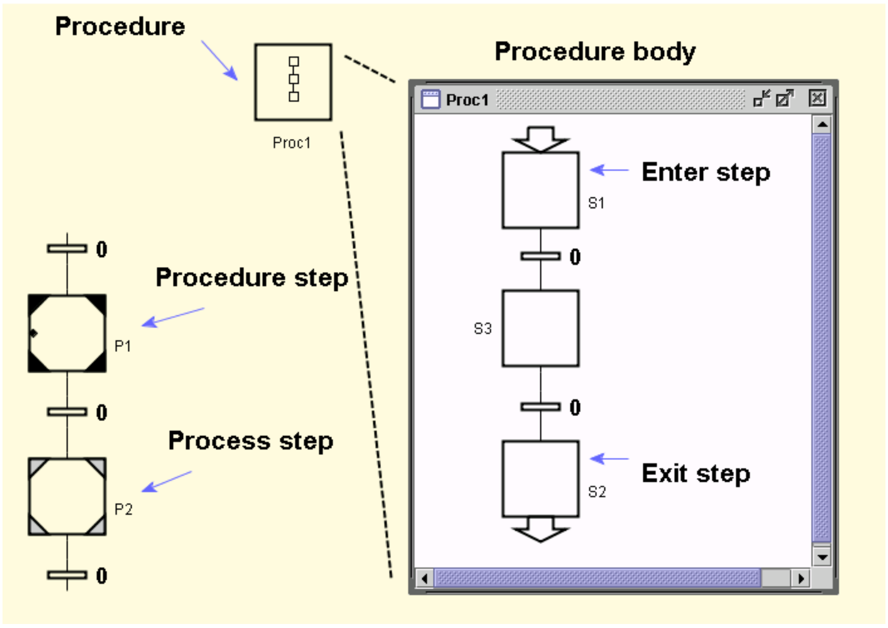


Figure 3.9 Procedure step, a way to orchestrate basic services into combined services [*Grafchart*]

4

Problem description

4.1 Problem description

Manufacturing companies are doing their best trying to match the markets demands for a flexible production process, short set-up times and increased reusability. The tendencies within the manufacturing industry today are shorter product life cycles and increased complexity in the production process. This requires an automation system that is more flexible to fast adapt to the changes. With a more advanced automation system dealing with the flexible production lines there is a growing need for more advanced control systems. A widely spread technology for dealing with these control systems is the use of programmable logical controllers (PLC) together with field buses. The implementation of PLCs requires a lot of work because there are no good way to connect the development of the PLC programs with the other planning of the production processes. The coding must be done all over again for each new project and with more complex projects with an increased degree of process logic the requirements on the PLC programs are also increasing. The programming is done in the very last step of the planning phase since the PLC code is very dependent on the hardware that is being used. This is one of the major reasons why the reusability of the PLC code is very limited. The cost of installing a new plant together with its set-up can sum up to almost one third of the total cost over the plant's lifetime [Jammes et al., 2005]. The production processes are changed more often due to the shorter life cycles of the products which calls for changes in the control systems for the different automation devices more often. Re-engineering the code is a costly process and since each new automation device has a unique control system as much as eighty percent of the engineering effort can be spent on only reimplementing this device. With a higher level of reuseability this effort can be spent elsewhere and much time can be saved [Jammes et al., 2005]. How can the development of control system be better integrated already in the planning phase and is there a way to be able to reuse the already developed system in similar projects?

5

OPC UA

5.1 Address space model

The address space model is the OPC UA equivalence of a meta model. It defines the basic structure and building blocks for OPC UA. OPC UA uses a node concept for expressing the different information. It works as a framework for the modelling on an abstract level defining the different node classes and reference types illustrated in Figure 5.1[Mahnke et al., 2009].

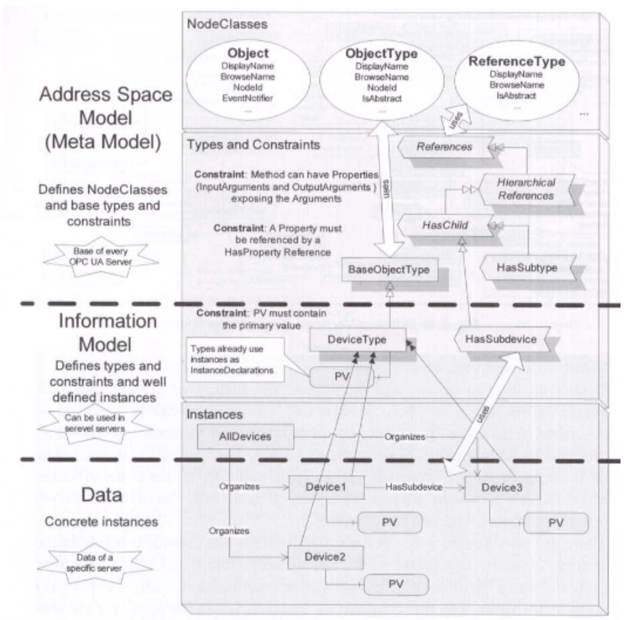

Address Space Model (Meta Model)
Defines NodeClasses and base types and constraints
Base of every OPC UA Server
Information Model
Defines types and constraints and well defined instances
Can be used in serevel servers
Data
Concrete instances
Data of a specific server
NodeClasses
Object
DisplayName
BrowseName
NodeId
EventNotifier
ObjectType
DisplayName
BrowseName
NodeId
IsAbstract
ReferenceType
DisplayName
BrowseName
IsAbstract
uses
Types and Constraints
Constraint: Method can have Properties (InputArguments and OutputArguments) exposing the Arguments
Constraint: A Property must be referenced by a HasProperty Reference
References
Hierarchical References
HasChild
HasSubtype
BaseObjectType
Constraint: PV must contain the primary value
Types already use instances as InstanceDeclarations
DeviceType
PV
HasSubdevice
Instances
AllDevices
Organizes
Device1
HasSubdevice
Device3
Device2
PV

Figure 5.1 Address space model, Information model and data [Mahnke et al., 2009]

5.2 Information modelling

OPC UA is all about exchanging information where the main objective for the server is to make information available for the client to access. The information model uses the concept of the address space model to structure the data. There are different levels of the information model where the top level works as a base for the other. It is the Base OPC UA information model and contain the base node of the different node types shown in Figure 5.2. Different vendors can produce their own information model for their device to make sure that it is always described in the same way. Multiple information models can be used together to describe different part of the information made available to the client. This project focuses on the information model that models the device and the services offered by the field devices. It is derived from the base information model that is already defined in the specifications [Mahnke et al., 2009] [*OPC UA Specifications*].

