TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Martin Lahtein 178224IASM

EXTERNAL SIL SIMULATOR FOR VALMATIC AUTOMATION SYSTEM

Master's Thesis

Supervisor: Eduard Petlenkov
PhD
Co-supervisor: Hando Nurga
MSc

Tallinn 2019

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Martin Lahtein 178224IASM

VÄLINE TARKVARALINE SIMULAATOR VALMATIC AUTOMAATJUHTIMIS-SÜSTEEMILE

Magistritöö

Juhendaja: Eduard Petlenkov
PhD
Kaasjuhendaja: Hando Nurga
MSc

Tallinn 2019

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Martin Lahtein

22.04.2019

Abstract

Nacos Valmatic Platinum is an integrated alarm monitoring and control system (IAMCS) developed by Wärtsilä Valmarine and used on marine vessels. Simulation of field devices in factory acceptance tests are currently done by either using electronic circuits or including the simulation software in the project itself. The aim of this thesis is to develop a simulator that handles the problems of these methods. Using a virtual implementation of Valmatic system and connecting to the IAMCS via OPC UA protocol solves both the issues of scalability and transparency. The process of configuring a simulator for an existing control system is simplified by automating most parts of the procedure, requiring only the knowledge of standard tools used in Valmarine from the user.

This thesis is written in English and is 63 pages long, including 8 chapters, 23 figures and 6 tables.

Annotatsioon
Väline Tarkvaraline Simulaator Valmatic Auomaatjuhtimissüsteemile

Nacos Valmatic Platinum on integreeritud alarmi-, monitoorimis- ja juhtimissüsteem (IAMCS) arendatud Wärtsilä Valmarine poolt, mida kasutatakse laevadel. Juhtimisseadmete simuleerimist süsteemi kasutuselevõtu testidel teostatakse peamiselt kahel viisil – elektroonikaskeemidega või lisades simulatsioonitarkvara testitavale projektile. Käesoleva töö eesmärk on luua simulaator, mis parandab nende meetodite kasutamisel ilmnevad probleemid. Kasutades virtuaalset Valmatic süsteemi, ühendades selle IAMCSga OPC UA kommunikatsiooniprotokolli kaudu, lahendatakse nii skaleeruvuse kui simulatsiooni läbipaistvuse probleemid. Simulaatori ülesseadmise protsess olemasoleva juhtimissüsteemi jaoks on suuremalt osalt automatiseeritud, seades kasutaja teadmistele vaid nõude olla tuttav Valmarines kasutatud standardsete tööriistadega.

Lõputöö on kirjutatud inglese keeles ning sisaldab teksti 63 leheküljel, 8 peatükki, 23 joonist, 6 tabelit.

List of abbreviations and terms

SIL	Software-in-the-loop
HVAC	Heating, ventilation, and air conditioning
I/O	Input/output
PAC	Process application controller
IAMCS	Integrated alarm, monitoring, and control system
FIC	Field interface controller
MFD	Multi-functional display
HMI	Human-machine interface
TCP/IP	Transmission control protocol/Internet protocol
SCADA	Supervisory control and data acquisition
RTU	Remote telemetry unit
IED	Intelligent electronic device
UDFB	User-defined function block
NAT	Network address translation
XML	Extensible Markup Language
LCH	Level control high
LCL	Level control low
LAHH	Level alarm high high

Table of contents

List of figures

List of tables

1 Introduction

Nacos Valmatic Platinum is an integrated alarm monitoring and control system (IAMCS) developed by Wärtsilä Valmarine and used on marine vessels. It provides distributed process control for systems that ensure safe and stable operation of the ship. The IAMCS includes all field devices, from simple sensors, measuring some physical quantity, to pumps and other actuators carrying out the control functions for machinery, HVAC and power management systems. Other large systems such as main engines, fuel management and propulsion are also interfaced to the IAMCS. Controllers in Valmatic are redundant to ensure system operability in case of a failure in one of the controllers. The automation system is described in further detail in the second chapter of this thesis.

Before delivering the developed control software to a vessel, the functionality is verified by the client in a factory acceptance test. Currently, two methods are used for simulating field devices – either by external electronic circuits connected to I/O modules or by including simulation software inside the project itself. Both methods have significant drawbacks.

Simulating with external circuits works well for smaller systems, but is not suitable for extensive system testing, where large number of devices is involved. Since all the circuits have to be implemented in hardware, this approach can become rather expensive. Modifying simulation circuits for non-standard devices is hard and generally requires creating a new circuit after all. Finally, the setup of hardware simulation is a very time-consuming task. The advantage of this type of simulation is that the IAMCS is not altered in any way and remains truthful to the system that is eventually used on the ship.

Including simulation programs inside the automation project itself introduces significant alterations to the software that is eventually used on the vessel. This practice is not allowed by some shipyards, because it reduces the simulation transparency and makes the product credibility uncertain. The advantages of this approach are scalability and being easily configurable.

The goal of this thesis is to develop a simulator that captures the benefits of both approaches and solves their disadvantages. Therefore, the simulator has to be implemented in software to deal with the scalability issue. Additionally, it must be connected to the automation system externally, making as few alterations to the IAMCS as possible. This makes the simulations more transparent for the client. Finally, the simulator has to be easily configurable by all engineers involved in developing the control software to reduce the time spent on simulator setup.

Main part of this thesis is divided into six chapters:
In chapter two, an overview of the Valmatic automation system is given and the requirements for the simulator are established.
In chapter three, suitable communications protocol for connecting the simulator to the IAMCS is selected.
In chapter four, suitable platform for simulation implementation is selected.
In chapter five, the communications protocol is studied in more detail and the simulator is tested for scalability performance requirements.
In chapter six, structured steps for setting up the simulator are proposed. Ways for automating parts of this process are also studied.
In chapter seven, a simulator is set up for an existing automation system to demonstrate feasibility of the solution and identify the limitations.

2 Simulator requirements

Valmatic automation system enforces redundancy on multiple levels. System bus network is connected in ring topology to ensure operability in the case of a faulty cable. Process application controllers (PAC) are redundant to increase reliability and enable updating software without interrupting the process control. IAMCS layout is given on Figure 1.

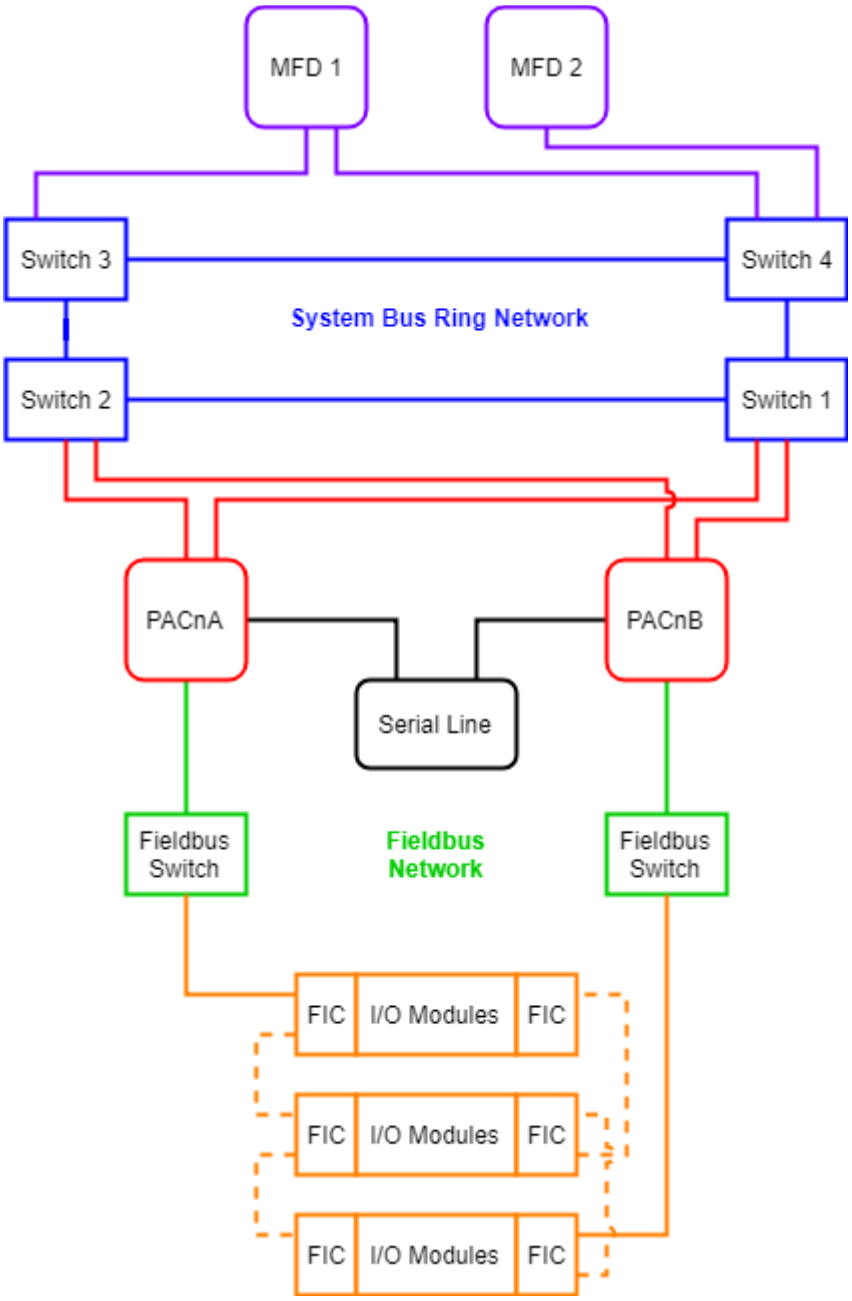

Figure 1. Valmatic IAMCS Layout

Field devices are scanned with an interval of 20 ms. Standard PAC cycle time for control logic is 400 ms or 800 ms, depending on application. Field interface controllers (FIC) are used to establish communication between I/O modules and PACs. Serial line can be used for interfacing with other systems. Multi-functional Displays (MFD) serve the purpose of HMI.

Control programs for Valmatic projects are developed with programming languages defined in IEC 61131-3 standard using Straton IEC Development Environment. In addition to control logic, these programs handle communication with fieldbus devices, system maintenance as well as interacting with HMIs and are executed cyclically by Straton IEC Runtime software in PACs.

Device control programs are created with user-defined function blocks (UDFB), that serve as templates for certain types of devices. These function blocks can have 4 types of variables. Input variables are read-only and not used in Valmatic. Output variables are write-only and are used for tags that handle displaying information on HMI. In-out variables support both read and write functions and are used for both input and output signal tags, as well as tags that are accessible from the HMI, such as device mode selection, operator commands and timing parameters. UDFB structure and instance for an automatically controlled single-speed motor is seen on Figure 2.

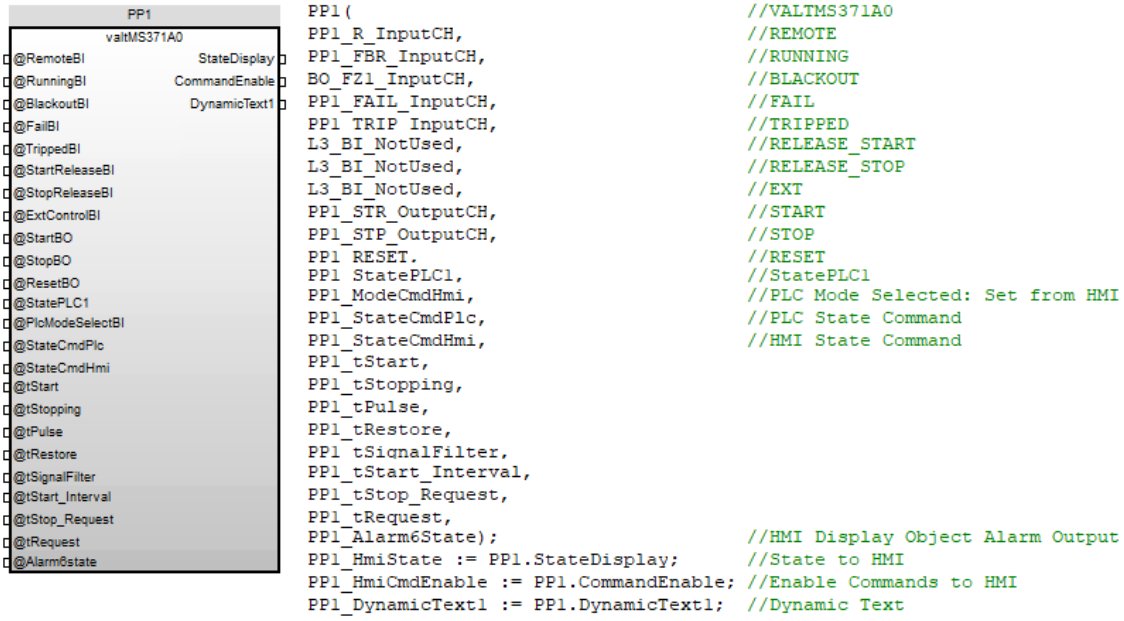

Figure 2. Single-speed motor control UDFB and instance

Before each production system is accepted by the customer for delivery, it must pass a formal systems acceptance test. This test determines that the product is properly constructed, meets key requirements and is ready for operational use. Additionally, the test must be capable of being performed quickly without increasing the manufacturing costs [1]. Therefore, the first requirement is to implement simulation logic in software, which can be scaled to larger projects without increasing the cost. This type of simulation, where the plant is modelled in software without physically connecting any of the I/O signals, is called software-in-the-loop (SIL) simulation. It provides an inexpensive way for validation of the control system [2]. Implementing the simulator in software requires making automation system input and output signal tags accessible to the simulator. The latter will read output signal tag values and execute simulation logic on that information resulting in updating the values of input signal tags. E.g. activating feedback after a delay when start command is given.

Second requirement for the simulator is to be externally connected to the automation system. Running simulation programs in the same project as control programs can lead to uncertainties about the product credibility and is not suitable for formal acceptance testing. Establishing connection to the automation system via the system bus network seen on Figure 1 can be done by using a suitable communications protocol that works on TCP/IP. Setting up this connection should be done with as few adjustments to the existing automation system as possible.

Performance requirements are defined by the ability of the system to detect signals that are activated only for a certain period of time. Activating an output command for one PAC cycle means the pulse length is 400 ms. These signals should be caught by the simulator. Control system PACs are running with a fixed interval, which can vary by some degree every cycle. They should be able to detect input signals generated by the simulator, that are slightly longer than the cycle time – e.g. 500 ms signals for PACs with 400 ms cycle. This property must scale to 2000 signals per PAC without deteriorating the performance. Number of PACs used in a project can be up to 40.

