TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

ILARI VENÄLÄINEN

DESIGN AND IMPLEMENTATION OF A SYSTEM STATUS VIEW OF AN AUTOMATION SYSTEM

Master of Science thesis

Examiner: Professor Kari Systä
Examiner and topic approved by the
Faculty Council of the Faculty of
Computing and Electrical Engineering
on 4th May 2016

ABSTRACT

ILARI VENÄLÄINEN: Design and implementation of a system status view of an automation system
Tampere University of Technology
Master of Science thesis, 68 pages
June 2016
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiner: professor Kari Systä
Keywords: system status view of an automation system, Qt framework, QML

This thesis covers designing and implementing a next generation of a currently used system status view of an automation system. The system status view is designed for a PC application that is based on a Qt framework and used to configure and monitor an automation system. The system status view is used mainly for monitoring an automation system, but some features for configuring the automation system are designed also. The currently used system status view is implemented with Qt Widgets, but for the next generation view, other user interface technologies provided by Qt are studied. Based on the study, another user interface technology is chosen for the implementation of the system status view.

The thesis first introduces the related automation system with an example automation system. The PC application is used to configure and monitor the example automation system. After the background is described, requirements for the implementation of the new system status view are specified, and the system view is designed based on the requirements. The design uses object oriented programming methods and UML diagrams as design tools. The design process will lead to a completed system status view that fulfils the specified requirements set for it. Last, the thesis process and further development ideas are presented.

The implemented system status view has been put to use by the customer company of Wapice Oy that ordered the work. More features than presented in this thesis for the new system status view were ordered by the customer, and the design and implementation for the features is done in the near future.

TIIVISTELMÄ

ILARI VENÄLÄINEN: Automaatiojärjestelmän tilanäkymän suunnittelu ja toteutus
Tampereen teknillinen yliopisto
Diplomityö, 68 sivua
Kesäkuu 2016
Tietotekniikan koulutusohjelma
Pääaine: Ohjelmistotuotanto
Tarkastaja: professori Kari Systä
Avainsanat: automaatiojärjestelmän tilanäkymä, Qt-sovelluskehys, QML

Tässä työssä käydään läpi automaatiojärjestelmään liittyvän tilanäkymän seuraavan sukupolven suunnittelu ja toteutus. Tilanäkymä suunnitellaan Qt-sovelluskehykseen perustuvaan PC-sovellukseen, jota käytetään automaatiojärjestelmän konfigurointiin ja monitorointiin. Tilanäkymää käytetään pääasiassa automaatiojärjestelmän monitorointiin, mutta tilanäkymään suunnitellaan myös toimintoja, joiden avulla automaatiojärjestelmää voidaan konfiguroida. Olemassa oleva tilanäkymä on toteutettu Qt Widgets -käyttöliittymäteknologialla. Työssä tutkitaan myös muita Qt:n tarjoamia käyttöliittymäteknologioita, ja toinen käyttöliittymäteknologia valitaan uuden tilanäkymän toteuttamiseksi Qt Widgets -käyttöliittymäteknologian sijaan.

Aluksi työssä tutustutaan työhön liittyvään automaatiojärjestelmään esimerkkiautomaatiojärjestelmän avulla. Esimerkkiautomaatiojärjestelmä konfiguroidaan käyttäen PC-sovellusta, ja sitä monitoroidaan PC-sovelluksen kautta. Taustakuvauksen jälkeen uuden tilanäkymän vaatimukset määritellään ja tilanäkymä suunnitellaan vaatimuksiin perustuen. Suunnittelutyössä hyödynnetään olio-ohjelmoinnin menetelmiä sekä UML-kaavioita, ja lopulta suunnittelutyö johtaa toteutuneeseen tilanäkymään, joka täyttää sille asetetut vaatimukset. Viimeiseksi työssä käydään läpi työn kulku ja esitetään jatkokehitysideoita.

Wapice Oy:n asiakasyritys, joka tilasi työhön liittyvän toteutuksen, on ottanut uuden tilanäkymän käyttöönsä. Asiakas on tilannut lisää toiminnallisuutta uuteen tilanäkymään, ja niiden suunnittelu ja toteutus tullaan tekemään lähitulevaisuudessa.