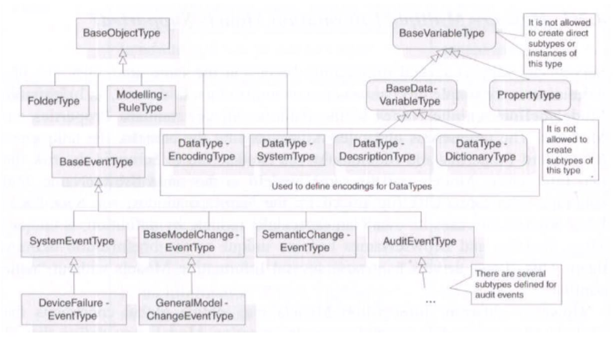


Figure 5.2 Base TypeDefinitions [Mahnke et al., 2009]

5.3 Building blocks

OPC UA uses a concept of nodes and references to do all the information modelling. There are eight node classes listed in Table 5.1 that are defined by the OPC UA specifications [*OPC UA Specifications*]. Nodes are used to represent different entities that the server is making available and references are describing the relationship between two nodes. The number of node classes are predetermined and new classes cannot be created by the person designing the server [Lange et al., 2010]. Each node has a specific number of different attributes that describes the node. There are some attributes that are the same for every node and some attributes that are depending on the node class. The structure of a node is shown in Figure 5.3.

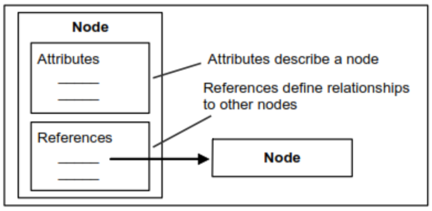

Node
Attributes
References
Attributes describe a node
References define relationships to other nodes
Node

Figure 5.3 Model of a node [*OPC UA Specifications*]

5.4 ObjectTypes vs Objects

In classic OPC the only types that were possible to describe were the standard data types such as integers and strings. OPC UA offers the possibility to create types at an object level. In this way it is possible to describe not only the data itself but also the information behind it such as from what kind of device the data is coming. Creating types describing each device allows for a simple implementation of creating an instance of this object type when an object like this needs to be modelled. OPC UA allows for the vendors to create types describing their products and these types can be used by the developer when modelling an object of this type.

Creating types is only available for objects and variables. Methods on the other hand are always bound to an object or an object type therefore they have no type definition. It is not mandatory in OPC UA to use type definitions and instead do as in classic OPC where type definitions where not available. This comes from the fact that OPC UA still has all the features supported by classic OPC.

Example of type programming on device level in OPC UA

In Figure 5.4 a type description of a motor is illustrated. It does not represent a physical motor, it works as a blueprint of the functionality offered by this kind of motor. It can be seen that the motor have some kind of set-up and configurations and then it has the methods to start and stop. This kind of type models can be supplied by the manufacturer to allow the developer to only create an instance of the type when a motor like this need to be modelled. On the right side in Figure 5.4 an instance of the motor type can be seen. It represent a physical motor and has the same structure as the type definition describes [Mahnke et al., 2009].

The implementation in Figure 5.4 has its limits since the location of the node is already determined and limits the use of this motor type to only model motors that match this type. This limitation can be avoided by using a more generic design on device level and model all functionality as services as in DPWS.

Node Class	**Description**
Variable	A variable is used for representing a value on the server and it always belong to an object.
Object	Objects represent real objects and can stretch from a small sensor to a large plant. Objects have variables and methods and can fire events
Method	A method is a function on the server that the client can call. It has properties describing its inputs and outputs
View	A view is a way to limit the access to the address space for different users. A view can represent parts of the address space that the designer wants the user to be able to access. This functionality is not yet implemented in the toolboxes used for this project.
DataType	This type represent the type that a value of a variable can have. There are a large number of defined data types in the specifications.
VariableType	It describes the meaning of the variable, not the value itself. It can also expose the structure below the node.
ObjectType	It works as a blueprint for an object describing the underlying node structure. It is used when creating instances of an object.
ReferenceType	There are several different kinds of reference types that are described in the OPC UA specifications.

Table 5.1 Node classes

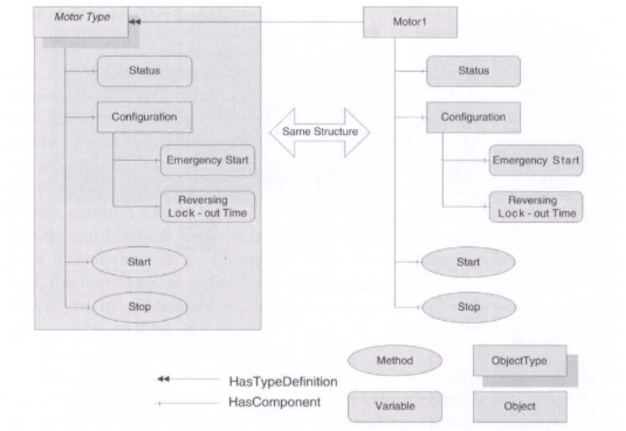


Figure 5.4 Type definition of a motor and an instance [Mahnke et al., 2009]

6

Tools

6.1 Introduction

This chapter will present the different tools used during this project. First a tool for modelling the information model has been used where the modelling of services has been investigated. Two toolboxes have also been used for creating the server and client for the implementation. A short introduction to each one of them is given in this section.

- Unified Automation UaModeler
- Prosys OPC UA JAVA SDK Client
- Softing OPC UA C++ Server Development Toolkit for Linux

6.2 Unified Automation UaModeler

UaModeler is a tool developed by Unified Automation that offers the possibility to create a graphical representation of the address space. It enables the user to add nodes and relationships to best describe the available information on the server. Figure 6.1 shows how that structure is formed in UaModeler. The intention is not to explain the full power of UaModeler but to give the reader a short explanation to what functions are available and the possible benefits of using this kind of program before starting to develop a project and implementing OPC UA. The two main parts are the two folders Objects and Types. In the type folder the different types are listed and based from these types instances are created and allocated as objects on the server under the object folder in Figure 6.1. In the Object folder the physical devices are modelled by creating an instance of the desired type(s).