Final requirement is the possibility for every engineer to develop their own simulation programs and configure them to the simulator. For most of the devices, standard simulation templates can be developed, but every project varies in some way or another. For these variations, new simulation programs are best to be written by the engineer who

develops control software for the same part. This means that the simulator programming language should be familiar or easily learnable for all engineers working on the project.

List of requirements:

1. Software implementation of simulated plant
2. External connection to the automation system
3. Minimal changes to the automation system for simulator setup
4. Ability to detect fixed length signal pulses
    a. 400 ms pulse output signals
    b. 500 ms pulse input signals
5. Scalability
    a. Up to 2000 signals per PAC
    b. Up to 40 PACs
6. Easily configurable and usable by every software engineer

3 Communications protocol

This chapter gives an overview of fieldbus communication protocols supported by Straton IEC that work on TCP/IP. Protocol suitability is determined on adoption and widespread use in the industrial automation field to ensure sustainability of the solution, as well as potential communication performance.

One of the main focuses on the development of SCADA protocols has been interoperability. This means that devices made by different vendors can use the same protocol to communicate with each other without any need for mapping between protocols [3]. SCADA protocol with wide support from different vendors offers more options for the simulator implementation environment in the next chapter. Widely used SCADA protocols are Modbus, IEC 60870-5-104, DNP3, MQTT, OPC Classic and OPC UA. Straton IEC supports every one of these protocols. For MQTT and OPC DA only client functionality and for OPC UA only server functionality is supported.

3.1 Modbus TCP/IP

Modbus is an openly published network protocol developed by Modicon (now Schneider Electric) in 1979. It is most widely used network protocol used in the industrial manufacturing environment. Modbus devices communicate using a master-slave model in which only the master can initiate transactions. Slaves respond by supplying the requested data to the master, or by taking requested action [4]. Modbus TCP/IP embeds Modbus packets in TCP segments and assigns TCP port 502 to the Modbus protocol. Data from slaves is accessed by using function codes to read or write coils (single bit value) or registers (16-bit value). Multiple data items located on consecutive addresses can be accessed in a single query and 32-bit values (E.g. REAL data type in IEC 61131-3) can be mapped to two consecutive registers [5]. Main benefit of Modbus protocol is wide adoption in the industrial automation field, meaning that most engineers are familiar with the protocol. Biggest drawback is the master-slave model. Requests are initiated by the master periodically, even when the value of demanded variable has not changed. This introduces unnecessary traffic to the network and may pose problems for scalability.

3.2 IEC 60870-5-104

IEC 60870-5 is a group of communication protocol standards for electric utility systems, mainly power system automation, developed by the IEC Technical Committee 57. The IEC 60870-5-101 companion standard profile defines data types, commands and other communication details that are needed for communication with RTUs in electrical systems. It is extended by IEC 60870-5-104 by using TCP/IP as the underlying transport protocol [3]. This standard has limited support from vendors outside the electric utility field. This narrows the number of options for simulation environments significantly. Additionally, technical support and possibility for further development is also limited.

3.3 DNP3

DNP3 (Distributed Network Protocol Version 3) is a telecommunications standard that defines communications between master stations, RTUs and other IEDs. It was developed to achieve interoperability among systems in the electric utility industry and designed specifically for SCADA applications. DNP3 supports multiple-slave, peer-to-peer and multiple-master communications. It supports polled and quiescent operation modes. The latter means that in the absence of change, the system remains in quiet state and the outstation sends a response only to report a change in the system [6]. DNP3 has the same issues as IEC 60870-5-104 by being used only by the electric utility industry.

3.4 MQTT

MQTT (Message Queueing Telemetry Transport) is a lightweight protocol extension to TCP/IP. It avoids many of the overheads associated with Modbus by using a publish-subscribe model. Nodes must register their interest in receiving information from a publisher by contacting a broker. This protocol is designed for low bandwidth networks and devices with limited processing capability [7]. Network bandwidth limitations are not significant in Valmatic IAMCS. MQTT is designed for very specific applications and not very widely adopted in the industrial automation field, compared to Modbus or OPC. This limits the number of simulation environment options.

3.5 OPC Classic

Open Platform Communications (originally OLE for Process Control) Classic is a set of standards that describe server-client software architecture to collect process data from devices and pass them to SCADA systems. The Data Access (DA) specification proposes a way to exchange real-time process data in a specified format that includes process value, time stamp and reliability. Alarms and Events (A&E) specification defines an interface for server and clients to exchange information on alarms, events and their acknowledgements. Historical Data Access (HDA) specification enables querying for data values and statistics within a specified time span. Commands specification provides an interface for executing defined commands. OPC Classic has become a de-facto standard accepted in the process and automation industry [8].

3.6 OPC Unified Architecture

The issue with OPC Classic is using different address space for every specification, which cannot be merged together even if the same variable with its aggregated values is used. OPC UA unifies all of these address spaces into one, characterizing the new object as variables, events and methods, providing the functionalities of OPC-DA, OPC-HDA, OPC-A&E and OPC-Commands. Security was also neglected in the development of OPC Classic. OPC UA provides security on both the application and communication layers. Former deals with user authentication and authorisation and the latter includes confidentiality, integrity and application authentication [8]. Implementation of OPC UA Communication Stack is not linked to any specific technology, which allows it to be mapped to future technologies as necessary, without negating the basic design. OPC UA supports two methods for data exchange between client and server. Read and write services allow the client to read or write the values of one or more variables. The publish-subscribe model uses monitored items to exchange data between client and server. The three types of monitored items include subscription for data changes of variable values, pre-defined events or aggregate values calculated based on current variable values in client-defined time intervals [9]. OPC UA is accepted as an International Electrotechnical Commission standard IEC 62541. Wide support from vendors in the automation industry makes OPC UA a suitable option for the simulator. The publish-subscribe model provides an efficient use of network bandwidth by reporting only changed values, improving system scalability.

3.7 Conclusion

IEC 60870-5-104 and DNP3 are electric utility industry-specific protocols, with few applications outside this field. MQTT is for limited network bandwidth uses, which is not an issue for the Valmatic IAMCS. Modbus and OPC UA are meant for more general use and have support from many vendors. OPC UA publish-subscribe model provides superior performance over Modbus master-slave model. Straton supports only OPC UA server functionalities, but there are many client options on the market, which are studied in the next chapter.

4 Simulation platform

As described in chapter 2, the simulator will read output signal tag values from the automation system, execute simulation logic on that information and then update the values of input signal tags. Models for devices controlled by Valmatic are simple, consisting mostly of Boolean algebra, timed delays and linear ramping of analog values.

In this chapter an overview of OPC UA compliant environments for the implementation of simulation logic is given. Automated systems on a vessel may consist of many duplicate devices, that have identical simulation logic. Programming them should be done with object-oriented approach in a widely used language. Pricing of the third-party software is considered to make the final decision.

4.1 MATLAB/Simulink

Simulink is a popular control system simulation software developed by MathWorks that is covered in most engineering curriculums. In [10] is provided a solution for communication with OPC UA servers using Open62541, an open-source implementation of OPC UA in ISO C language. For continuous simulation, Simulink S-Functions are used to execute the C code. MATLAB supports object-oriented programming and is priced at €2000 for perpetual license [11]. Simulink is mainly used for testing controllers designed for continuous process control. Systems controlled by Valmatic have simple models and do not need such sophisticated simulation. Additionally, no official support for OPC UA connections by MathWorks makes the sustainability of the solution uncertain.

4.2 National Instruments LabVIEW

LabVIEW is systems engineering software for monitoring and control applications developed by National Instruments. It provides a graphical programming interface for enhanced visualization of the controlled process. Both OPC UA communications and object-oriented programming are supported. Perpetual licence of LabVIEW costs $2999

with $660 added for OPC UA toolkit [12]. As with Simulink, LabVIEW's graphical programming interface is more suited for testing controllers for continuous processes and not convenient for systems consisting of many devices and processes. One of the requirements for the simulator was the ability for every software engineer to design their own simulation programs. Since LabVIEW is not in the set of standard tools used by Valmarine engineers, arming them with required knowledge would take a lot of time and introduce additional costs for the company.

4.3 Inductive Automation Ignition

Ignition is a SCADA platform software developed by Inductive Automation. It provides graphical HMI with an extensive library of symbols. Python scripts can be used for component customization and tag handling for simulation logic. Connection to OPC UA servers is supported. Licence pricing for Ignition starts from $10000 [13]. As with LabVIEW, Ignition and Python are also not used as standard tools by Valmarine engineers and would also require training. Ignition is also most expensive of the observed options.

4.4 Valmatic

Another option is to use virtual implementation of Valmatic system for the simulator. This means that PAC runtime, executing the simulation logic, is ran on a virtual machine. Virtual MFD can be used for real-time human interaction with the simulation. Multiple PACs can be used on the same simulation computer to distribute simulation programs by systems or improve scalability. The number of virtual PACs is limited only by the processing power of simulation computer. This concept is illustrated on Figure 3.

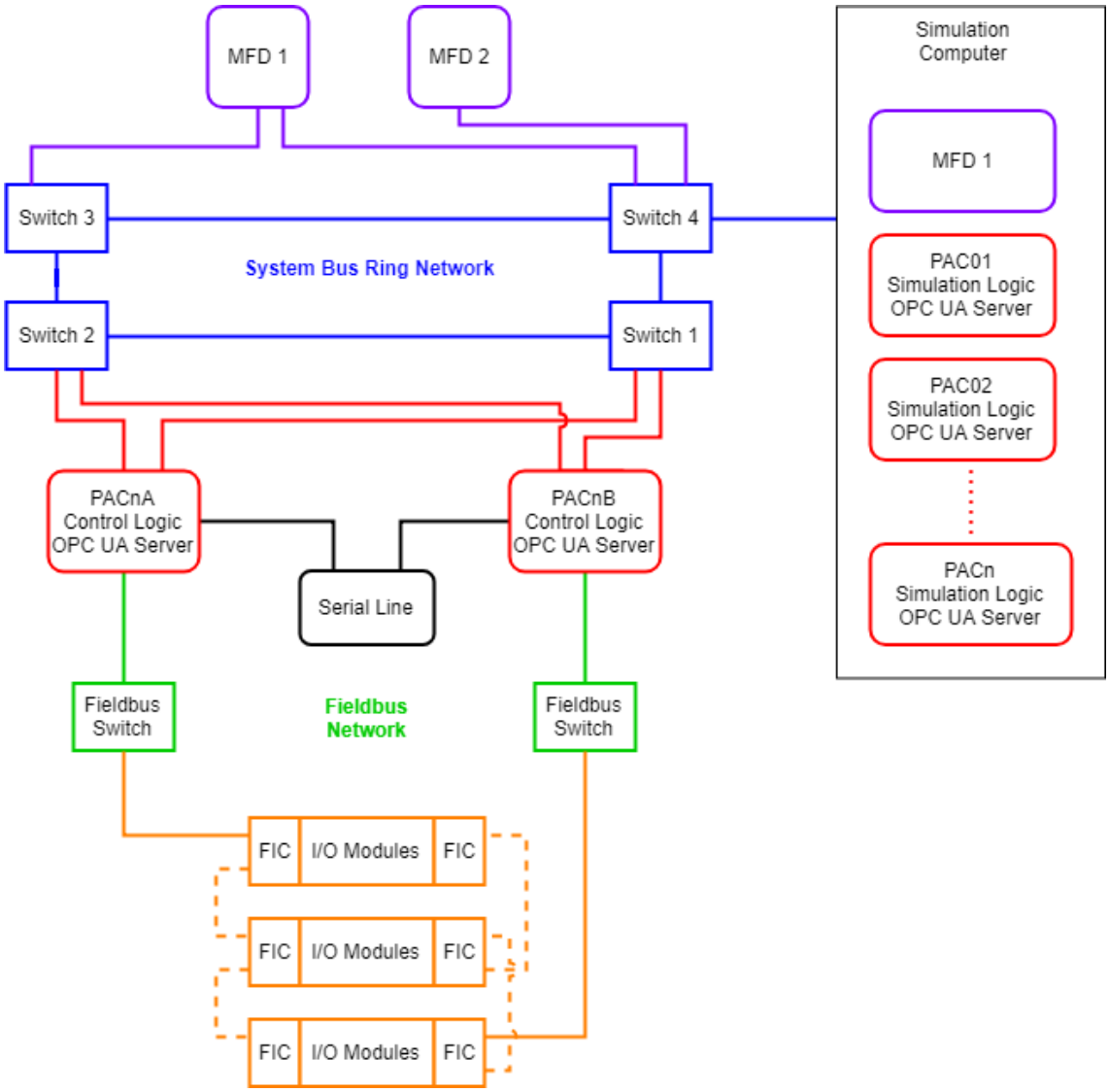

Figure 3. Valmatic IAMCS layout with simulator

Valmatic supports only the configuration of OPC UA servers meaning that the communication between the servers on automation and simulator PACs is established by using third-party OPC UA server bridging software. One of the options on the market is Skkynet DataHub UA Bridge. It provides the functionality of connecting to multiple servers as a client, read data from one and write this data to another, forming a bridge. Price of perpetual licence for this software is $1680 with $280 annual support plan [14].

This solution brings the benefit of using IEC 61131-3 defined programming languages, that are familiar to every control engineer in Valmarine, reducing the adoption time considerably.

4.5 Conclusion

Training engineers to use new software can be a long and expensive process. Since neither Simulink, LabVIEW nor Ignition are in the set of standard tools used by Valmarine engineers, using any of them as simulation environment requires equipping engineers with mandatory knowledge, before the simulator can be used efficiently. Using Valmatic for simulation brings the benefit of familiar programming and configuration interfaces, reducing the adaption time significantly. IEC 61131-3 defined programming languages meet all of the requirements for device modelling. Even with using Skkynet DataHub UA Bridge software for server bridging, this option is economically most reasonable.

5 Performance testing

In chapter 2, the requirements for performance were stated – automation system should be able to detect signals with pulse length of 500 ms and simulator must detect signals activated for one automation system PAC cycle. To confirm this property, a test system, that generates required signals, was developed. In the first section, the simulation system configuration and connection to automation system is covered. Second subchapter gives an overview of configurable parameters on both the client and server side and suggests optimal values for best performance. Finally, the tests results are analysed.