PREFACE

This master's thesis was written for the Department of Pervasive Computing in Tampere University of Technology (TUT). The thesis was written and the related work was implemented during the years 2015 and 2016. The system status view introduced in the thesis was ordered by a customer company of Wapice Oy in 2015, and the implementation of the system status view was completed in the same year. However, minor bug fixes and functionalities were implemented until March 2016. My co-workers at Wapice Oy created the initial concept for the system status view, and after the implementation was started, it became clear that by extending the concept, the task would be worth a thesis.

I would like to thank Otto Bothas (Wapice Oy) for guidance with the design and architecture of the system status view and Tommi Asp (Wapice Oy) for reviewing most of the program code related to the system status view and for advising with the technologies used to implement it. I would also like to thank the examiner of the thesis, Professor Kari Systä (TUT), who provided a lot of useful information for the structure of the thesis and guided me in the writing part of the thesis.

Tampere, 13.5.2016

Ilari Venäläinen

TABLE OF CONTENTS

1. INTRODUCTION 1

2. BACKGROUND AND ENVIRONMENT 3

2.1 Currently used system status view 7

3. GOALS OF THE THESIS 10

3.1 Basic functionalities 10

3.2 List view and multi-system support 13

3.3 Usability and performance 14

3.4 Support for CANopen devices 15

4. SYSTEM DIAGNOSTICS 17

4.1 States of the state machine 17

4.1.1 Bootloader 18

4.1.2 Initialization state 19

4.1.3 Pre-operational state 20

4.1.4 Operational state 22

4.1.5 Test state 22

4.1.6 Stopped state 23

4.2 Diagnostics of CANopen devices 23

4.3 CAN channel failure handling 26

4.4 Transmitting states of the modules to the PC application 28

5. ARCHITECTURE AND DESIGN OF THE SYSTEM VIEW 30

5.1 Qt Quick, Qt WebEngine and Qt Widgets 30

5.2 Design of the system view 33

5.2.1 Basic functionalities 34

5.2.2 List view and multi-system support 38

5.2.3 Usability and performance 40

5.2.4 Support for CANopen devices 43
5.3 Architecture of the system view 44
6. IMPLEMENTATION OF THE SYSTEM VIEW 47
6.1 Descriptions of C++ classes 48
6.2 Descriptions of QML documents 54
6.3 Showing the state transition of a module in the system view 58
7. EVALUATION 61
7.1 Realized goals 61
7.2 Future improvements 62
8. CONCLUSION 64
8.1 Thesis process 64
BIBLIOGRAPHY 66