After the types are created and the physical device is described using these types the structure can be graphically visualized in UaModeler using the Graphic View, a built in function in UaModeler. It shows how the different objects are connected and describes the relationship between two components.

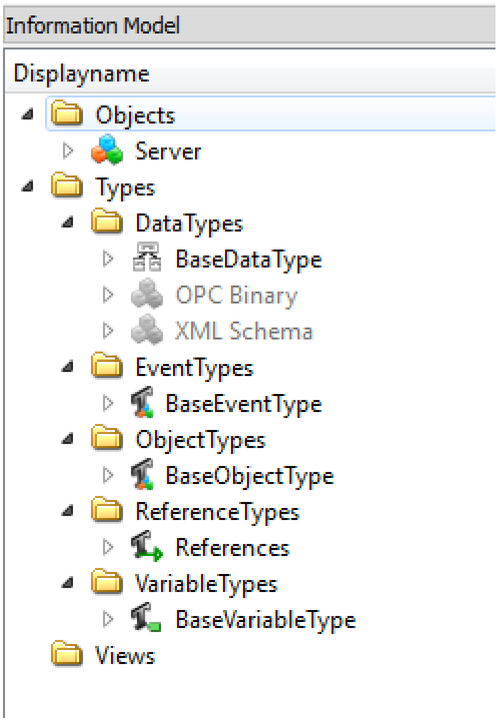


Figure 6.1 UaModeler

6.3 Prosys OPC UA Java SDK Client

This is a toolbox developed by Prosys which is a company based in Espoo, Finland, that is developing OPC software. This Java SDK contains all the OPC UA specifications that are developed by the OPC Foundation. Prosys is the leading company on developing a Java SDK for OPC UA. JGrafchart is a program written in Java and therefore the Java SDK was chosen for the client side in this project. When using their toolbox for developing the OPC UA Client they supply a tutorial with the basic functionality that is needed for a simple set-up.

6.4 Softing OPC UA C++ Server Development Toolkit for Linux

This is a toolbox that is developed by Softing, a software company with their headquarters in Haar, Germany. Softing is a partner of the Smart Factory and therefore their toolbox is chosen for the development of the server running C++ in this project.

Like Prosys, Softing also offers a tutorial on how the basic settings are handled in their toolbox.

7

Service modelling in OPC UA

7.1 Introduction

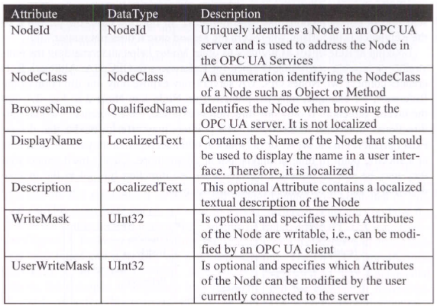
This chapter describes how services can be modelled in OPC UA, describing the benefits of using generic devices that can host services as well as how it has been done using the tool UaModeler.

7.2 Generic OPC UA device

Developing different types of objects is a powerful tool in OPC UA that can reduce the engineering effort when implementing a production process. There are multiple benefits from using type definitions of devices and creating instances of these when a device needs to be modelled instead of creating a new object from scratch every time. Using predefined types of devices such as the motor type in Figure 5.4 limits the possible structure of the motor. With this approach, using the type in Figure 5.4, it is necessary to always organize the methods start and stop directly under the motor type object and the configuration settings must be in a specific object. There is no possibility to modify this structure. What if there could be a generic device that could represent all kind of devices with any possible structure. Then the device could in one implementation represent a simple motor or a simple sensor but in another implementation represent a smart device of a motor with built in temperature sensors. The goal is to have one generic device that has no predefined structure as the example with the motor type in Figure 5.4 has. In that example the motor has some structure already with methods and objects, the goal in this case will be to have a blank device that can represent any possible device modelled in the automation industry.

In OPC UA there are eight node classes as described in Table 5.1 that form the available building blocks. Each node has a certain number of predefined attributes where the standard ones that all node classes have are shown in Figure 7.1 [Mahnke et al., 2009]. The first step is to create each device as an object containing no other objects, variables or methods. The only thing that should be included in a generic device type should be properties that every possible device in a plant could have. The goal for this device model is to be as generic as possible to be able to fit all possible needs

There are two features for a device that are of great importance. Each device must have some form of unique id number for identification since the client must be able to verify that the device it is connected to is the same as last time. It could be disastrous if a client connecting to a device is connected to a similar device but not the exact same, located somewhere else in the factory. The second property that each device needs is some build information such as version number and manufacturer.



Attribute	DataType	Description
NodeId	NodeId	Uniquely identifies a Node in an OPC UA server and is used to address the Node in the OPC UA Services
NodeClass	NodeClass	An enumeration identifying the NodeClass of a Node such as Object or Method
BrowseName	QualifiedName	Identifies the Node when browsing the OPC UA server. It is not localized
DisplayName	LocalizedText	Contains the Name of the Node that should be used to display the name in a user interface. Therefore, it is localized
Description	LocalizedText	This optional Attribute contains a localized textual description of the Node
WriteMask	UInt32	Is optional and specifies which Attributes of the Node are writable, i.e., can be modified by an OPC UA client
UserWriteMask	UInt32	Is optional and specifies which Attributes of the Node can be modified by the user currently connected to the server

Figure 7.1 Attributes of a node

7.3 Services modelled in OPC UA

In SOA-AT all functionality of the field devices are encapsulated and made available to the client in form of services. The device will be assigned a number of hosted services depending on what kind of field device it represents. For DPWS this approach is illustrated in Figure 3.2 with the device at the top with its hosted services organized below. There are no developed standard for how services are modelled in OPC UA and very little research is done in this area. The book "OPC Unified Architecture" written by Wolfgang Mahnke et al. [Mahnke et al., 2009] can according to Tom Burke, president of the OPC Foundation, be used as a handbook when developing OPC UA products. How types are to be used to describe different devices are well covered as well as the possible benefits of it but how services can be modelled using a generic device is still new ground.