5.1 Simulation system structure

As depicted on Figure 3, the simulator computer may be running multiple virtual machine PACs. The number of servers that DataHub OPC UA Bridge can connect to is not limited but having too many bridges may deteriorate the performance during large system testing. This issue can be mitigated by running multiple instances of DataHub OPC UA Bridge and limiting the number of servers that one instance is connected to. This structure is illustrated on Figure 4.

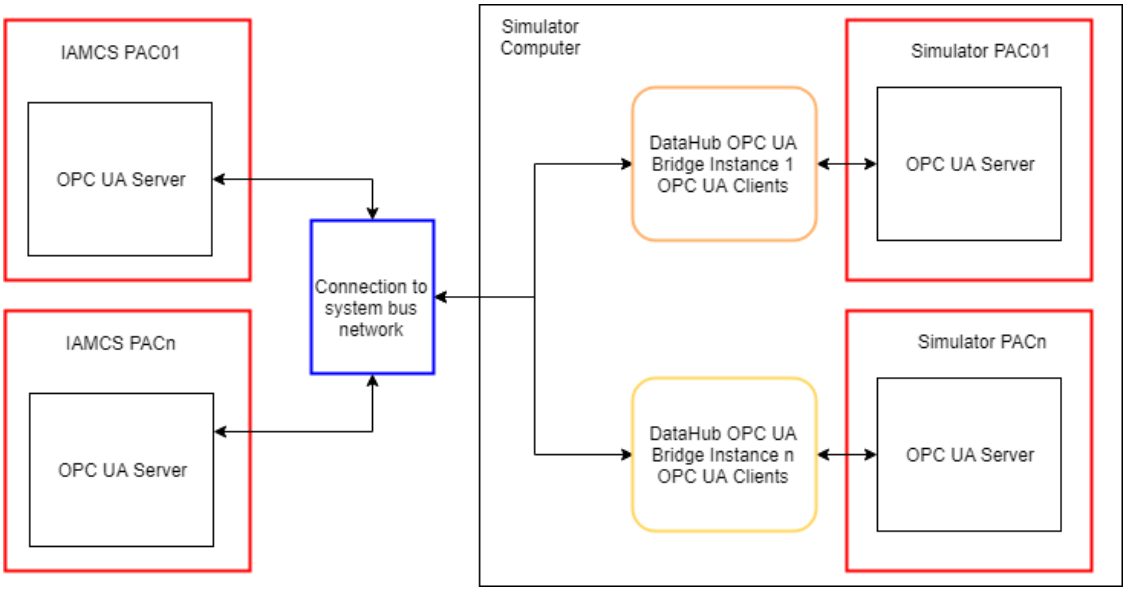

Figure 4. Simulation system configuration

PAC IP addresses are fixed in the Valmatic system, which results in identical addresses for both IAMCS PACs and virtual PACs on simulator computer, that cannot be connected directly into the same network. To mitigate this issue, network address translation (NAT) can be used. NAT is a common security function for changing the IP address during Ethernet packet transmission enabling a device to have different internal and external IP addresses [15]. Two routers can be used to create a wide area network (WAN) between the two local area networks (LAN). NAT rules can be configured for one of the systems to translate their LAN IP addresses to WAN IP addresses that can be accessed by devices in the other LAN. Figure 5 illustrates this concept.

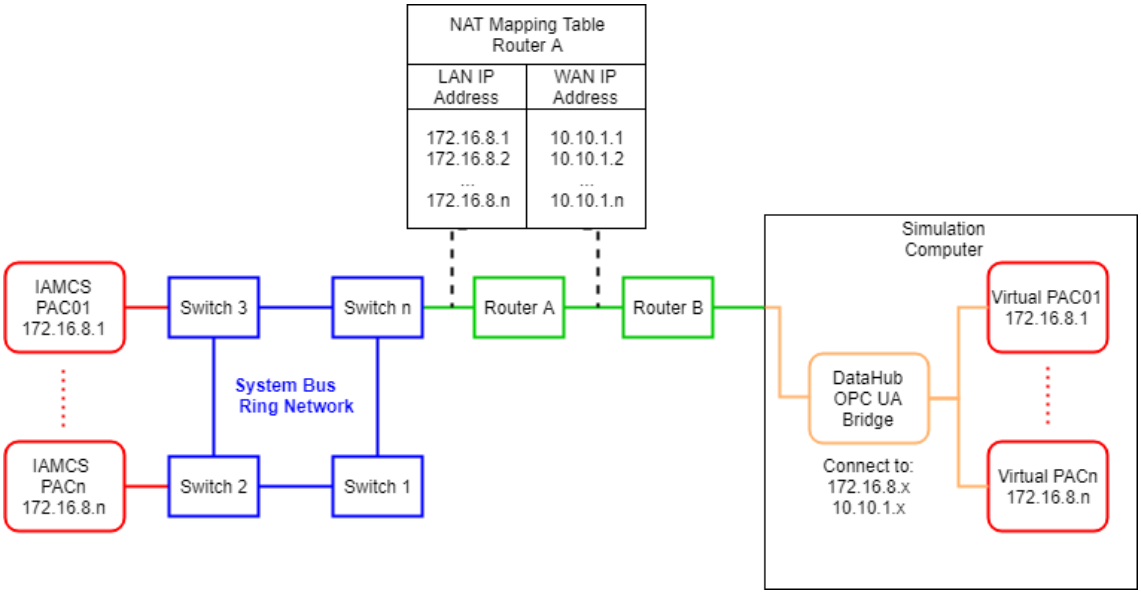

Figure 5. Network connection between simulator and IAMCS

5.2 OPC UA configuration

OPC Unified Architecture is built on a client-server model, where servers are used to store data, that can be accessed by one or multiple clients. Two methods for data exchange between clients and servers are offered – basic read/write operations and the publish-subscribe model [9]. The purpose of this chapter is to study these models, choose most suitable one among them and evaluate the optimal configuration for selected option.

Words starting with capital letters are used to describe terms and concepts only applicable in the context of OPC Unified Architecture and are defined in OPC UA Specifications 1 through 14.

Each OPC UA Server has an AddressSpace – a set of Nodes accessible by Clients using OPC UA Services. Nodes are described by Attributes – E.g. Node ID, description, value, value data type, etc. Node value Attribute is linked to a real object that is cyclically sampled by the Server [16]. Two methods exist for the Client to access Node Attributes.

The Read Service is used to read one or more Attributes of one or more Nodes. Server responds to Read request with queried information regardless of whether the Attribute value has changed or not [17].

Publish-Subscribe model uses MonitoredItems to access Node Attributes. MonitoredItems are entities in the Server created by the Client that monitor some real-world process via AddressSpace Nodes. Upon the detection of a data change or an event/alarm occurrence, a Notification is generated and transferred to the Client by a Subscription. A Subscription is an endpoint in the Server used for Notification publishing to Clients. Clients control the publishing rate by sending Publish Requests [18]. Figure 6 illustrates both the Read Service and the Publish-Subscribe model.

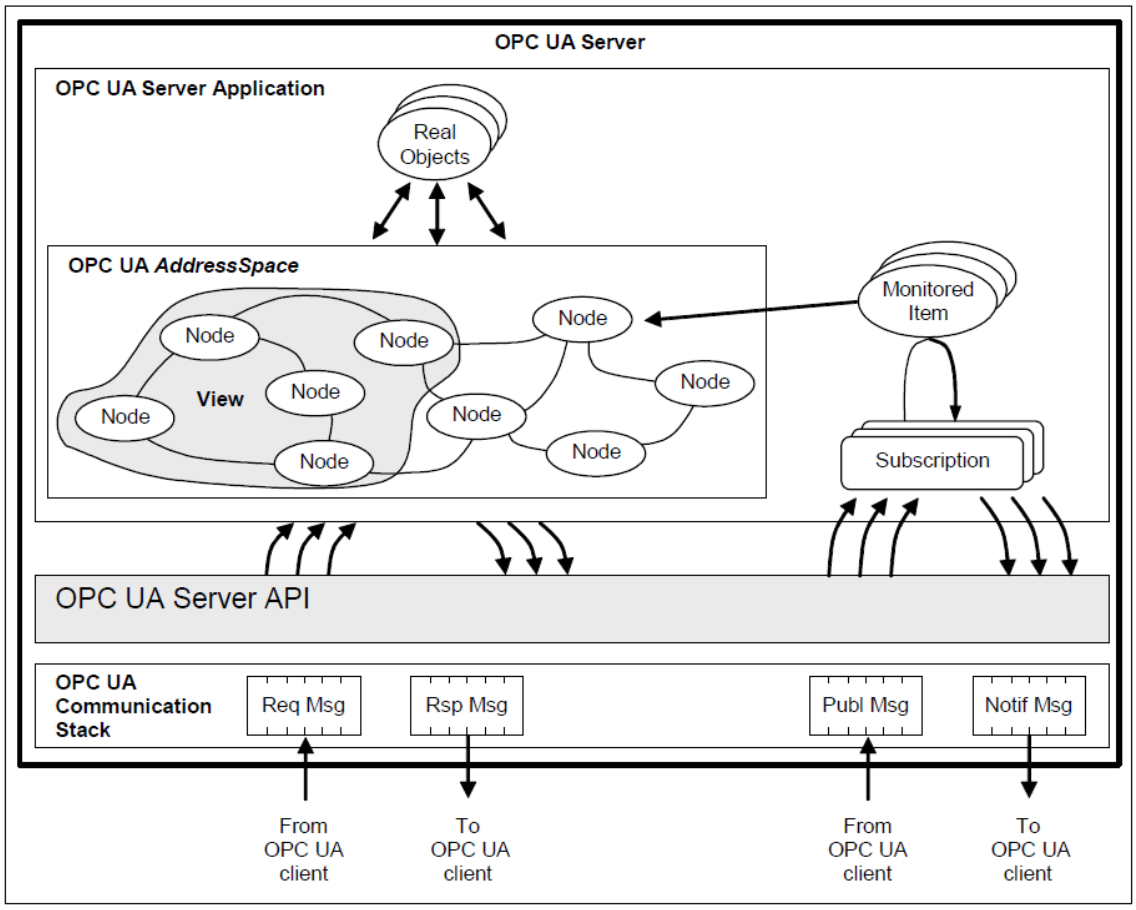

Figure 6. OPC UA Client-Server model [18]

Clients can write new values to Server Node Attributes by using the Write Service. Servers determine the value for Node AccessLevel Attribute that indicates if Clients are enabled to use the Write Service to alter the values of Node Attributes [16].

Valmatic simulations have many binary signals, that change value infrequently. Requesting the values of these unchanged variables is unnecessary and causes extra load on the network. The Publish-Subscribe model sends only Attribute values that have changed, reducing the size of transferred messages significantly. The Pub-Sub model is used for data exchange between automation system and simulator to ensure best performance.

Both the Server configured in Straton IEC and Clients configured in DataHub OPC UA Bridge present a set of parameters that describe the Publish-Subscribe model of communication. The purpose of next subchapters is to study these parameters in detail and propose values for optimal communication performance.

5.2.1 Straton IEC OPC UA server parameters

Establishing a connection between a Client and a Server requires opening a SecureChannel, a communication channel that ensures the confidentiality and integrity of all Messages exchanged with the Server [19]. This channel is used to secure the data coming from application Sessions, logical connections between Clients and Servers that are used to manage state information. Examples of state information are Subscriptions, user credentials and continuation points for operations that span multiple requests [18]. The routine work of information, settings and commands transmission between Clients and Servers is the responsibility of Sessions. The number of Sessions should be limited by the Server application for protection against rogue Clients and denial of service attacks [17].

Parameter “Max. sessions” limits the number of clients that can be connected to the server simultaneously. Value should be equal or greater than the number of required bridges to this server.

Each MonitoredItem identifies the Node Attribute to monitor in the Server AddressSpace and the Subscription to use for periodical Notification publishing. Notifications are data structures that describe the occurrence of data changes and Events. Most important

parameters defined for MonitoredItems are the sampling interval, filter rule and queue size. The rate at which the Server is sampling its underlying source for data changes is defined by the sampling interval. Every time a MonitoredItem is sampled, it is evaluated by the Server against the defined filter criteria – E.g. deadband calculation. If this evaluation produces a positive result, a Notification is generated and queued for transfer by the Subscription. The size of the queue is defined when the MonitoredItem is created. Most relevant parameter for a Subscription is the Publishing Interval – a cyclic rate at which the Subscription attempts to send a NotificationMessages to the Client [17]. The minimum value of publishing interval is limited to 100 ms on Straton IEC OPC Servers. This concept is illustrated on Figure 7.

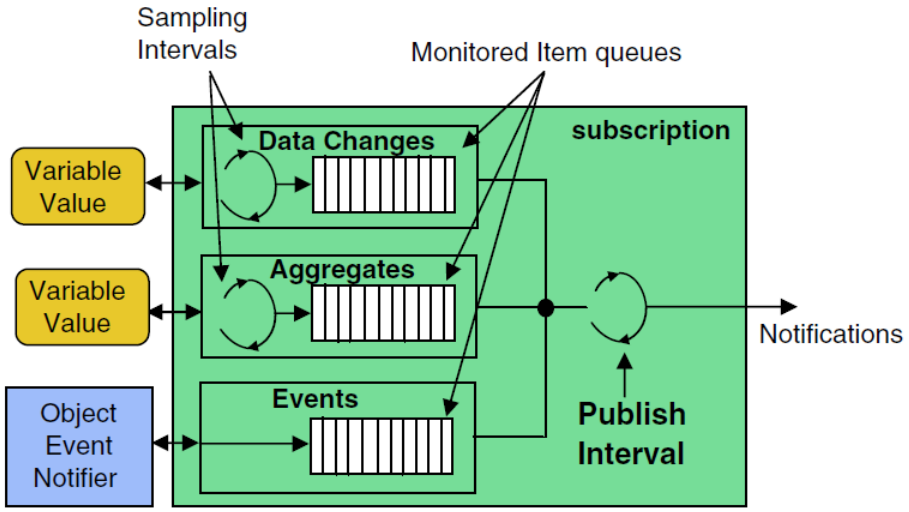

Figure 7. OPC UA Subscription [9]

NotificationMessages contain a list of Notifications that have not been sent to the Client yet. Publish Requests produced by the Client are queued to the Session upon reception. Each publishing cycle, one request is de-queued and processed by a Subscription related to this Session, if there are any unreported Notifications. When there are not, the Publish Request is not de-queued from the Session, and the Server waits until the next cycle and checks again for Notifications. At the beginning of a publishing cycle, if there are unreported Notifications but no Publish requests queued, the Server enters a wait state for a Publish Request to be received and processes it immediately upon reception [17].