LIST OF FIGURES

2.1 Structure of the automation system 6

2.2 Currently used system status view 8

3.1 Initial UI design for the basic functionalities of the system view, part 1 . . 11

3.2 Initial UI design for the basic functionalities of the system view, part 2 . . 13

4.1 States of the state machine 18

4.2 CAN and CANopen protocols presented with the ISO-OSI model 24

4.3 States of the NMT state machine 25

4.4 Error states and state transitions in CAN protocol 27

5.1 Module level presentation of the grid view 35

5.2 States of the system in the status bar and their respective tooltips 37

5.3 List view presentation of the system view 38

5.4 System level presentation of the grid view 39

5.5 Additional states and their respective symbols used in system and module group levels 40

5.6 Module group level presentation of the grid view 41

5.7 Navigation in the system view 42

5.8 Loading image used for animation when views are being constructed . . . 43

5.9 Architecture diagram of the system view 45

6.1 Sequence diagram of the state transition of a CAN channel in the system view 53

6.2 Sequence diagram of the state transition of a module in the system view . 59

LIST OF TABLES

5.1 Comparison of user interface technologies provided by Qt 32

6.1 Distribution of lines of program code of the system view implementation . 47

7.1 Set and realized goals of the thesis 61

LIST OF ABBREVIATIONS AND SYMBOLS

AC	Air Conditioner
API	Application Programming Interface
CAL	CAN Application Layer [15]
CAN	Controller Area Network [32]
CiA	CAN in Automation [15]
CPU	Central Processing Unit
CSS	Cascading Style Sheets. Used to describe the presentation of Web pages. [29]
DC	Data Container
DLL	Dynamic Link Library [11]
GUI	Graphical User Interface
HAL	Hardware Abstraction Layer [14]
HTML	Hypertext Markup Language [2]
ID	Identifier
IDE	Integrated Development Environment
IOM	Input/Output Module
ISO	International Organization for Standardization
JS	Javascript [30]
LED	Light-emitting diode
LDU	Local Display Unit
MCM	Main Control Module
MG	Module group, ModuleGroup
MOS	The Meta-Object System [20]
MSV	Microsoft Visual Studio
MVC	Model-View-Controller [6]
MVD	Mode-View-Delegate [21]
NMT	Network Management [15]
OOP	Object-oriented programming
OSI	Open Systems Interconnection
PC	Personal Computer
QML	Qt Meta-object Language [16]
RAM	Random Access Memory
SGML	Standard Generalized Markup Language [2]

SV	System view, SystemView
TUT	Tampere University of Technology
UI	User Interface
W3C	World Wide Web Consortium [2]
XML	Extensible Markup Language [2]

1. INTRODUCTION

Automation systems are widely used in the industrial field. Usually, an industrial automation system is a complex entity that consists of different physical devices that together fulfil the requirement of the automation system. These physical devices can be used to control and monitor other devices or machines by using electronics and software. Industrial automation systems are commonly used in production plants and lines. If an automation system is used in a production line, it is quite clear that the automation system should be functional and complete the tasks assigned for it as long as it is running. However, it is not always the case because the devices used in the automation system might break or not work as intended. In these cases, the production line is most probably not producing anything and the automation system cannot proceed its tasks. This causes a company to lose money, and in some cases an automation system cannot provide its desired functionality for the customer.

Automation systems can be controlled by PC applications, and nowadays many of the automation systems can be connected to the Internet. If an automation system is connected to the Internet, it can be controlled and monitored from any location with an Internet connection. An alternative way to control and monitor an automation system is from a device that is particularly manufactured for this controlling and monitoring purpose. If a PC application is used to monitor an automation system, it can be used to detect problems in the automation system immediately. When a problem is detected in the automation system, it should be fixed as soon as possible to prevent it causing any harm either to the company or the customers using it.

The purpose of this thesis is to design and implement a system status view for an industrial automation system. The system status view is a part of a PC application that is used to configure and monitor the automation system. The automation system is similar to the previously described automation systems. The customer, which is one of the Wapice Oy's customer companies, configures the functionality of the automation system with the PC application. The end users who are using the automation system provided by the

customer, monitor the automation system with the PC application. The system status view is implemented as part of this PC application to monitor the underlying automation system. The PC application has an existing system status view, but in this thesis a next generation of that is implemented. The currently used system status view is only used for monitoring the automation system, and in the new design, some options for configuring parts of the automation system are introduced.

At first, the automation system discussed in this thesis is introduced, and an example system is configured with the PC application. After that the currently used system status view and its problems are introduced. When the currently used system status view is familiar to the reader, the goals of the thesis are set and the idea of a next generation of the system status view is discussed. After that the actual design and implementation of the new system status view are discussed and lastly the results of the thesis are examined. The thesis consists of eight main chapters with the following structure:

Chapter 1 introduces the contents of the thesis.

Chapter 2 introduces the PC application on a general level, and then the environment where the automation system could be used is described with an example automation system. Last, the currently used system status view and the problems it has are described.

Chapter 3 presents the goals of the thesis.

Chapter 4 covers the system diagnostics of the automation system that are related to the new system status view.

Chapter 5 includes the design and architecture of the new system status view.

Chapter 6 describes the implementation of the new system status view.

Chapter 7 evaluates the realized goals of the thesis, and future improvements to the new system status view are discussed.

Chapter 8 includes a conclusion of this thesis and describes the thesis process.

2. BACKGROUND AND ENVIRONMENT

The system status view is part of a PC application that is used for an industrial automation system. The PC application is implemented using a Qt framework (discussed in Chapter 5), and it is a stand-alone program used in a Windows environment. The automation system consists of different types of embedded devices. The embedded devices related to this thesis are referred to as (hardware) modules or devices. The modules of this system use Controller Area Network (CAN) [32] protocol for communication.