Actions are executed in OPC UA by calling methods. Methods are described in section 7.3 and are always belonging to an object as a component. Each method has information about what parameters that are needed as input and output [Mahnke et al., 2009]. There is no standardized way to express a method in OPC UA or explain how they should be designed, only the signature of it. A method is something the client can call and in return receive a result as reply. It works similar to an operation in DPWS where the operation is called with a certain input returning a calculated result [Theorin, 2013].

Methods can be thought of as operations from DPWS but how the structure between operation and service should be is not yet defined for OPC UA, illustrated in Figure 7.2, as it is for DPWS. DPWS showed some promising results [Bohn et al., 2006] for implementing SOA-AT on embedded devices so a first approach is to try modelling services in OPC UA would be to use the same structure as defined for DPWS in Figure 3.2. The devices are modelled as generic objects with no predefined functionality beside the ID number and device information. Services are created as objects where methods can be added as components. The space shown in Figure 7.2 between the service level and the level of the methods are yet to be structured and for this the first approach is to follow how it is done in DPWS using one layer to organize the methods in folders that are located under the services. The folder object will be a component of the service object and it will have the different methods as components. This is the structure is shown in Figure 7.3 where a motor is modelled. At the top is the generic device that in this case represents the motor and as components of the motor one service, *motorService*, is located. The service can then have two folders organizing the different methods from the bottom layer. The added layer of folders offers a more organized view instead of having all the methods directly below the service object.

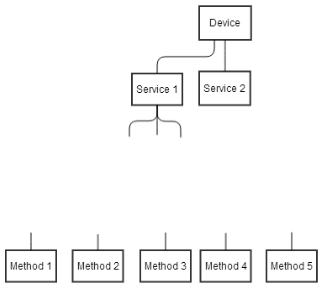


Figure 7.2 OPC UA service structure

OPC UA is not limited to this approach for modelling services as illustrated in Figure 7.3. It is free for the designer to add as many or as few layers of folders between the services and the methods as he or she feels is appropriate. The node notation in OPC UA allows for more options when modelling services than what

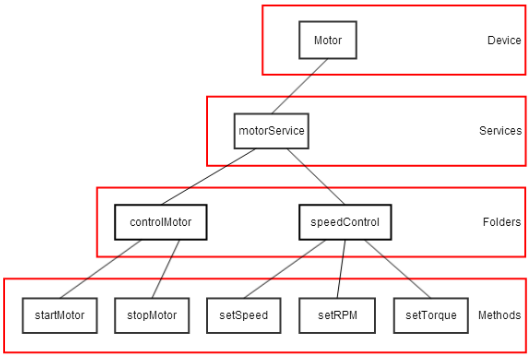


Figure 7.3 OPC UA services modelled according to DPWS structure

is possible in DPWS where the structure from Figure 3.2 is predefined. Methods can also be placed at different levels in OPC UA, they do not need be at the same level as in DPWS. Figure 7.4 illustrates how methods can be organized at different levels. This allows the designer to structure some methods that is not used so often in a folder. The methods that are more important can be located directly under the service object for easier access.

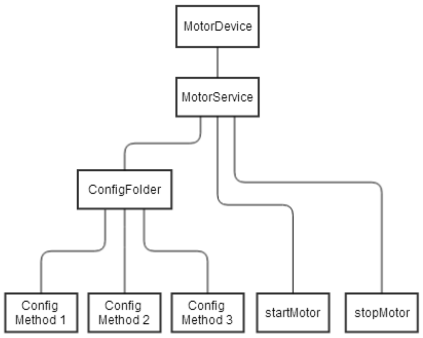


Figure 7.4 OPC UA services with a free structure for

OPC UA Method

In Figure 7.5 it is illustrated how it works when a client makes a service call to a method. The call may contain input parameters that are specific for this method and the method may return some specific output parameters. A method is a component of an object and can be discovered by the client through browsing the object the method belongs to [*OPC UA Specifications*] The different inputs and outputs for a methods are defined as properties of the method [*OPC UA Specifications*]. There are special browsing functions built in to the toolbox for retrieving the methods that belong to a node as well as commands for finding out the in- and output arguments and how it works is described in Listing 8.18 [Prosys, 2012].

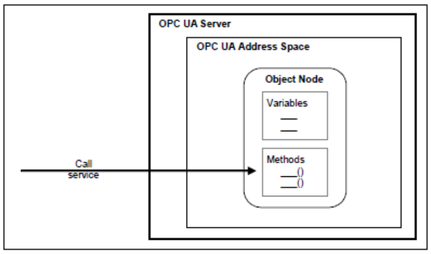


Figure 7.5 Method Service set [*OPC UA Specifications*]

7.4 Service library

UaModeler described in section 6.2 offers the possibility to create a library like functionality of the services in the information model as shown in Figure 5.1. Here the different services are listed as subtypes of the type service. These different services can be allocated on different devices to get the desired functionality.

Development of a control procedure

The services that are developed and described in the library in Figure 7.6 are used in the process of creating control system where the first step is to create an abstract process description. The process description is describing what is done in the process, not specifically how or which machine is doing what. For the process to be executable a process logic needs to be developed based on the process description which means that process needs to be broken down into steps describing what is done where. The services must then be orchestrated as illustrated in Figure 7.7 [Theorin et al., 2013] to match the process logic. The service orchestration is done in three steps where the first step is the abstract process description describing what is done step by step. The next stage is to determine how the different steps

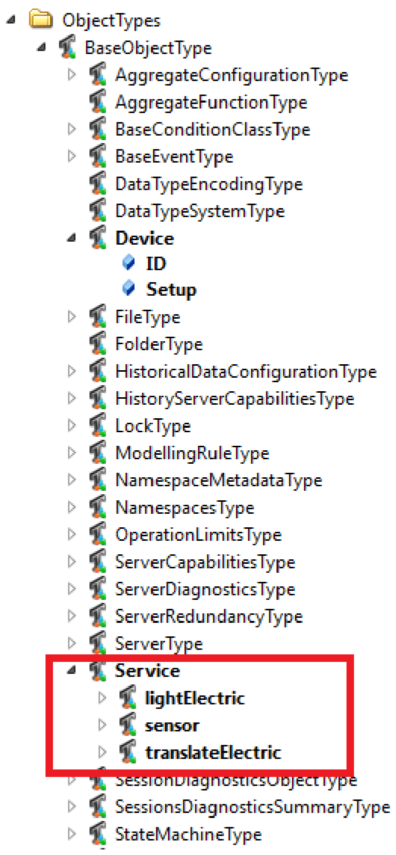


Figure 7.6 Service library

in the process description is realized using abstract functions. This step is sometimes called functional realization since the functionality needed for doing the steps in the process description is described here. The functions are abstract and not connected to a specific hardware. In the third and final step the abstract functions from the functional realization step are converted to real services that are connected to the physical field devices [Ollinger and Zühlke, 2013]. It is first in the last step that the services described in the library are allocated on the device