Subscriptions have a keep-alive counter that counts the number of consecutive publishing cycles in which there have been no Notifications to report to the Client. When the

MaxKeepAliveCount parameter value is reached, a Publish request is de-queued and used to return a keep-alive message to inform the Client that the Subscription is still active. Subscriptions have a lifetime counter that counts the number of consecutive publishing cycles in which there have been no available Publish Requests. When the LifetimeCount parameter value is reached, the Subscription is closed [17]. Subscriptions inside a Session are illustrated on Figure 8.

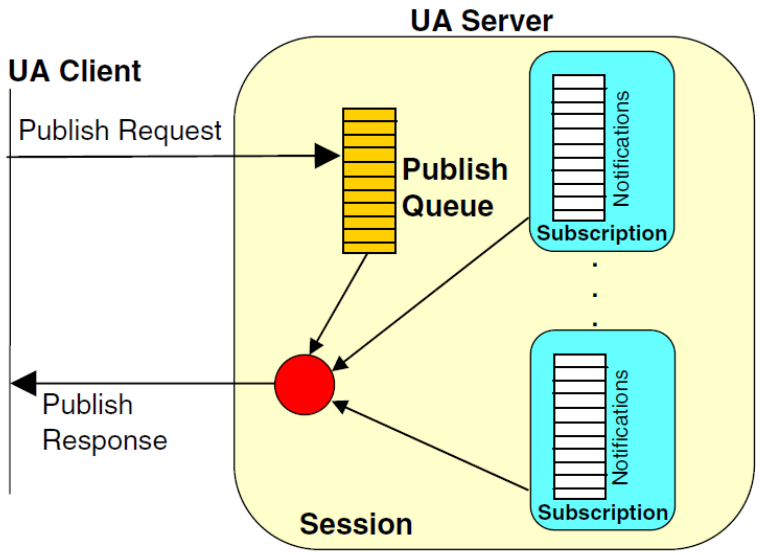

Figure 8. OPC UA Session [9]

Normally the number of created Subscriptions should be minimal, but two exceptions to this rule exist:

a) Items that are monitored require different Publishing Intervals.

b) Total number of MonitoredItems exceeds the enabled limit of MonitoredItems per Subscription set by the Server.

The number of Subscriptions created by a client is determined by the parameter "Max. MonitoredItem per Subscription". The value of this parameter should always be greater than or equal to the total number of items to be monitored.

Parameter "Max. Subscriptions per Session" limits the number of subscriptions that can be created within a Session. Value for this parameter should enable creating enough Subscriptions to monitor all required items.

Filling the publish queue with infinite number of requests should be limited by a Server application. When a Server receives a new Publish Request that exceeds this limit, it shall de-queue the oldest and return a Publish Response with the result set to

"Bad_TooManyPublishRequests". If a Client receives this Service result for a Publish Request, it shall wait until a response is received for one of the active requests before sending another Publish Request [17]. The size of Publish Request queue is set by "Max. PublishRequest per Session" parameter. The value should be greater or equal to the number of created Subscriptions for the Session.

Final configurable parameter "Max. DataChangedValue per MonitoredItem" determines the size of MonitoredItem queue seen on Figure 7. Two policies for managing new Notifications when the queue is full exist and are selected by discardOldest parameter when creating a MonitoredItem. If the value for this parameter is TRUE, the Server discards the oldest Notification from the queue and inserts the new one. Setting the parameter value to FALSE makes the Server replace the last Notification added to the queue with the most recent one. Parameter maxNotificationsPerPublish determines the number of Notifications sent in a single Publish response with value 0 indicating no limit. If the queue size is one, it becomes a buffer that always contains the newest Notification. In this case of sampling interval of the MonitoredItem being faster than the publishing interval of the Subscription, the MonitoredItem will be over sampling and the Client will always receive the most up-to-date value [17]. In the context of the simulator, only the latest value is of interest to us and therefore the value for this parameter should be equal to 1.

Table 1 contains the parameter value ranges of OPC UA servers created in Straton IEC.

Table 1. Straton IEC OPC UA server parameter value ranges

Parameter	**Value range**
Max. sessions	1 – 100
Max. Subscriptions per Session	0 – 100
Max. MonitoredItem per Subscription	0 – 65535
Max. PublishRequest per Session	1 – 100
Max. DataChangedValue per MonitoredItem	1 – 100

5.2.2 DataHub OPC UA Bridge parameters

The “Maximum Update Rate (milliseconds)” parameter is used to define values for both the sampling and publishing intervals and is limited to a minimum of 10 ms. Note that Straton IEC OPC UA Servers limit the minimum value of publishing interval to 100 ms. Therefore, a request for an update rate of 10 ms would result in sampling interval of 10 ms and publishing interval of 100 ms.

Parameter “Monitored Item Queue Size” determines how many Notifications are stored in the queue as discussed in 0. The value should match “Max. DataChangedValue per MonitoredItem” configured for the OPC UA Server.

Parameter “Max Request Item Count” regulates the number of MonitoredItems per Subscription and is covered in 0. The value should be equal to the one configured for “Max. MonitoredItem per Subscription” in the OPC UA Server.

Table 2 contains some of the fixed parameter values requested by DataHub Clients when creating a Subscription.

Table 2. Fixed parameter values requested by DataHub Clients

MaxKeepAliveCount	1
LifetimeCount	1000
discardOldest	FALSE
maxNotificationsPerPublish	0

5.3 Test results

In the second chapter, performance requirements for the simulator were stated as the ability to detect with 400 ms and 500 ms pulses by the simulator and automation system respectively. To confirm this property, a test system was set up with the configuration seen on Figure 4 and Figure 5. Automation system PACs were configured to run with a cycle of 400 ms. No execution time limits were set for virtual controllers in the simulator.

Test programs in both the automation system and simulator have 1 input and 1 output variable. The program in automation system was configured to toggle the output signal every cycle – resulting in a square wave signal with 800 ms period. Simulator test program was configured to generate output pulses with a period of 1000 ms. Both programs count the number of times the input signal is activated. 1000 instances of these programs were created to meet the requirement of 2000 for total number of signals. The output of each instance was bridged to an input of the instance on the other side with DataHub OPC UA Bridge. The number of generated signals by both sides is 100 and is compared to the number of input signals counted on the other side.

OPC UA Server and Client parameter values used in the test are presented in Table 3.

Table 3. Test system parameters

Server parameters	**Value**
Max. Sessions	1
Max. Subscriptions per Session	1
Max. MonitoredItem per Subscription	2000
Max. PublishRequest per Session	1
Max DataChangedValue per MonitoredItem	1
Client (Bridge) parameters	**Value**
Maximum Update Rate (milliseconds)	10
Monitored Item Queue Size	1
Max Request Item Count	2000

Figure 9 illustrates the test setup from simulator's point of view.

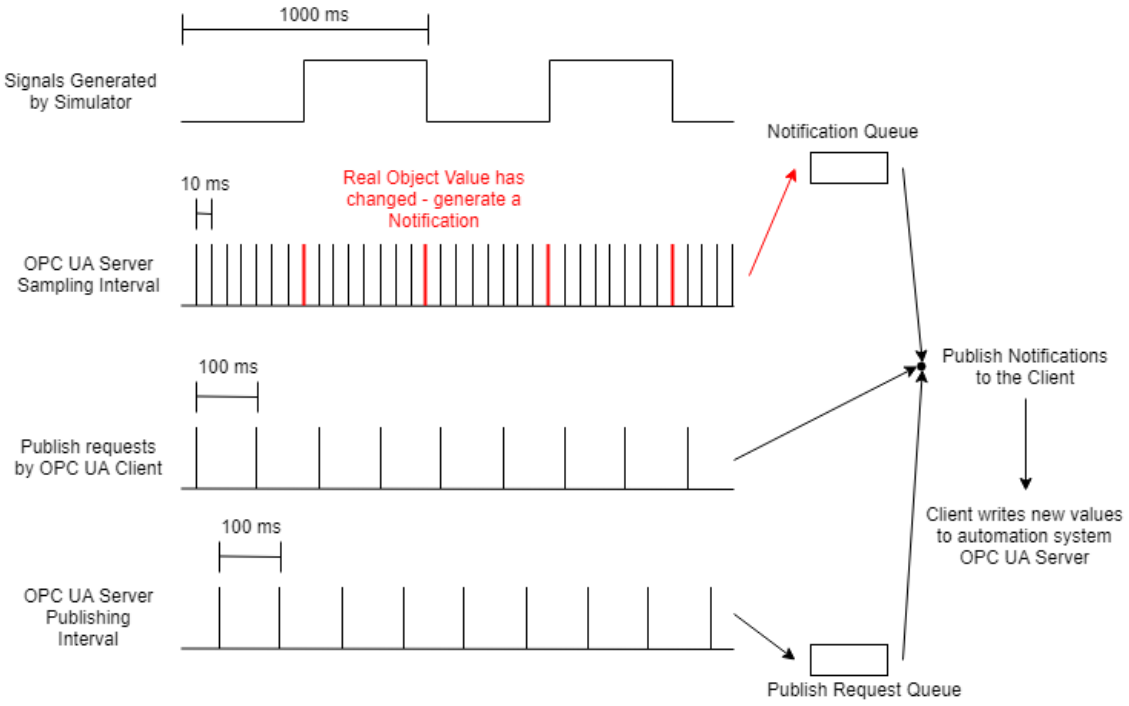

Figure 9. Simulator test setup timing diagram

The results came out positive as every single pulse out of the total 200 000 pulses ($2000\ instances \cdot 100 \frac{pulses}{instance}$) generated was detected. Reducing the pulse duration produced by the simulator below 450 ms starts to introduce an error in the number of detections. The error increases as the pulse duration approaches 400 ms – the cycle time programmed for the PACs of the automation system.

"Max. Subscriptions per Session", "Max MonitoredItem per Subscription" and "Max Request Item Count" parameters were changed to increase the number of created Subscriptions to 4, with 500 MonitoredItems per Subscription. This alteration had no significant impact on the performance. Therefore, the requirement for the values of these parameters is for the product of "Max. MonitoredItem per Subscription" (Which is equal to "Max Request item Count") and "Max. Subscriptions per Session" to be greater than the number of desired tags in the OPC UA Server.

The test system was extended to 3 PACs with still 2000 signals per PAC totalling in 6000 bridges. The results were unaffected by this adjustment. Since the simulator is connected to system bus through one Ethernet port, the network speeds were observed during the

test - each PAC used 2-3 Mbps of network bandwidth. The sum was increasing linearly with the number of PACs. Lowest speed limitations are set in the firewalls at 300 Mbps. Simple calculations show that the system could be scaled up to 100 PACs which is much more than used in any of the real projects and thus can be the confirmation for simulator's scalability.

6 Simulator configuration steps

The number of simulated devices can be in the hundreds, requiring thousands of signals to be bridged between OPC UA Servers and cannot be configured manually. The purpose of this chapter is to propose structured steps to set up the simulator for an existing automation system and provide insights for automating this process using configuration files available on the existing system.

Requirements for configuring a simulator for an existing project:

1. Simulator project
    a. I/O signal tags
    b. Simulation logic
2. OPC UA Servers on automation system PACs and simulator PACs
3. Bridge signals between OPC UA Servers

I/O signal tags are available on the existing project and should be exported from there. Simulation logic can be implemented by creating UDFB templates for each type of simulated device. I/O signal tags can then be connected to instantiate the programs for a single device, similarly to the control template depicted on Figure 2. It was stated in chapter 4.4 that the number of virtual PACs on the simulator computer is not limited. Pairing every automation system PAC with a single virtual PAC in simulator computer by means of having same signal tags and devices simplifies OPC UA Server setup significantly. This means that the OPC UA Servers on these PACs will be identical, requiring only one configuration for a pair of PACs (Automation system PAC and simulator PAC). Every one of these requirements can be met by using a copy of existing automation system project as the basis for the simulator:

1. I/O signal tags are already defined
2. UDFB instances have already been created
3. Number of PACs is equal in both projects

Using this approach, the setup procedure can be divided into 4 steps:

1. OPC UA Server setup on existing project
2. Creating a copy of existing project to be the basis of simulator project
3. Replacing device control templates with simulation templates in the simulator project
4. Configure DataHub OPC UA Bridge between the two projects

To configure the simulator, 3 XML files have to be exported from the existing project.

1. I/O channel configuration data, that has tag names and types for all physical signals to field devices.
2. Serial I/O configuration data, that includes all tag names and types for signals transmitted over serial communication channels (E.g. Modbus, Profibus, NMEA, etc.)
3. UDFB configuration data, consisting of UDFB instances of all control programs that can be used to select desired devices that need to be simulated and configure a DataHub OPC UA Bridge for the signals.

First two files will be used to set up OPC UA Servers.

In first step, OPC UA Servers are configured for all PACs. Required parameters, their values with source comments are presented in Table 4.

Table 4. OPC UA Server parameter values

Server parameter	**Value**	**Source / Comment**
Max. sessions	1	One bridge to all Servers
Max. Subscriptions per Session	1	All MonitoredItems in one Subscription
Max. MonitoredItem per Subscription	Calc.	Calculated from the number of total tags added to the Server
Max. PublishRequest per Session	1	Equal to the number of created Subscriptions
Max. DataChangedValue per MonitoredItem	1	Keep only the newest value in Notification queue

All I/O channel and serial I/O signal tags use complex L3_BI, L3_AI, L3_BO or L3_AO data types used to communicate both the value and signal state. The corresponding IEC 61131-3 value type is BOOL for binary, REAL for analog signals and UINT for signal

state. Since Straton IEC OPC UA Servers do not support complex data types, the value and state have to be included as separate tags.

OPC UA Server Nodes are described by 4 parameters:
NAME - variable name in IEC database (Real Object)
TAG – Node ID in the OPC UA Server
MODE – determines access rights for the client (Read, Write, Read/Write)
TYPE – data type used for Node Value Attribute
Using the ID and channel type elements in I/O and serial I/O signal export files, values for all parameters of the nodes can be defined. The module_ID element is used to get PAC number that owns the signal to generate separate import file for each PAC.

After creating a Node for every signal value and state, a copy of the project can be made to be used as a basis for the simulator project. This ensures that all signals are already defined for the project and included in the OPC UA Servers for which the IP address is the only change to be made.