CAN is a bus network, which means that each node in the network is connected by a single cable or twisted pair wire to the bus. A message transmitted by a node to a CAN bus is received by all the other nodes that are connected to the same bus [8]. CAN is more closely discussed in Sections 4.2 and 4.3.

The PC application is mainly used for two purposes: configuring and monitoring a specific variation of the automation system. Configuring is done by selecting a set of modules for the automation system and specifying what kind of tasks each module should execute when the system is running. An automation system that is running can be monitored, which means observing and adjusting it.

The PC application shows different kinds of views for configuring and monitoring an automation system. The views are constructed dynamically from XML files. The XML files contain the data for constructing all the views shown in the PC application. XML (Extensible Markup Language) is a standardized, text-based format for describing documents and data structures. XML is based on Standard Generalized Markup Language (SGML), which was part of an IBM document-sharing project created in 1974, and later became an ISO standard in 1986. Hypertext Markup Language (HTML), which describes a document layout and display in a standardized way and is part of every Web browser and website, was the first widely known adaptation of SGML. Even though SGML could share documents and HTML could describe the layout for the documents, there was no standardized format for describing and sharing data stored in the documents. Since the standardized format for data did not exist, programmers parsed HTML documents in var-

ious ways to fetch the required data. In 1998, World Wide Web Consortium (W3C) came up with the first recommendation for standardized format to separate data from the documents and the format was named XML. The structure of XML can be quite complex, but it is not intended to be read by a human-eye: XML parsers and other tools are used. [2]

An XML file that is related to a shown view in the PC application is read while initializing the view. Retrieved data from the XML file may contain user access level information related to the view, in addition to data used for constructing the view. A user has to login when the PC application is started and after a successful login, the current user level is accessible. Depending on the user level, some parts of the initialized view may be hidden or the view might not be visible at all.

The views shown in the PC application can be divided into two categories: configuration and monitoring views. Alternative names for these categories are respectively a configuration side and a monitoring side. The main purpose of the configuration side is to show the current XML configuration of an automation system. The XML configuration is a description of the automation system that a user is currently configuring or monitoring. The configuration side can be seen as an offline presentation of the automation system: there is no interaction with a running automation system, and it is used to configure an automation system. The views of the configuration side present the XML configuration of an automation system. The views are used to inspect and modify the XML configuration of the inspected automation system. During the login process, the user selects an automation system that he/she would like to access. All user levels are granted with an access to the monitoring side, whereas only some of the user levels have access also to the configuration side.

There are two kinds of components related to hardware modules: system components and application components. The system components are components that are essential to any automation system where the PC application is used. For instance, a CAN handler is a system component that processes messages received from a CAN bus by a module. The application components fulfil the requirements of the automation system. It could be said that the application components manage the behaviour of the automation system and the tasks assigned to the automation system. Two kinds of parameters can be configured in the PC application: system parameters and application parameters. These parameters are used in their respective components: system and application components. The system parameters are used to configure the system components that are commonly used within the whole automation system. The application parameters are used to configure applica-

tions that are executed by the modules. The values of these parameters are included in the XML configuration file.

A system package includes a configuration binary file, which is generated from the XML configuration file of the respective automation system, and application binaries for modules. The PC application generates the configuration binary from the XML configuration before the system package is uploaded to the modules. The system package is uploaded to the modules with the PC application via Ethernet or CAN by using a CAN adapter. The applications of modules are started after the upload. If everything can be initialized as configured in the XML configuration file, the modules are operational and will execute tasks as configured.

After a successful upload, the PC application can be used to start monitoring the automation system. The views of the monitoring side are also constructed from XML files, but the XML configuration cannot be modified on the monitoring side. The monitoring side is used to interact with a running automation system, and it can be seen as an online presentation of the automation system: the monitoring views show the system and the application parameters of a running automation system in real time. After the upload both the PC application and the automation system have the same configuration, and a connection to the system can be established by using Ethernet or by using a CAN adapter. When the connection is established, neither the configuration side nor the monitoring side can be used to modify the XML configuration. If the PC application is disconnected from the system and the configuration is modified before re-connecting, an upload is required. The upload is required to guarantee a compatible version of the XML configuration in both the PC application and the automation system. When a system is running and the PC application is connected to it, the monitoring side is used to observe the system. In practice this means that a user monitors values, which are related to the system or the application parameters, in different ways: using monitor views, trend views, a log view and custom views. There is also an independent view for system status, which presents the current state and status of the automation system to which the PC application is connected.