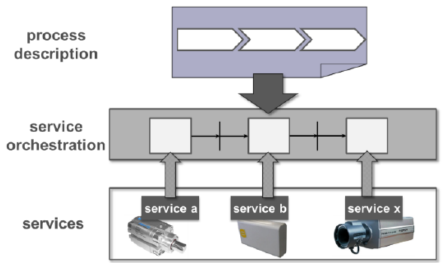


Figure 7.7 Process description

8

Client and Server implementation together with JGrafchart

8.1 Introduction

In the second part of the project the goal was to create a set-up with an OPC UA server and client and this will be explained in this chapter. On the server a simple device from Ficshertecknik is to be modelled and its functionality made available to the client. How can the client connect to the server and establish a subscription of values representing the different sensors and the motor? The device will be controlled from a JGrafchart application where the monitored values from the client will be used to fire the transitions. This part will also explain some of the basics when setting up a project with an OPC UA client and server.

8.2 Fischertechnik device

The device used for this application is a simple device from Fischertecknik that consist of a conveyor belt with one sensor on each side indicating when an object is present. There is a motor driving the conveyor belt left and right. There are also some extra lights that can be turned on and off but they are not used. The device can be seen in Figure 8.1.

8.3 Server

The server for this project is developed using the toolbox supplied by Softing, a partner of Smart Factory. The server is written C++ since it is supposed to run on an embedded device connected with the equipment form Fischertechnik. In the beginning of the process of creating an OPC UA server there are some steps that are

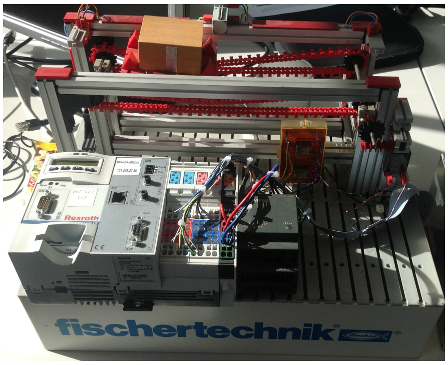


Figure 8.1 Fischertechnik device used in Kaiserslautern

necessary to do dealing with the set-up and security features as described in Listing 8.1

Listing 8.1 Initial set-up for creating a server

```
// Create an application instance
ApplicationPtr pApp = Application::instance();

// Create a description of the application
ApplicationDescription appDesc;

// defining that it is a server application
appDesc.setApplicationType(EnumApplicationType_Server);

// some settings
appDesc.setApplicationName(LocalizedText(_T("OPC UA TestSErver"),
    _T("en")));
appDesc.setApplicationUri(_T("urn:") + hostName +
    _T("/Softing/OpcUa/TestServer"));
appDesc.setProductUri(_T("urn:Softing/Products/OpcUa/SampleServer2"));

// Initializes the application with the created description
pApp->initialize(&appDesc);

// Starts the application
pApp->start();
```

Once the initial set-up for the server is done one or more endpoints needs to be open for allowing clients to connect to the server as described in Listing 8.2.

Listing 8.2 Setting up endpoint for the client to access the server

```
Server::EndpointPtr endpoint = Endpoint::create();
endpoint->setUrl(<url>);

// Setting the security level of the server
endpoint->addSecurityConfiguration(EnumMessageSecurityMode_None,
    SecurityPolicyUri_None);

// Setting user settings
UserTokenPolicy userTokenPolicy;
userTokenPolicy.setPolicyId(_T("Anonymous_Policy"));
userTokenPolicy.setTokenType(EnumUserTokenType_Anonymous);
endpoint->addUserTokenPolicy(&userTokenPolicy);

// Adding endpoint to application
Application::instance()->addEndpoint(endpoint);

// Opening the endpoint to make the server available to clients
endpoint->open();
```

The server is now open for a client to connect to and at this stage the security is set to none. The choice to use no security at start is done to focus on first implementing a working set-up where values can be monitored and later when actual hardware is used the desired security layers can be added. The server is at the moment blank, there is no information modelled in the address space. No nodes have been added at this stage. There are three ways to insert nodes to the address space as shown in Listing 8.3. The first option for inserting a node is the *init(<typedefinition>)* function that initialize a node in the address space of the type described by the type definition. The second option for inserting nodes in the address space is by using the function *insert()*. The difference between the two is that for the second one the user is responsible for adding components and references which is done automatically in the first case when using the type definition. The last option, *insertTree()*, is similar to the insert function but it allows for the user to add a structure of nodes to the address space.

Listing 8.3 Three ways to insert a node to the address space

```
// Three ways for inserting nodes in the address space.
Server::BaseNode::init( <typedefinition> );

Server::BaseNode::insert();

Server::BaseNode::insertTree();
```

8.4 Modelling a sensor on the server

There are two sensors modelled on the server and they are implemented as described in Listing 8.4. Each sensor is a simple boolean variable that indicates whether an object is present or not.

Listing 8.4 How to create a sensor variable on the server

```
Server::VariablePtr var1;
var1 = Server::Variable::create();
var1->setNodeId(NodeId(namespaceIndex,ADDRESS_SENSOR_ONE_OBJECT_PRESENT));
var1->setDisplayName(LocalizedText(_T("Sensor_1"),_T("en")));
var1->setDescription(LocalizedText(_T("this is sensor 1"),_T("en")));
var1->setDataType(Statics::DataTypeId_Boolean);
var1->init(Statics::VariableTypeId_BaseVariableType);
var1->setBrowseName(QualifiedName(_T("Sensor_1"),typeNamespaceIndex));
var1->setAccessLevel(EnumAccessLevel_CurrentRead);

// Here the sensor variable is added as a component of the device.
parentDevice->addReference(Statics::ReferenceTypeId_HasComponent,var1);
```

Running server on embedded device

The goal for the server is to run on an embedded device connected to a physical unit from Fischertechnik but due to limited availability of area boards the functionality of the device is simulated using the *l* and *r* key on the keyboard for setting the values of each sensors, *l* is for changing the value of the left sensor and *r* is for the right one. A complete set-up of equipment is ordered but due to long delivery times it will be included as future work.