Every control program in Valmatic is created as an UDFB that is instantiated for a single device by attaching corresponding I/O signals. Since the copied project includes instances for all devices, simply changing the UDFB library of control templates to simulation templates is the simplest way to configure simulation logic to the project. The drawback of this approach is that the structure of the simulation template must be identical to the control template, limiting some of the potential features.

Final step is to bridge the data between automation system OPC UA Servers and simulator OPC UA Servers. Cogent DataHub offers the possibility of loading user-defined configuration files on startup. Bridge function is used to define a bridge between two nodes in different OPC UA Servers. The syntax is described as follows:
(*bridge source destination flags multiply add srcmin srcmax dstmin dstmax*)
bridge – function identifier
source – starting point node address identifier
destination – ending point node address identifier

flags – bit-coded parameter

1 – Forward bridge from source to destination

2 – Inverse bridge from destination to source

16 – Clamp output to the minimum

32 – Clamp output to the maximum

256 – Nodes are copied without transformation

512 – Apply linear transformation to nodes

1024 – Map node values to a range

4096 – Bridge is disabled

Bits for 256, 512 and 2014 are mutually exclusive.

multiply – Multiplier value for linear transformation

add – Adder value for linear transformation

srcmin – Minimum range map value for source node

srcmax – Maximum range map value for source node

dstmin – Minimum range map value for destination node

dstmax – Maximum range map value for destination node

The nodes between automation system OPC UA Server and simulator OPC UA Server are bridged without transformation meaning the *flag* value is equal to 257. An example of a bridge function instance could look like this:
(bridge “SIMPAC01:PAC01.InputSignals/K1442_InputCH.Value”
“REALPAC01:PAC01.InputSignals/K1442_InputCH.Value” 257 1 0 0 0 0 0)

Exporting all UDFB instances as an XML file enables a filter (e.g. automatically controlled pumps) to be applied and extract signals that are used for desired devices. These signal names, along with the PAC number that owns them can then be combined to create the OPC UA bridge configuration file.

7 Simulation of oily bilge system

This chapter serves as the proof of concept for the research and development done in the preceding parts of the thesis. Oily bilge system is configured to serve as a part of the existing automation system to be simulated. Previously proposed configuration steps are followed to identify possible shortcomings in this approach.

7.1 System description

The main bilge system is arranged to drain any watertight compartment other than ballast, oil or water tanks and discharge the contents directly overboard. These watertight compartments are collectively referred to as bilge wells that are connected to the main bilge line via an automatically controlled valve. Wells that collect bilge water from rooms with operating machinery cannot be emptied directly overboard. The contents of these wells are contaminated with oil and other chemicals and are pumped to an oily water separator before being discharged overboard or collected to specific tanks and handed over to suitable facilities when in port. Discharging oil from a ship is harshly regulated by national treaties and will result in large fines for both the crew and company [20]. Normally bilge pumps are redundant to enable stand-by operation if one of the pumps fails. There is a valve between bilge pumps and the main line as well as between the settling tank and pumps. Every bilge well is equipped with three binary level sensors. Level control high (LCH) signal commands the opening of corresponding valves and starting of one of the pumps. Level control low (LCL) signal commands the pump to be stopped and valves closed. In case the water level in a bilge well rises above LCH limit, level alarm high high (LAHH) alarm is triggered in the IAMCS. In case of an emergency condition, even oily bilge water can be discharged directly overboard.

The oily bilge system used as an example is reduced to 9 bilge wells with a valve for each, one pump, one settling tank and a valve between the latter two. This results in a total of 89 signals divided between two PACs. The system is illustrated on the user-interface presented on Figure 10.