8.5 Client

The client is the part accessing the data made available by the server and in this project the client is developed in Java using the toolbox from Prosys as described in section 6.3.

Setup

When creating and setting up a client from scratch in OPC UA there are some steps that are always needed to be done. They will be described briefly in the following section and for a more detailed description the reader is instructed to study the toolbox specific tutorial that most developers supply with their SDK.

Since Java is the chosen platform for development of the client this section focuses more on how it is done in Java. Every OPC UA application must contain some information about itself that works as a description of it for other devices, Listing 8.5 describes how it is done. OPC UA has a high focus on the security and therefore the security settings must be defined, which level of security is desired. The client needs to know the URI of the server that it will try to connect where the URI is of the format: <Protocol>://<Host>:<Port>:<ServerName> [*Grafchart*]

Listing 8.5 Settings needed for the ApplicationDescription

```
ApplicationDescription appDesc = new ApplicationDescription();
appDesc.setApplicationName(newLocalizedText(APP_NAME,Locale.ENGLISH));
appDesc.setApplicationType(ApplicationType.Client);
appDesc.setApplicationUri("urn:192.168.0.1:4880:Client");
appDesc.setProductUri("urn:smartFactory:Client");
```

The toolbox contains a client UaClient class that covers most of the functionality for connection to the server as well as the actual OPC UA communications. The result of this is a simple interface for the user when constructing an application. Only a few lines of code is needed as seen in Listing 8.6

Listing 8.6 How a client object is created

```
try{
       UaClient client = new UaClient(serverUri);
}catch (URISyntaxException e){
       e.printStackTrace();
}
```

The large focus on high security in OPC UA is easily implemented in the client requiring only one line of code as seen in Listing 8.7.

Listing 8.7 Available security modes in OPC UA

```
client.setSecurityMode(SecurityMode.NONE);
//client.setSecurityMode(SecurityMode.BASIC128RSA15_SIGN);
//client.setSecurityMode(SecurityMode.BASIC128RSA15_SIGN_ENCRYPT);
//client.setSecurityMode(SecurityMode.BASIC256_SIGN_ENCRYPT);
```

Once the set-up for how and where to connect is done it is only a simple command for the client to connect to the server shown in Listing 8.8. Once this is done it is possible for the client to start browsing the server to find out what kind of data that is available.

Listing 8.8 How to connect to the server

```
client.connect();
```

It is just as simple to disconnect from the server as it is to connect. There is only one command for that as well as shown in Listing 8.9.

Listing 8.9 How to disconnect from the server

```
client.disconnect();
```

Browsing

In OPC UA all the data and services are modelled on the server and made available for the client to access once connected. The structure of the address space in form of nodes and references allows the client to browse from one node to the next in order to find the desired services. The browsing always starts in the root folder and how the nodeId for this is retrieved is shown in Listing 8.10

Listing 8.10 Identifying nodeId of Root folder

```
NodeId nodeId = Identifiers.Rootfolder;
```

The browsing of the address space is done by calling the function browse on the address space and using the nodeId of the current position as input as shown in Listing 8.11. This function returns a list of reference descriptions for all the references leaving the current node.

Listing 8.11 How to browse the address space

```
List<ReferenceDescription> references =
    client.getAddressSpace().browse(nodeId);
```

The nodeId of the target node can be found by following the reference, shown in Listing 8.12 in a similar way as the browsing was done.

Listing 8.12 How to follow a reference

```
//action is an integer indicating the number of the chosen reference
try {
        ReferenceDescription r = references.get(action);
        NodeId target;
        try {
                target = browse(client.getAddressSpace()
                .getNamespaceTable().toNodeId(r.getNodeId()));
```

Read & Write values

The client can write a value to the server and all that is needed for this is the nodeId and the attributeId and is done as in Listing 8.13 where nodeId is the node that a value is written to, the attribute represents the attribute of this node that is to be changed and value is the new value that is written to the node.

Listing 8.13 Write value on the server

```
boolean status = client.writeAttribute(nodeId,attributeId,value)
```

There is a built in system of status codes in OPC UA where each error has a unique status code that is returned when a call is made. There are many different status codes that could indicate if the operation was done immediately or asynchronously as well as status codes that indicates if something went wrong and the reason for it. A complete list of all the error messages can be found in the specifications [*OPC UA Specifications*]. The client can read a value on the server in two ways where the first one is rather simple and not so efficient. The function in Listing 8.14 only reads on attribute from one node.

Listing 8.14 Simple read function for only one attribute

```
DataValue value = client.readAttribute(nodeId,attributeId);
```

It should be avoided to call the read function for an individual item. The other read function can do multiple reads in one call, which means it can read many attributes at once as shown in Listing 8.15.

Listing 8.15 Read for multiple attributes

```
client.read()
```

For monitoring values on the server, for instance to monitor the value of a sensor, there are better ways than to read a value over an over again to see if it has changed. Creating a subscription of a data value is the preferred way when clients needs to monitor value changes of a variable.

Subscription

In OPC UA there are three different types of monitored items that a client can create a subscription of. The most common one to subscribe to is the monitoring of a certain value on the server, once again it could be the simple example with the value of a sensor. Subscribing to the change of a variable value is considered as one of the most important features in OPC UA. The client can also subscribe to generated events from the server. The last option for the client is to subscribe to aggregated values in a client specified time interval [Mahnke et al., 2009]. How a subscription for the first kind is created is shown in Listing 8.16. What is executed once the value that is subscribed to is changed is shown in Listing 8.17.

Listing 8.16 Creating a subscription

```
//create subscription
if(subscription == null){
        subscription = createSubscription();
```

```
       client.addSubscription(subscription);
}
// pause the monitoring while editing the subscription
subscription.setPublishingEnabled(false);

UnsignedInteger attributeId = readAttributeId();
MonitoredDataItem item = new MonitoredDataItem(nodeId, attributeId,
    MonitoringMode.Reporting);

subscription.addItem(item);
//subscription is started again after that the new item is added
subscription.setPublishingEnabled(true);
subscription.setPublishingInterval(100);

if(!item.hasChangeListener(dataChangeListener)){
       item.addChangeListener(dataChangeListener);
}

// After the subscriptions has been edited they are updated
subscription.updateItems();
```

Listing 8.17 When a change occurs on a monitored value

```
@Override
public void onDataChange(MonitoredDataItem sender, DataValue
    prevValue, DataValue value){

       // operation done when the monitored value has changed.
}
```

Listing 8.18 Description of different method commands

```
//How to find the methods offered by an object

List<ReferenceDescription> methodRefs =
    client.getAddressSpace().browseMethods(nodeId):

// Input Output arguments for a method:

Argument[] inputArg = method.getInputArguments();
Argument[] outputArg = method.getOutputArguments();

// How to call a method:

Variant[] outputs = client.call(nodeId, methodId, inputs);
```

Device modelled on the server

The device modelled on the server is a very simple application with a conveyor belt with one sensor at each side detecting when a object is present. Once an object is detected the motor driving the conveyor belt is stopped and the system makes a short break and then the motor starts in the opposite direction. The pause, shown in Figure 8.2 by the *StopMotorRight* and *StopMotorLeft*, represent the time when work can be done on the product. The sensors are modelled as a boolean variable where true indicates an object present. This value is set to be only readable since the client shall not be able to set this value. The motor is modelled as an integer where negative values mean one direction, zero represents not moving and positive values represent the other direction. This value is both readable and writeable since the client must be able to start and stop the motor depending on the position of the object on the conveyor belt.

Figure 8.2 illustrates the control application software for the conveyor belt. The two socket inputs named *SensorLeft* and *SensorRight* are monitoring the values on the server indicating if there is an object present or not. This is established by creating a subscription of this value as described in section 8.5. When the value on the server changes from true to false and vice versa the corresponding transition *SensorLeft* or *SensorRight* fires. The four different steps, not counting the initial step at the top, have each one operation to control the value on the motor.

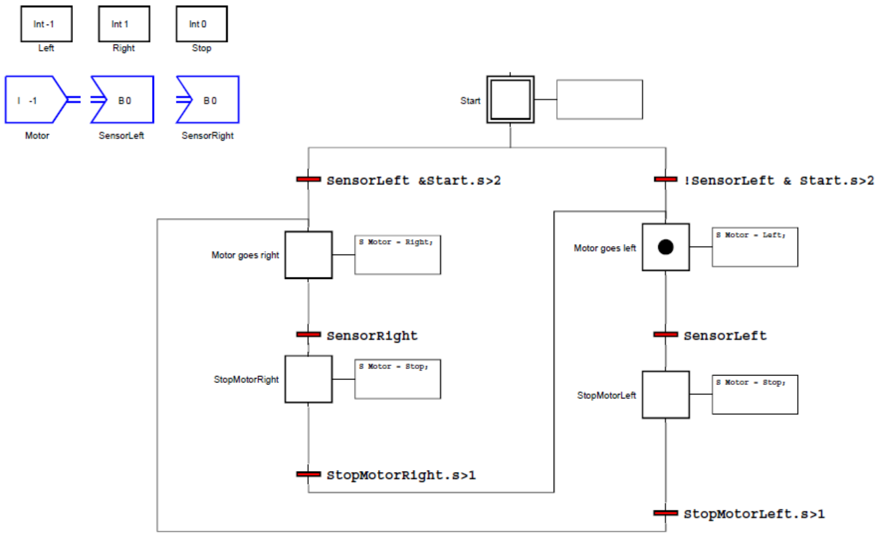


Figure 8.2 Conveyor belt control application in JGrafchart

JGrafchart as OPC UA client

JGrafchart has today an integrated generic DPWS implementation that allows the user to directly browse for available devices and services [Theorin, 2013]. In this project JGrafchart is connected to the OPC UA client using socket communication, blue boxes in Figure 8.2, and then the OPC UA client browses the server and creates the subscriptions of the desired values. These values are then sent back to JGrafchart again using the same socket. There are two socket inputs, one for each sensor, and one socket output for the motor value. The complete connection set up is shown in Figure 8.3 showing the server on the left side and how the client is connected with JGrafchart using standard socket communication on the right side.

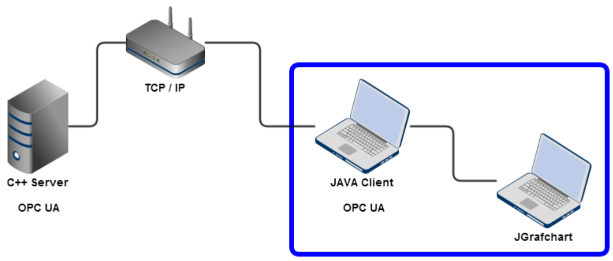


Figure 8.3 Connection set up of the project

9

Results

9.1 Result from service modelling

The results on how services can be modelled in OPC UA are illustrated in Figure 7.3 and 7.4. The structure chosen for the objects between services and methods can freely be organized according to the present demands. At the top is a generic device, just like for DPWS, but the structure of the hosted services are not predefined. Each service can have its own structure of objects with methods located at different levels shown in Figure 7.4.

A service library has been created in UaModeler as illustrated in Figure 7.6 as well as the type definition for the generic device. This type definition has only the basic set up information and an unique ID number as components listed in Figure 7.6. Instances of this type are created when modelling any kind of field device in a factory. Once a generic device is instantiated the services needed for describing the functionality of the corresponding field device are allocated as hosted services on the device. This is accomplished by creating an instance of a service in the library and locate it as a component of the device object.

9.2 Result from implementation

Using the OPC UA server together with the client to control the simulated version of the conveyor belt worked well. The subscriptions of the sensor values are manually created by browsing the address space of the server until the desired variable is located and chosen for a subscription. Once the subscriptions of the values were created the control application in JGrafchart shown in Figure 8.2 is started and the device is controlled. The subscriptions of sensor values on the server works well and each change is detected and the corresponding transition fires.

10

Discussion

The goal was to investigate how services can be modelled in OPC UA and to create a library of different services. The tool UaModeler has proven to be very useful for this. It offers an environment where the information model can be modelled in a graphical way as in Figure 6.1. The created library of services would enable the developer to reuse their work in many applications.

The generic device type together with the service library offers a way for the engineer

The desire to use a generic device for describing every possible field device in a factory can be accomplished in a similar way as for DPWS. UaModeler described offers a powerful way to model services and also shows a graphical representation of the information model. The benefit of starting with first modelling the services in UaModeler is that it offers a better overview and the services can be listed like a library shown in Figure 7.6. How the structure of the services shall be organized must be tested on real application to experience the pros and cons.