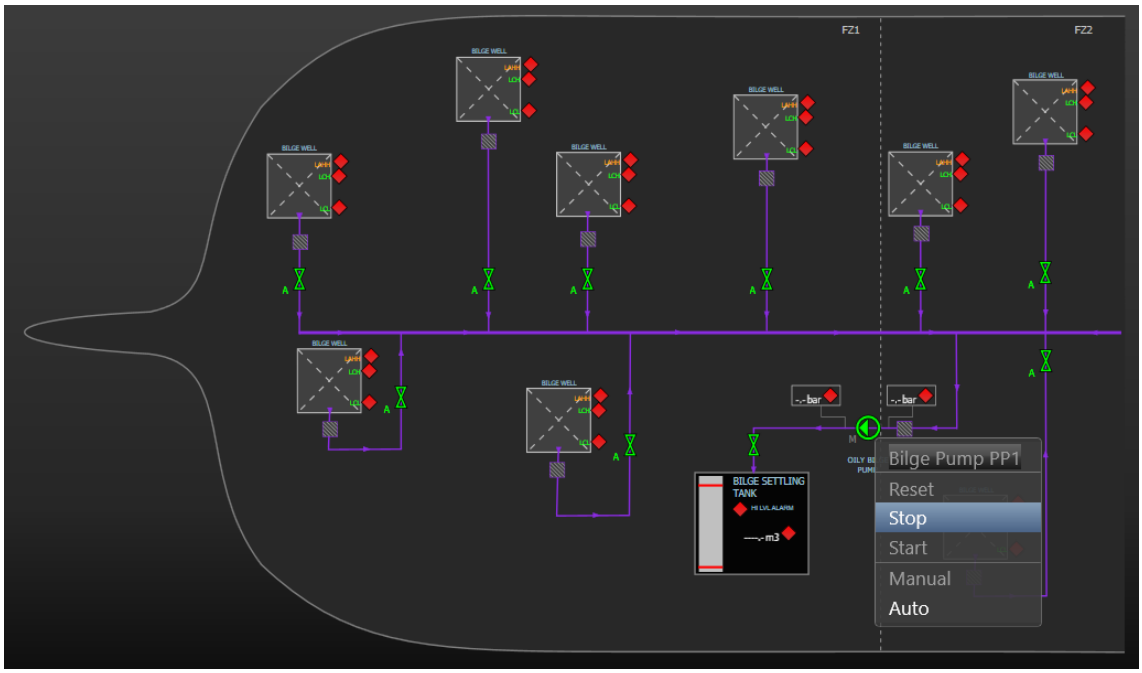

Figure 10. Oily bilge system HMI

7.2 Simulator setup

As the first step, all I/O signals were exported from the existing project. The XML structure is presented on Figure 11. Following the instructions stated in chapter 6, a Python script was developed to generate an XML import file for the OPC UA Server and a configuration file for DataHub OPC UA Bridge.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<IoChDescription>
   <IoCh SFB_TYPE            = "Input/Output Channel"
         PLC_NO              = "2"
         INTCHANNEL          = "0"
         ID                  = "BO_FZ1"
         YARD_ID             = ""
         CRUPD_NAME          = "true"
         CRUPD_REM           = "true"
         CRUPD_YARD          = "true"
         REF_CNT             = "8"
         MACH_GRP            = "0"
         NAME_1              = "Blackout Firezone 1"
         CH_TYPE             = "DI"
         CH_SPEC             = "CC=Alm/Evt  :Superv"
         AO_SUPERV           = "false"
         MODULE_ID           = "010103"
         MODULETYPE          = "VM-BI.16-SUP"
        ...
```

Figure 11. Part of the IO channels export XML file

Python programming language version 3.7 was selected for automating parts of the simulator configuration [21]. Python combines the power of general-purpose programming languages with the ease of use of domain-specific scripting languages like MATLAB. It has vast number of libraries for data processing and enables direct interaction with the code, using a terminal [22]. Python 3.7 comes with built-in XML handling submodules such as the ElementTree XML processor that is extended by lxml library [23]. The latter has two big advantages over other available XML processing tools. First is the ease of programming, since it is based on the ElementTree package, that was developed to simplify and streamline XML processing. The lxml package also boosts the performance, improving the experience of working with large xml files [24].

A single Python script was developed to parse the data from XML file of exported IO list from the existing project and generate import files for both Straton OPC UA Servers and DataHub OPC UA Bridge configuration. Module IDs consist of 6 digits and are used to identify the signal location in the system. First two digits represent the PAC number, middle two digits show the FIC number and final two digits are for the I/O module. The script iterates through the signals in the IO lists and counts the number of PACs used in the project. This number is used to create a loop that generates an XML file for each PAC.

For each PAC, the IO list is iterated one more time to count the number of signals handled by the controller. This value is used for the “Max. MonitoredItem” parameter in Server configuration.

The XML structure consists of 6 levels of elements, that are described in Table 5.

Table 5. XML structure of OPC UA Server configuration file

Level	**Name**	**Attributes / Comments**
1 (Root)	K5project	Version
2	Fieldbus	-
3	K5BusOPCUA_s	Attributes describing the OPC UA Server, including the ones in Table 4.
4	Fieldbusmaster	IP address and port number for the Server.
5	Fieldbusslave	Used to divide Server Nodes into logical groups.
6	Fieldbusvar	A Node in the server, described by the attributes presented in Chapter 6.

Nodes are split into two categories at the fifth, fieldbusslave level – input and output signals. This is not necessary but improves the manual management of nodes when the need arises.

For bridge configuration, the values of output signals were forwarded from automation system server to simulator server. Input signal values, states as well as readback state of output signals were bridged from the simulator to the automation system. After the Servers were configured on existing project, it was copied to be the basis for the simulator. The IP addresses were changed manually.

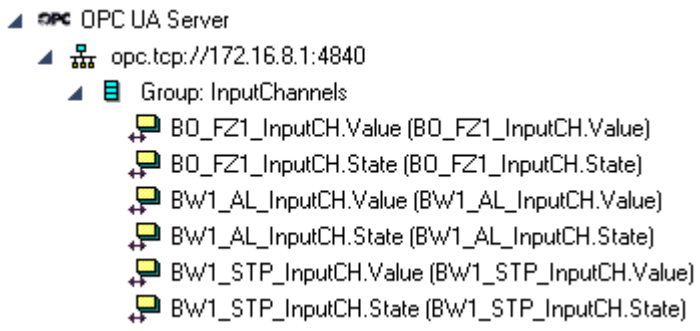

Figure 12. OPC UA Server on simulator PAC01

Control templates were swapped with simulation templates from another library. Creating the latter with identical name and structure as the control templates ensures that the project can be generated without making any additional changes. This is explained in more detail in the next subchapter.

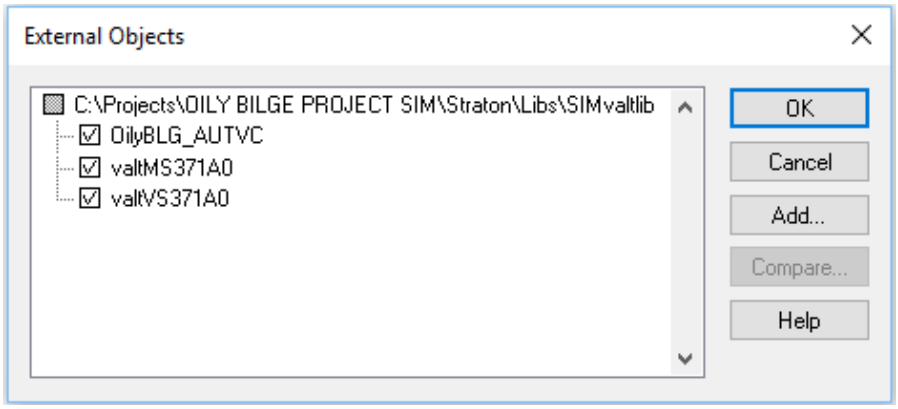

Figure 13. Simulation templates from new library

A Cogent DataHub Client was configured for each Server, seen on Figure 14.

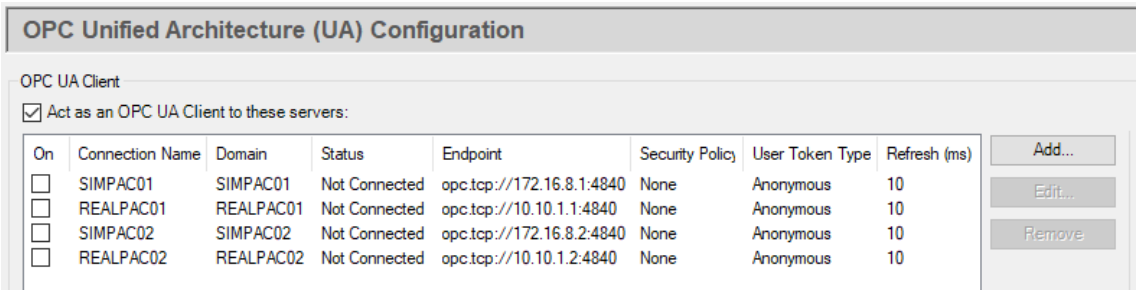

Figure 14. OPC UA Clients in Cogent DataHub

Bridge configuration file generated by the Python script is loaded on every startup of Cogent DataHub.

7.2.1 Simulation templates

Example oily bilge system uses two kinds of automatically controlled devices – a single speed pump and single-acting valves. The pump is controlled by 5 input and 3 output signals that are described in Table 6.

Table 6. Single speed pump control signals

Signal	**Type**	**Function**
Remote	In	Manual switch in device control cabinet, used to select between local and remote modes.
Running	In	Indication of motor running state.
Blackout	In	Indication of supply power failure for the device.
Fail	In	Minor failure in the device, control functionalities are retained.
Tripped	In	Major failure in the device, motor has stopped and cannot be started.
Start	Out	Start command for the device.
Stop	Out	Stop command for the device.
Reset	Out	Command to reset failed state alarms after they have been resolved.

Having identical structure for control and simulation templates is limiting the number of simulation features but is suitable for scenarios where the device can be simulated only by input and output signals and without human interaction. To add the feature for different simulation scenarios the HMI configuration is changed. This enables giving commands for the device from simulator HMI and is illustrated on Figure 15. Buttons were added to the user interface to activate blackout signals.

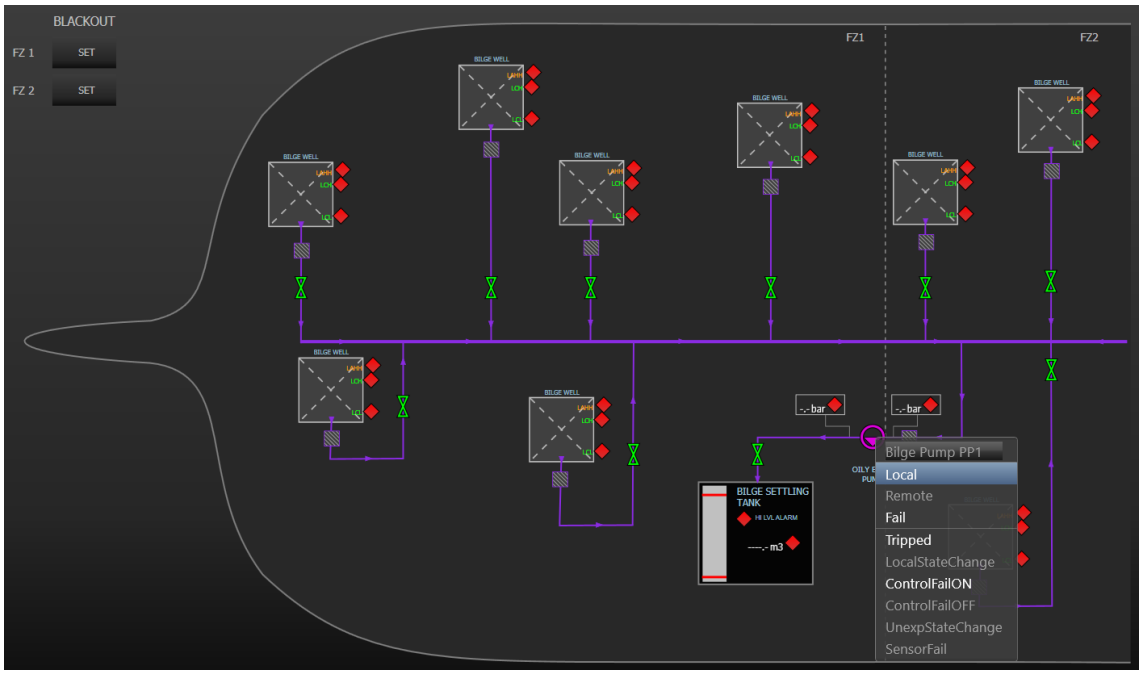

Figure 15. Simulator HMI

Simulation templates for the pump, bilge wells and bilge settling tank are included in Appendix 2. Template for single-acting valve follows closely the pump template and is not included.

In addition to feedback activation according to output signals, pump simulation template has a total of seven supplementary user-triggered functions.

Local and Remote commands are used to select how the device is controlled by toggling the remote input signal of the pump. A device in remote control mode can be controlled by either operator commands from the HMI or commands from auto logic. Locally controlled pump is switched on and off from a local motor control panel. “LocalStateChange” command is used to toggle running feedback signal if the device is in local control mode.

Fail signal is used to indicate that the device is experiencing a minor failure, maintaining the possibility of control, but informing the operator with an alarm. After the failure has been fixed, the alarm is rectified by the “Reset” command seen on Figure 10.

Tripped signal is used to indicate that the pump is experiencing a major failure and has stopped running. Control option is restored after the failure has been fixed and alarm reset by the operator.

“UnexpStateChange” command is used to simulate feedback fault and toggle the signal if the device is in remote control mode. The device should change state only if another signal confirms a reason for the feedback fault, e.g. blackout or tripped. Operator is informed of this state by an alarm.

It was noted earlier that for each I/O signal in Valmatic system both the value and state are used. In case of a faulty signal, the latter informs the operator of the possible source for failure, e.g. sensor failure, IO module failure, earth fault etc. A faulty signal should be ignored by the system. “SensorFail” command is used to manipulate the signal states.

Blackout signal indicates that the power supply to the device is flawed. This can be caused by a tripped circuit breaker or stopped generator. All working devices will enter their normal state (stopped for motors) with indication that the state change was caused by a blackout. After power supply has been restored, devices are restarted automatically, if configured to do so by the operator. In the example oily bilge project, the devices are divided into two fire zones, for both of which a blackout can be simulated.

Bilge well valve auto control template was replaced by well volume simulation template. Well volume is ramped up if the valve is closed or pump stopped. Upon reaching 85% capacity, high level signal is activated. Volume is ramped down if the valve is opened and pump running. Low level signal is activated if the value has dropped down to 5%. Valve identifier index is used to set a different filling rate for every well.

Final simulation program was created for bilge settling tank. Volume is ramped up if at least one bilge well valve is opened, settling tank valve is opened and the pump is running. Volume is ramped down if any of these conditions is not met. Reaching 95% capacity, high alarm is activated that informs the system to stop filling the settling tank.

7.3 Conclusion and shortcomings

Oily bilge system was simulated for three scenarios – normal operation with bilge well valves and bilge pump in auto modes, unexpected close of bilge settling tank valve and finally blackout restore for both fire zones. All tests worked as expected without notable peculiarities in system behaviour. This confirms that the automatic generation of OPC UA Servers and bridges work correctly. I/O signal tags are connected to right UDFBs and properly bridged between Servers.

Real-time human interaction with the simulation was not stated as a requirement for the simulator but was added as an extra feature to the solution. This enables engineers to simulate different scenarios and run more sophisticated tests by adding only one additional step to the configuration process.

The requirement of being easily configurable by all engineers working on control software is a rather subjective as everyone's level of experience is different. When designing the configuration steps, no assumptions were made for user knowledge apart from being familiar with the set of standard tools used in Valmarine. OPC UA Server and Client parameters are selected by the setup automation script, leaving the user only with the task of importing the generated files.

Currently, the most time-consuming task in the configuration process is creating UDFB templates for simulation programs. As with control templates they can be standardized for most of the devices.

Although the simulator meets all the requirements, there exist ways to improve the solution.

Using Skkynet's Cogent DataHub OPC UA Bridge introduces an issue of third-party dependence on further development of the product as well as fixing potential bugs in the software. This can be solved in the future if COPA-DATA develops OPC UA Client support for Straton IEC, this would also simplify the simulator configuration process.

Simulator user-interface inputs are limited to 8 for each device and all devices using the same template will have same commands. Additionally, some scenarios take 2 slots to indicate active selection, e.g. switching control fail ON and OFF. The list of available

scenarios should be reduced to 8 most important ones and indication for active situation developed without needing two separate HMI inputs.

The networking solution of using two firewalls and NAT can be improved in two ways. First is to implement NAT in software. Second is to establish a way to deviate from standard Valmatic IP addresses without changing them manually every time after new software is downloaded and still preserving all functionalities of a working system. This would reduce the amount of additional hardware required for simulator setup.

Finally, the feature of simulating with redundant PACs was not analysed. Current solution uses single PAC as the source of control signal values. Cogent DataHub offers a way to configure redundant OPC UA Servers and switch between them if data quality from one is bad.

8 Summary

There are two distinct ways for simulating field devices in Valmatic factory acceptance tests – hardware electronic circuits and including simulation software to the project itself. Former brings the benefit of keeping the IAMCS truthful to the system that will eventually be used on the vessel, but is not suitable for comprehensive system testing, where the number of involved devices is large. Software simulations are scalable and easily configurable but alter the IAMCS significantly.

The objective of this thesis was to capture the advantages of both approaches and solve their limitations. The requirement was to develop an externally connected SIL simulator for the Valmatic automation system, that meets the scalability requirements and is easily configurable by all engineers involved in the project.

Implementing the simulator in software requires the communication of I/O signals between the IAMCS and simulator. General adoption in the industrial automation field to ensure sustainability of the solution and assessment of performance were main factors in the communications protocol selection process. OPC UA is widely acknowledged by the process and automation fields and accepted as International Electrotechnical Commission standard IEC 62541. Additionally, the performance of OPC UA publish-subscribe model proved to be superior over other general use protocols, such as Modbus.

Virtual implementation of Valmatic system was selected as the simulator platform for one major argument – Every engineer is familiar with IEC 61131-3 programming languages reducing the adoption time significantly. These languages also meet the requirements for device model complexity, which mostly consists of Boolean algebra, timed delays and linear ramping of analog values. The simulation logic is executed by virtual machine controllers on a single simulator computer. This choice came with the downside of Valmatic not supporting the configuration of OPC UA clients. OPC UA servers are configured on both the IAMCS and simulator and Skkynet Cogent DataHub OPC UA Bridge is used to communicate data between them. Valmatic uses fixed IP addresses for its controllers, requiring NAT to be used for connecting the two networks.

The simulator was tested for performance and scalability requirements – being able to detect fixed length input and output signal pulses with 2000 signals per controller and up to 40 controllers in the system. Configuration parameters that describe the OPC UA communication model were theoretically analysed and values for optimal performance were suggested. The simulator passed tests for both requirements.

The number of simulated devices can be in the hundreds, requiring thousands of signals to be bridged between OPC UA servers and cannot be configured manually. A set of structured steps for setting up the simulator was proposed, without making any assumptions for the user knowledge, apart from being familiar with the set of standard tools used in Valmarine. OPC UA server and bridge configurations and parameter values were generated automatically by a Python script, leaving the user only with the task of importing the produced files.

Finally, feasibility of the solution was confirmed by setting up the simulator for an existing oily bilge collection control system. The result worked as expected, without any peculiarities in system behaviour, confirming the correctness of automatically generated configuration. Real-time human interaction with the simulation was added as an extra feature to enable engineers run more sophisticated tests. This was done by connecting a virtual HMI to the Valmatic system running on the simulator computer.

Although the solution meets all the requirements, it is still the first attempt to solve the problems of current simulation methods and ways for improvement exist. Future work involves removing DataHub OPC UA Bridge from the solution once OPC UA client functionality has been developed for Valmatic. NAT should be implemented in software instead of hardware firewalls, as in current solution. Finally, options for running simulations on redundant controllers should be studied.

References

[1] A. Kossiakoff, W. N. Sweet, S. J. Seymour and S. M. Biemer, Systems Engineering Principles And Practice, Hoboken, New Jersey: John Wiley & Sons, 2011.

[2] W. H. Kwon and S.-G. Choi, "Real-Time Distributed Software-In-The-Loop Simulation for Distributed Control Systems," in *International Symposium on Computer Aided Control System Design*, Kohala Coast, HI, USA, 1999.

[3] T. Teodorowicz, "Comparison of SCADA protocols and implementation of IEC 104 and MQTT in MOSAIK," University of Münster, Münster, 2017.

[4] J. Makhija, "Comparison of protocols used in remote monitoring: DNP 3.0, IEC 870-5-101 & Modbus," IIT Bombay, Mumbai, 2003.

[5] Modbus-IDA, 24 October 2006. [Online]. Available: http://www.modbus.org/docs/ Modbus_Messaging_Implementation_Guide_V1_0b.pdf. [Accessed 26 February 2019].

[6] G. Clarke and D. Reynders, Practical Modern SCADA Protocols: DNP3, 60870.5 and Related Systems, Oxford: Newnes, 2004.

[7] C. Johnson, "Securing the Participation of Safety-Critical SCADA Systems in the Industrial Internet of Things," in *International Conference on System Safety and Cyber Security*, London, 2016.

[8] M. H. Schwarz and J. Börcsök, "A Survey on OPC and OPC-UA: About the standard, developments and investigations," in *XXIV International Conference on Information, Communication and Automation Technologies (ICAT)*, Sarajevo, 2013.

[9] S. Cavalieri and F. Chiacchio, "Analysis of OPC UA performances," *Computer Standards & Interfaces,* vol. 36, no. 1, pp. 165-177, 2013.

[10] H. Elfaham, F. Palm, S. Grüner and U. Epple, "Full Integration of MATLAB/Simulink with Control Application Development using OPC Unified Architecture," in *2016 IEEE 4th International Conference on Industrial Informatics*, Poitiers, 2016.

[11] "MathWorks MATLAB," The MathWorks, Inc., 2019. [Online]. Available: https://uk.mathworks.com/products/matlab.html. [Accessed 10. 02. 2019].

[12] "National Instruments LabView," National Instruments, 2019. [Online]. Available: http://www.ni.com/en-us/shop/labview/labview-details.html. [Accessed 11. 02. 2019].

[13] "Inductive Automation Ignition," Inductive Automation, 2019. [Online]. Available: https://inductiveautomation.com/ignition/. [Accessed 11. 02. 2019].

[14] "Cogent DataHub," Skkynet, 2019. [Online]. Available: https://cogentdatahub.com/. [Accessed 15. 02. 2019].

[15] Moxa Inc., "Industrial Secure Router User's Manual," Author, Taipei, 2018.

[16] OPC Foundation, “OPC Unified Architecture Specification Part 3: Address Space Model Release 1.04,” Author, 2017.

[17] OPC Foundation, “OPC Unified Architecture Specification Part 4: Services Release 1.04,” Author, 2017.

[18] OPC Foundation, “OPC Unified Architecture Specification Part 1: Overview and Concepts,” Author, 2017.

[19] OPC Foundation, “OPC Unified Architecture Specification part 2: Security Model Release 1.04,” Author, 2018.

[20] D. A. Taylor, Introduction to Marine Engineering 2nd Edition, Oxford: Elsevier Butterworth-Heinemann, 1996.

[21] Python Software Foundation, [Online]. Available: http://www.python.org. [Accessed 26. 03. 2019].

[22] A. C. Müller and S. Guido, Introduction to Machine Learning with Python, Sebastopol, CA: O'Reilly Media, Inc., 2017.

[23] lxml development team, “lxml - XML and HTML with Python,” 2019. [Online]. Available: https://lxml.de/. [Accessed 28 03 2019].

[24] J. W. Shipman, Python XML processing with lxml, Socorro, NMEX: New Mexico Tech, 2013.

Appendix 1 – Python script for simulator automatic configuration

```
import lxml.etree as ET

Type_Dict = {'DI': {'Value': {'Var':'_InputCH.Value', 'Type':'BOOL'},
                    'State': {'Var':'_InputCH.State', 'Type':'UINT16'}},
             'DO': {'Value': {'Var':'_OutputCH.BoCmdValue', 'Type':'UINT16'},
                    'State': {'Var':'_OutputCH.BoReadbackState',
                              'Type':'UINT16'}},
             'AI': {'Value': {'Var':'_InputCH.Value', 'Type':'FLOAT32'},
                    'State': {'Var':'_InputCH.State', 'Type':'UINT16'}},
             'AO': {'Value': {'Var':'_OutputCH.Value', 'Type':'FLOAT32'},
                    'State': {'Var':'_OutputCH.AoReadbackState',
                              'Type':'UINT16'}}}

IO_List_tree = ET.parse("ExportIoCh.xml")
IO_List_root = IO_List_tree.getroot()
PAC_List = []

#Make a list of used PACs in imported I/O-list
for IoCh in IO_List_root:
       if IoCh.get('MODULE_ID')[:2] not in PAC_List:
              PAC_List.append(IoCh.get('MODULE_ID')[:2])

#Generate an OPC UA Server import file for each PAC
for pac_no in range(1, len(PAC_List) + 1):
       filename = "PAC0"+str(pac_no)+"_UA_Import.xml"
       signal_count = 0
       for IoCh in IO_List_root:
              if int(IoCh.get('MODULE_ID')[:2]) == pac_no:
                     signal_count += 1
       n = 1
       ip_addr = "10.10.1." + str(pac_no)
       root = ET.Element("K5project", version="1.1")
       networks = ET.SubElement(root, "networks")
       fieldbus = ET.SubElement(root, "fieldbus")
       server = ET.SubElement(fieldbus, "K5BusOPCUA_s", K5ID=str(n),
                           NAME="Straton OPC UA Server PAC0"+str(pac_no),
                           MAX_SESSION="1", MAX_SUBSCRIPTION="1",
                           MAX_MONITOREDITEM=str(signal_count),
                           MAX_PUBLISHREQUESTPERSESSION="1",
                           MAX_DATACHANGEDVALUE="1", LOGTRACE="1",
                           LOGTRACE_FILE="", PASSWORD="", USE_CERTIFICAT="1",
                           CTL="PKI/CA", SC="stratonopc.der",
                           SPK="stratonopc.pem",CRL="stratonopc.crl",__F="-1")
       n += 1
       fbmaster = ET.SubElement(server, "fieldbusmaster", K5ID=str(n),
       IP=str(ip_addr), PORT="4840", SECURITYPOLICY="1", __F="-1")
       n += 1
```

```
        fbslavein = ET.SubElement(fbmaster, "fieldbusslave", K5ID=str(n),
        NAME="InputChannels",__F="-1")
        n += 1
        fbslaveout = ET.SubElement(fbmaster, "fieldbusslave", K5ID=str(n),
        NAME="OutputChannels",__F="-1")
        n += 1
        for IoCh in IO_List_root:
              Ch_Type = IoCh.get('CH_TYPE')
              Ch_ID = IoCh.get('ID')
              Ch_Pac= int(IoCh.get('MODULE_ID')[:2])
              if (Ch_Type == 'DI' or Ch_Type == 'AI') and Ch_Pac == pac_no:
                    fbvarval = ET.SubElement(fbslavein, 'fieldbusvar',
                    K5ID = str(n),
                    NAME = Ch_ID+Type_Dict[Ch_Type]['Value']['Var'],
                    TAG = Ch_ID+Type_Dict[Ch_Type]['Value']['Var'],
                    MODE = '2',
                    TYPE = Type_Dict[Ch_Type]['Value']['Type'])
                    n += 1
                    fbvarval = ET.SubElement(fbslavein, 'fieldbusvar',
                    K5ID = str(n),
                    NAME = Ch_ID + ype_Dict[Ch_Type]['State']['Var'],
                    TAG = Ch_ID + Type_Dict[Ch_Type]['State']['Var'],
                    MODE = '2',
                    TYPE = Type_Dict[Ch_Type]['State']['Type'])
                    n += 1
              if (Ch_Type == 'DO' or Ch_Type == 'AO') and Ch_Pac == pac_no:
                    fbvarval = ET.SubElement(fbslaveout, 'fieldbusvar',
                    K5ID = str(n),
                    NAME=Ch_ID+Type_Dict[Ch_Type]['Value']['Var'],
                    TAG = Ch_ID+Type_Dict[Ch_Type]['Value']['Var'],
                    MODE = '2',
                    TYPE = Type_Dict[Ch_Type]['Value']['Type'])
                    n += 1
                    fbvarval = ET.SubElement(fbslaveout, 'fieldbusvar',
                    K5ID = str(n),
                    NAME = Ch_ID + Type_Dict[Ch_Type]['State']['Var'],
                    TAG = Ch_ID + Type_Dict[Ch_Type]['State']['Var'],
                    MODE = '2',
                    TYPE = Type_Dict[Ch_Type]['State']['Type'])
                    n += 1
        tree = ET.ElementTree(root)
        tree.write(filename, xml_declaration=True, encoding="iso-8859-1",
                  standalone=True, pretty_print=True)

#Generate Cogent DataHub OPC UA Bridge configuration file
bridge_cfg = open("Custom_OPC_UA_Bridges.cfg","w")
bridge_cfg.write(";;; Point-to-point Bridging\n")
for IoCh in IO_List_root:
        Ch_Pac = IoCh.get('MODULE_ID')[:2]
        Ch_ID = IoCh.get('ID')
        Ch_Type = IoCh.get('CH_TYPE')
```