JGrafchart uses a graphical syntax shown in Figure 3.6 and 3.7 that is common from other process control languages. This makes it a good candidate to be used for service orchestration to create control software. The procedure step shown in Figure 3.9 offers the possibility of creating composed services in a simple way which allows JGrafchart to easily model more abstract control software.

The implementation of the OPC UA server and client for controlling the device from Fischertechnik using JGrafchart works well. The initial plan was to integrate the client into JGrafchart using a similar block as for DPWS. This integration is in progress and will be included as future work. Integrated support of OPC UA in JGrafchart could offer a good way into the market for JGrafchart as a tool for service orchestration as well as a control software.

There is not only one way to do things in OPC UA, it offers many choices for the designer and much of the focus was spent on understanding how services can be modelled before trying to implement one. The implementation has only been done using a simulated device since the device had not arrived at the end of this project. The results from the simulations indicates that subscribing to values representing the sensors works well.

11

Future work

The project has resulted in a working set-up with a simulated device modelled on the OPC UA server and controlled by the client. So far the client and the JGrafchart application have been two separate applications connected using socket communication. The next step would be to integrate the OPC UA functionality into JGrafchart in a similar way as DPWS has been integrated with a special building block available under the I/O menu.

The second thing that is planned as future work is implementing the use of methods in OPC UA and not only read and write a value on the server.

A physical device from Fischertechnik is ordered to be used for the implementation but due to long delivery time it is included as future work.

Bibliography

Automation World. *Live webinar*. URL: `www.automationworld.com/elster-and-beckhoff-automation-connect-shop-floor-automation-directly-top-floor-sap-me-opc-ua` (visited on 01/21/2014).

Bohn, H., A. Bobek, and F. Golatowski (2006). "Sirena - service infrastructure for real-time embedded networked devices: a service oriented framework for different domains". In: *Networking, International Conference on Systems and International Conference on Mobile Communications and Learning Technologies, 2006. ICN/ICONS/MCL 2006. International Conference on*, pp. 43–43. DOI: `10.1109/ICNICONSMCL.2006.196`.

Department of Automatic Control, Lund University. *Grafchart*. URL: `http://control.lth.se/Research/tools/grafchart.html` (visited on 01/21/2014).

Jammes, F., A. Mensch, and H. Smit (2005). "Service-oriented device communications using the devices profile for web services". In: *Proceedings of the 3rd International Workshop on Middleware for Pervasive and Ad-hoc Computing*. MPAC '05. ACM, Grenoble, France, pp. 1–8. ISBN: 1-59593-268-2. DOI: `10.1145/1101480.1101496`. URL: `http://doi.acm.org/10.1145/1101480.1101496`.

Johnsson, C. and K.-E. Årzén (1998). "Grafchart for recipe-based batch control". *Computers and Chemical Engineering* **22**:12, pp. 1811–1828.

Lange, J., F. Iwanitz, and T. J. Burke (2010). *OPC from Data Access to Unified Architecture*. VDE Verlag.

Mahnke, W., S.-H. Leitner, and M. Damm (2009). *OPC Unified Architecture*. Springer.

Ollinger, L. and D. Zühlke (2013). “An integrated engineering concept for the model-based development of service-oriented control procedures”. In: *Proceedings of the IFAC Conference on Manufacturing Modelling, Management and Control(MIM-2013), June 19-21, Saint Petersburg, Russian Federation*. Vol. 7. IFAC PapersOnLine, pp. 1441–1446. URL: `http://www.ifac-papersonline.net/Detailed/60299.html`.

Ollinger, L., J. Schlick, and S. Hodek (2011). “Leveraging the agility of manufacturing chains by combining process-oriented production planning and service-oriented manufacturing”. In: *Proceedings of the 18th IFAC World Congress. World Congress of the International Federation of Automatic Control (IFAC-2011), August 28 - September 2, Milan, Italy*. Elsevier Science Ltd.

OPC Foundation. *Opc ua specifications*.

Prosys (2012). *Opc ua java sdk client tutorial*.

SmartFactory^{KL}. *SmartFactory*^{KL}. URL: `http://smartfactory.dfki.uni-kl.de/en` (visited on 01/21/2014).

Theorin, A. (2013). *Adapting Grafchart for Industrial Automation*. Licentiate Thesis ISRN LUTFD2/TFRT--3260--SE. Department of Automatic Control, Lund University, Sweden.

Theorin, A. and C. Johnsson (2012). “Graphical programming language support for service oriented architecture in automation”. eng. In: Uppsala, Sweden.

Theorin, A., L. Ollinger, and C. Johnsson (2013). “Service-oriented process control with Grafchart and the devices profile for web services”. In: *Service Orientation in Holonic and Multi-agent Manufacturing and Robotics*. Ed. by T. Borangiu, A. Thomas, and D. Trentesaux. Accepted for publication. Springer.

Wikipedia (2014). *Service-oriented architecture — wikipedia, the free encyclopedia*. [Online; accessed 11-January-2014]. URL: `http://en.wikipedia.org/w/index.php?title=Service-oriented_architecture&oldid=590048072`.

Lund University **Department of Automatic Control** **Box 118** **SE-221 00 Lund Sweden**	*Document name* MASTER´S THESIS *Date of issue* August 2014 *Document Number* ISRN LUTFD2/TFRT--5953--SE
Author(s) Johan Hagsund	*Supervisor* Alfred Theorin, Dept. of Automatic Control, Lund University, Sweden Charlotta Johnsson, Dept. of Automatic Control, Lund University, Sweden (examiner) *Sponsoring organization*

Title and subtitle

Implementation of Service Orchestrated control procedures in OPC UA for JGrafchart

Abstract

The automation industry is facing many challenges with higher demands on their production process. Technology used today does not allow for fast changes in the production line. This thesis will investigate how services can be modelled using a new standard OPC UA for data exchange. Encapsulation of the mechatronic functions as services will allow for creating control software using a SOA approach. An experimental set-up will investigate how an OPC UA server and client are created.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title 0280-5316	*ISBN*

Language English	*Number of pages* 1-52	*Recipient's notes*
Security classification		

http://www.control.lth.se/publications/