```
if Ch_Type == 'DI' or Ch_Type == 'AI':
      bridge_cfg.write('(bridge "SIMPAC' + Ch_Pac + ':PAC' + Ch_Pac +
      '.InputChannels/' + Ch_ID + Type_Dict[Ch_Type]['Value']['Var'] +
      '"' +' "REALPAC' + Ch_Pac + ':PAC' + Ch_Pac + '.InputChannels/'
      + Ch_ID +Type_Dict[Ch_Type]['Value']['Var'] + '" 257 1 0 0 0 0
      0)\n')
      bridge_cfg.write('(bridge "SIMPAC' + Ch_Pac + ':PAC' + Ch_Pac +
      '.InputChannels/' + Ch_ID +Type_Dict[Ch_Type]['State']['Var'] +
      '"' +' "REALPAC' + Ch_Pac + ':PAC' + Ch_Pac + '.InputChannels/'
      + Ch_ID +Type_Dict[Ch_Type]['State']['Var'] + '" 257 1 0 0 0 0
      0)\n')
if Ch_Type == 'DO' or Ch_Type == 'AO':
      bridge_cfg.write('(bridge "REALPAC' + Ch_Pac + ':PAC' + Ch_Pac +
      '.OutputChannels/' + Ch_ID +Type_Dict[Ch_Type]['Value']['Var'] +
      '"' +' "SIMPAC' + Ch_Pac + ':PAC' + Ch_Pac + '.OutputChannels/'
      + Ch_ID +Type_Dict[Ch_Type]['Value']['Var'] + '" 257 1 0 0 0 0
      0)\n')
      bridge_cfg.write('(bridge "SIMPAC' + Ch_Pac + ':PAC' + Ch_Pac +
      '.OutputChannels/' + Ch_ID +Type_Dict[Ch_Type]['State']['Var'] +
      '"' +' "REALPAC' + Ch_Pac + ':PAC' + Ch_Pac + '.OutputChannels/'
      + Ch_ID +Type_Dict[Ch_Type]['State']['Var'] + '" 257 1 0 0 0 0
      0)\n')
```

Figure 16 Python script for automatic configuration

Appendix 2 – IEC 61131-3 simulation programs

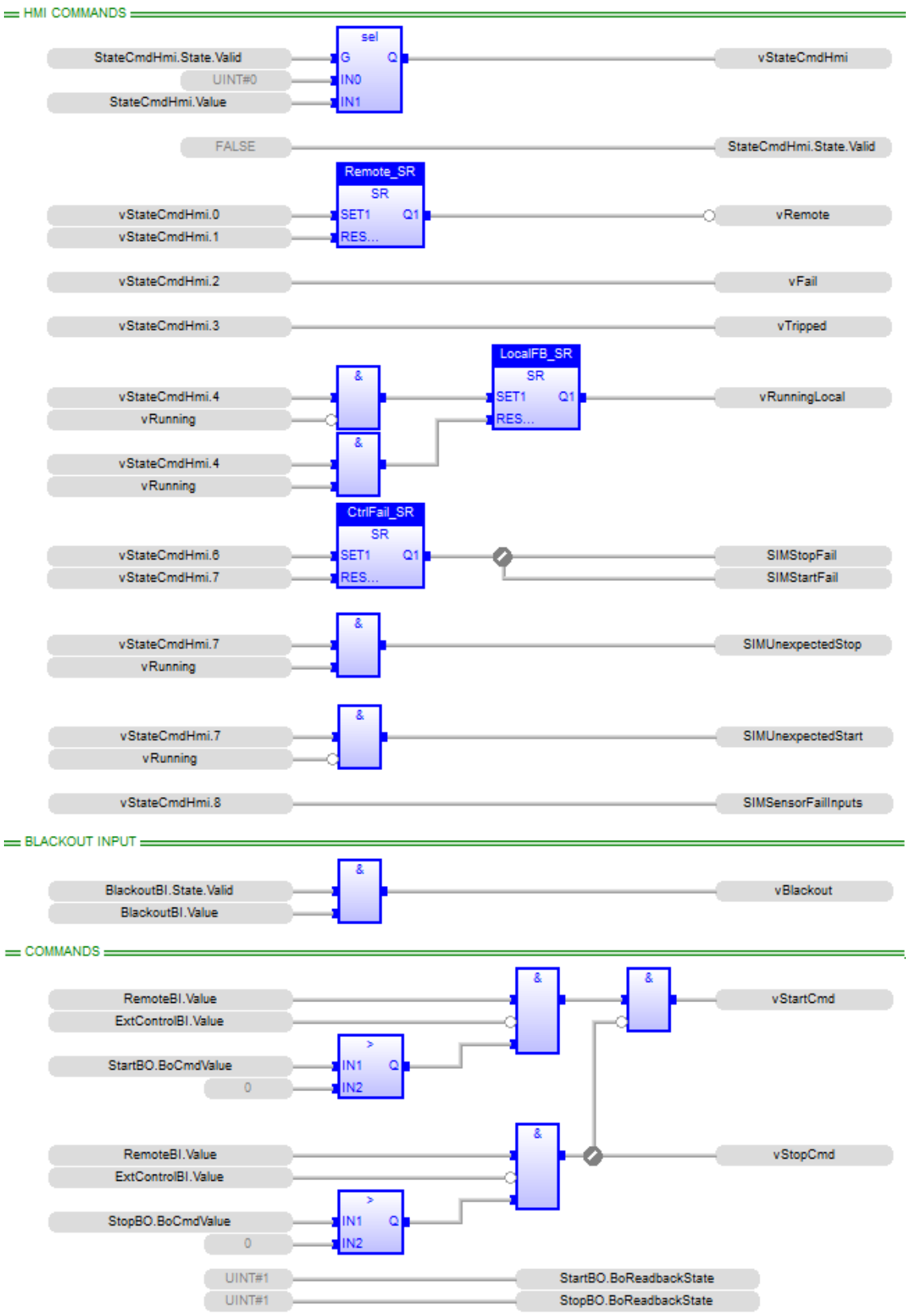

Figure 17. Pump control simulation template part 1

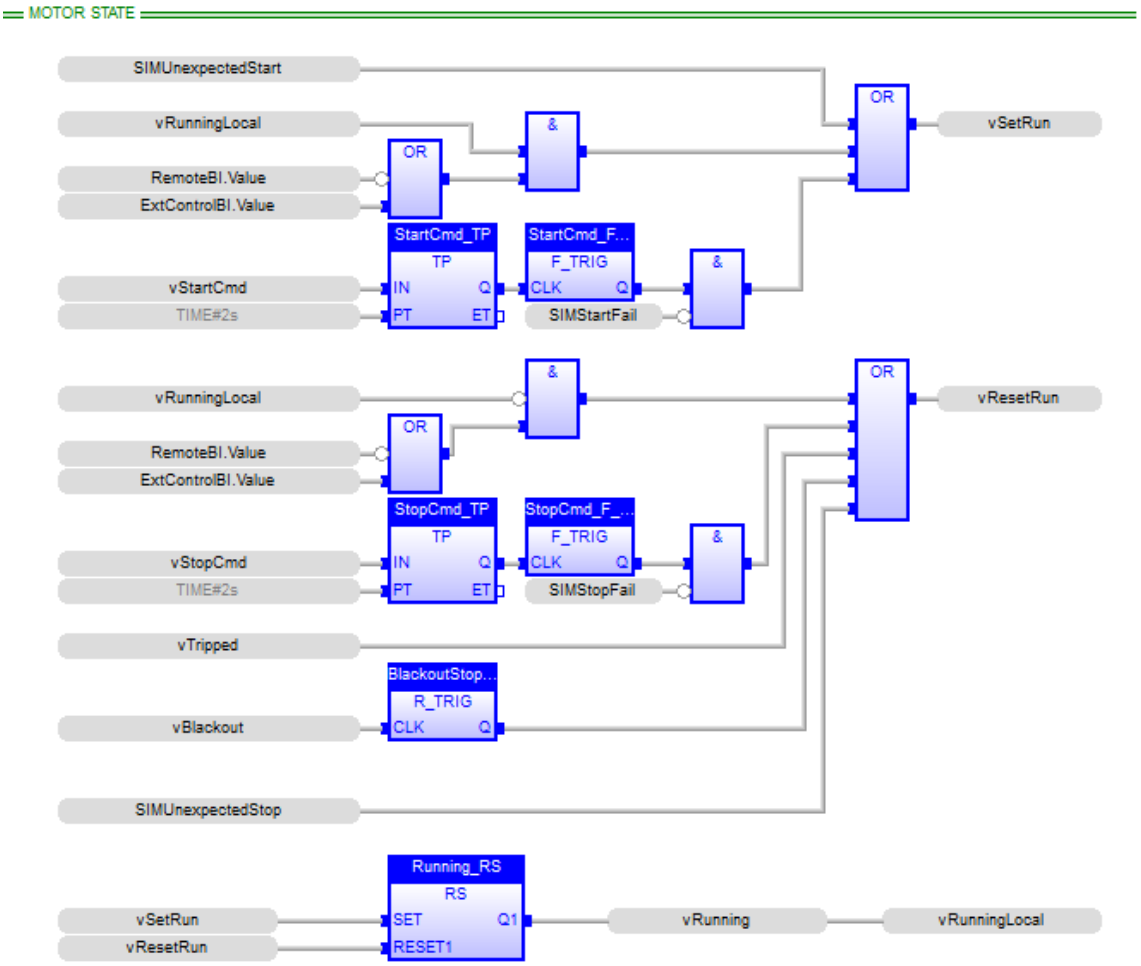

Figure 18. Pump control simulation template part 2

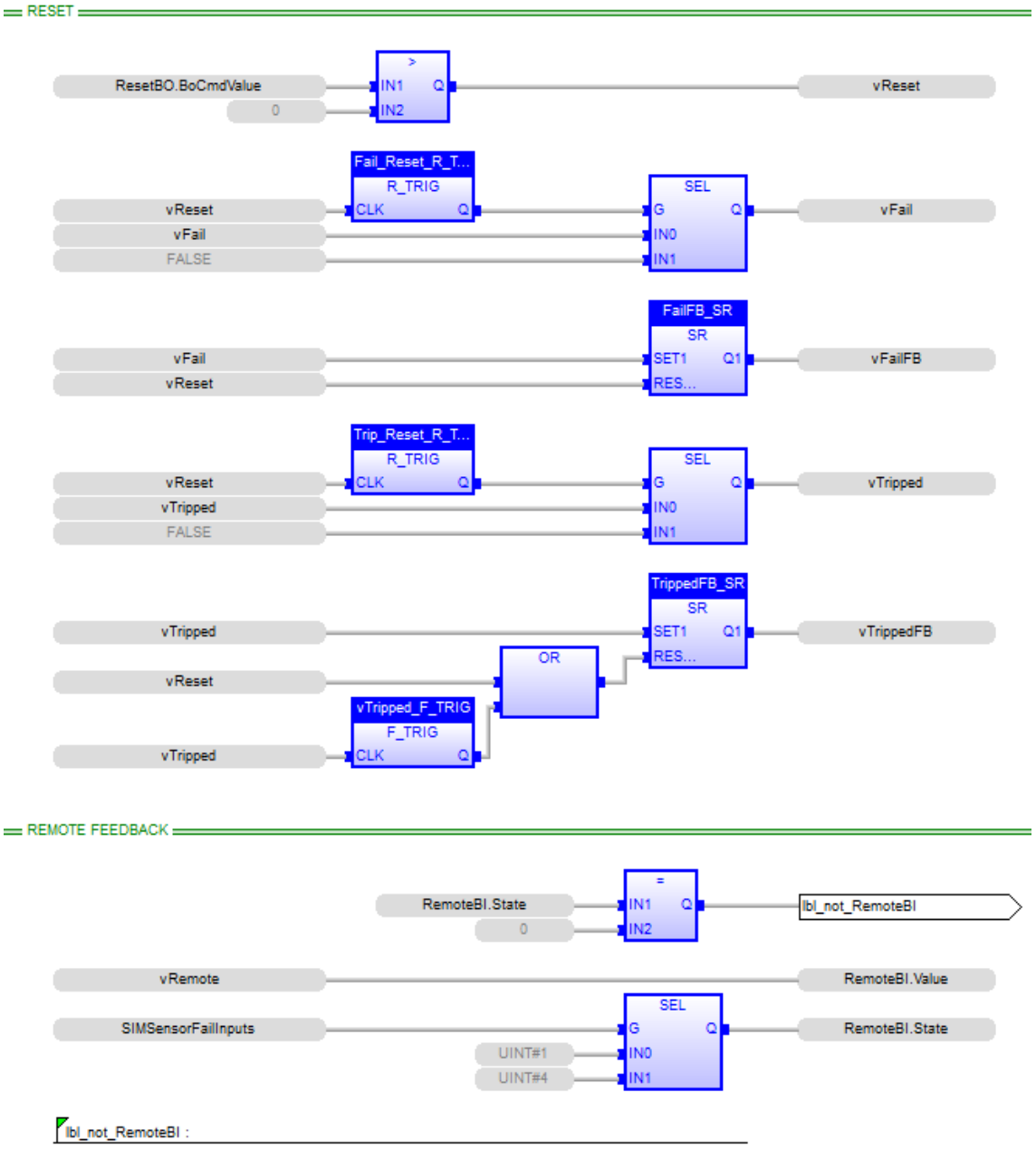

Figure 19. Pump control simulation template part 3

Figure 20. Pump control simulation template part 4

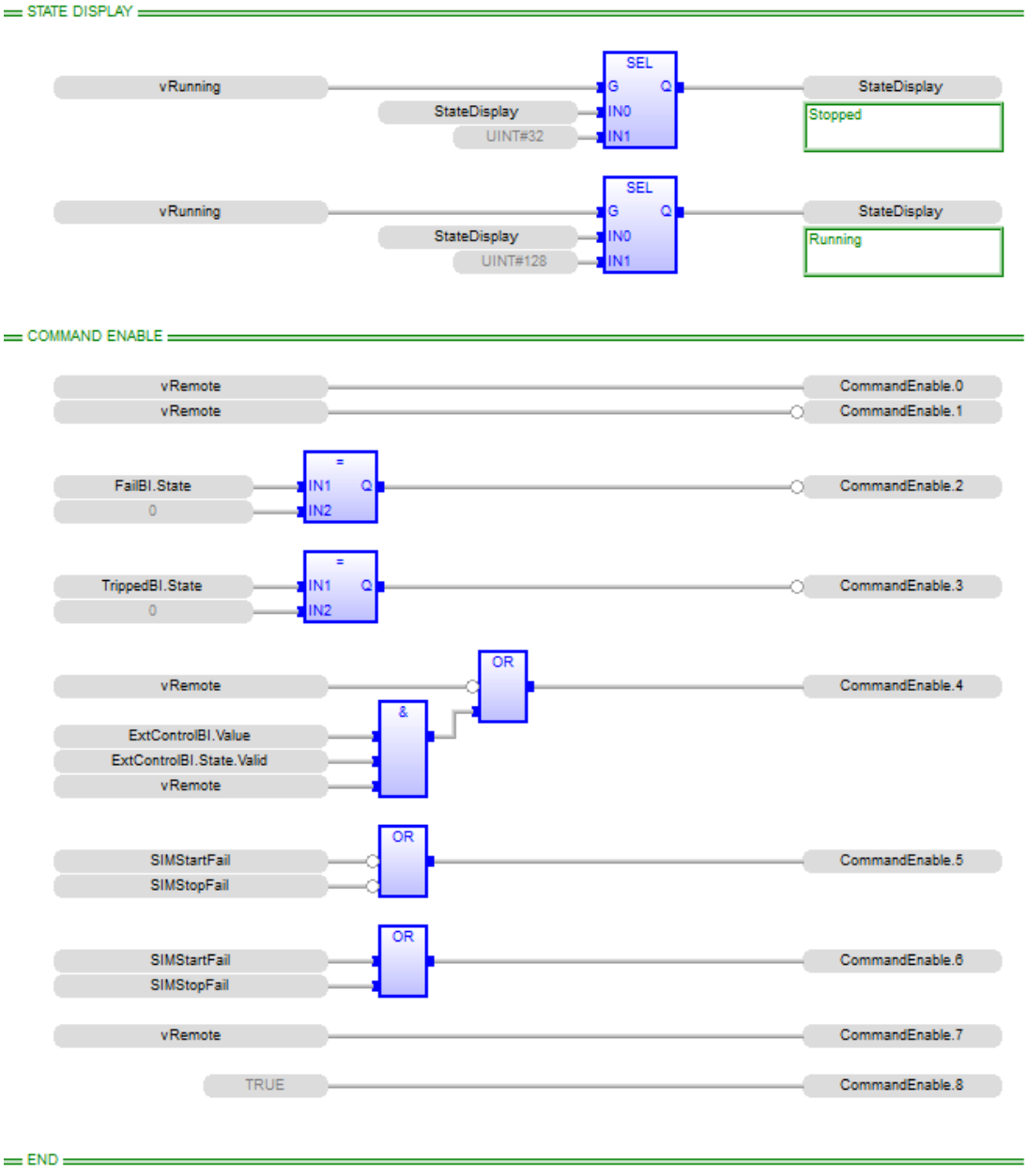

Figure 21. Pump control simulation template part 5

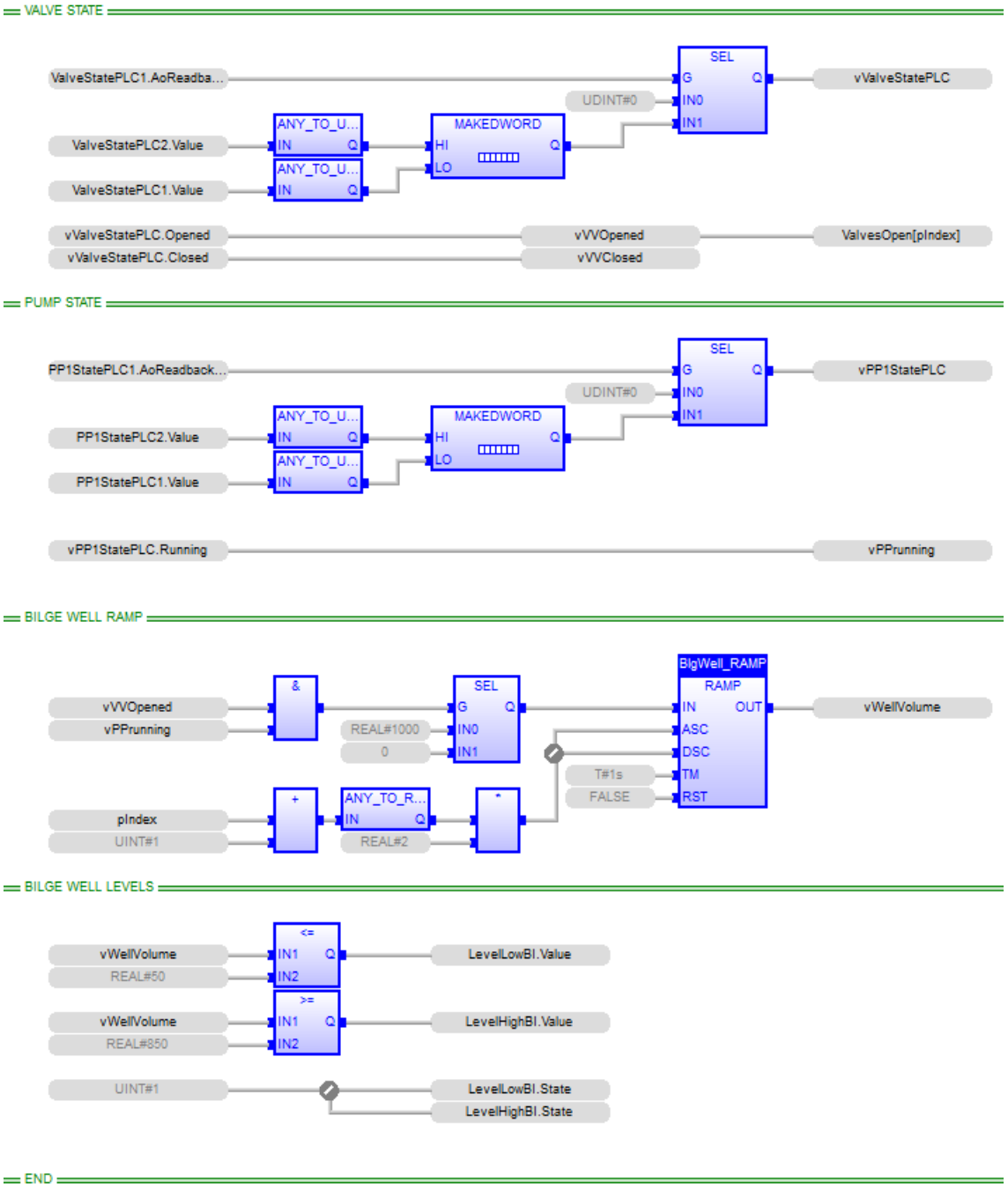

Figure 22. Bilge well simulation template

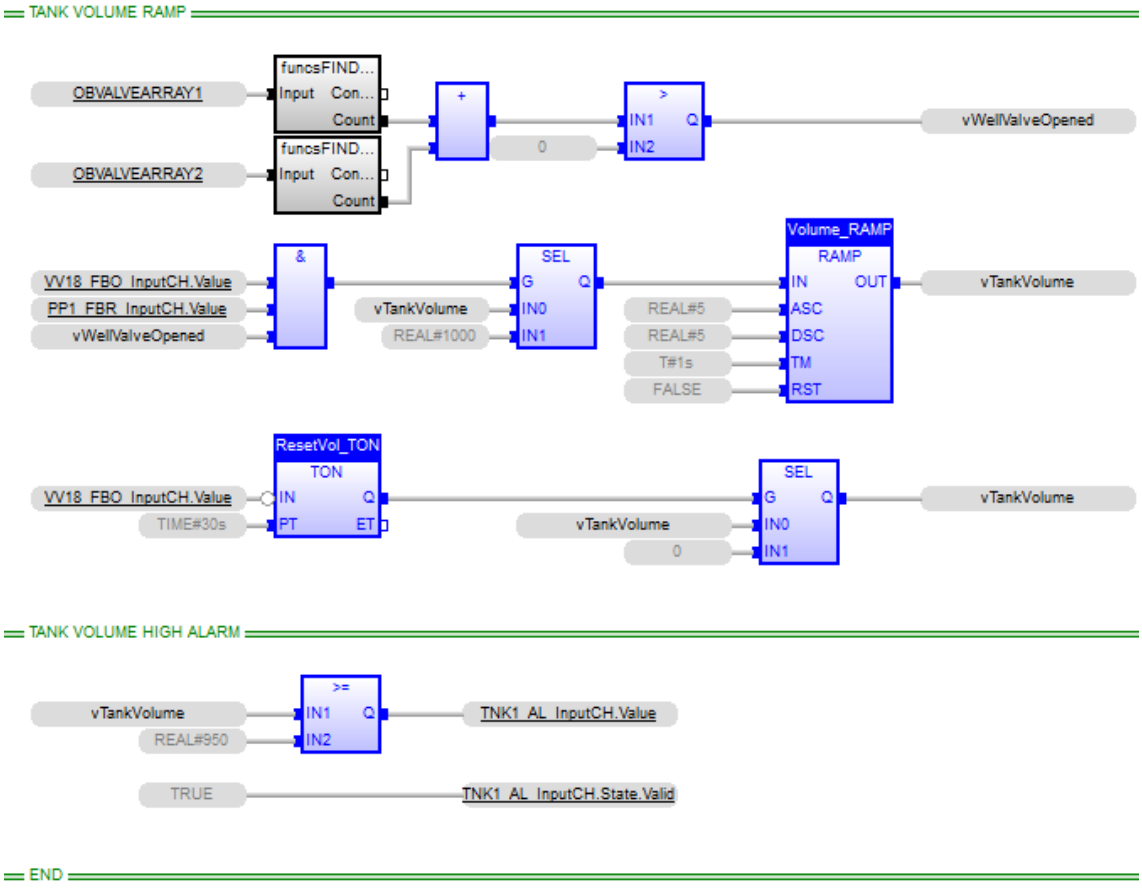

Figure 23. Bilge settling tank simulation program