Figure 2.1 presents a high level structure of the previously described automation system. The PC application is used to create a system package for an automation system, upload it to the modules and to monitor the automation system. The modules of the automation system are controlling other devices connected to them.

Now that the main parts used in the automation system are described, it is time to present an example where the PC application is used to configure applications for modules by

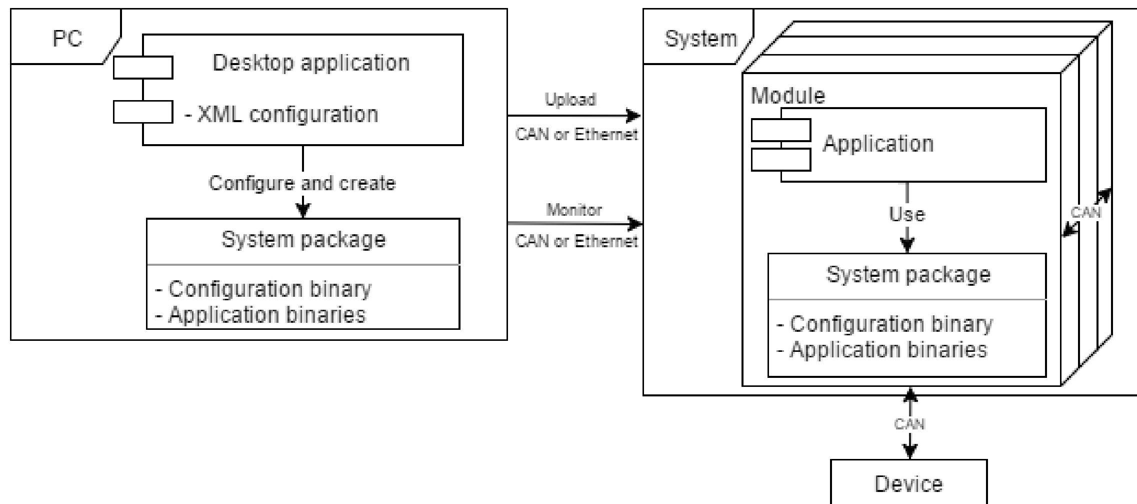

Figure 2.1 *Structure of the automation system*

designing a minimalistic air conditioner (AC) system. It is worth mentioning that the AC system is not designed for real use and the responsibilities of the modules are simple enough to make the example easier to follow. The AC system could be controlled by a single module, but more modules are introduced to have a more generic automation system where different types of modules exist. The modules and particles used by the AC system are described next. The AC system consists of four air conditioners, and each individual AC has one input/output module (IOM). The AC system also has a main control module (MCM) and a local display unit (LDU). A total of six modules are used, and all of them are connected to the same CAN bus. In addition, some thermal sensors in areas where temperature is controlled and fans to adjust the temperature of the areas are needed. A light-emitting diode (LED), which is used in the AC system, indicates if the AC system is currently adjusting temperature.

IOMs are connected to thermal sensors and the data fetched from the sensors is transmitted via CAN to the MCM. The MCM is connected to all the fans, and it controls the speed of the fans based on the information received from IOMs. The MCM also controls the LED that indicates the temperature is currently being adjusted. The local display unit shows which areas are currently being adjusted, what the temperature value in each thermal sensor is and what the state of the AC system is, i.e. is everything working as configured.

Now that all the modules and fans have been set, connections to sensors, CAN and fans are made, it is time to create an actual configuration and applications for the modules. The application running in IOMs could be as simple as reading data periodically from

thermal sensors and converting read values to Celsius unit. After reading and converting the values, they are transmitted to the MCM. An application programmed for the MCM checks if there is a need to adjust corresponding fan speed to match the current temperature of the area to the configured value and controls some analogue output (e.g. current), if necessary. If adjustments are made, the MCM also controls the global LED, indicating that some fans are being adjusted. Meanwhile the application running in the LDU has received the same information from IOMs as the MCM and displays it. In addition to this, the MCM transmits information to the LDU about fans that are currently being adjusted. The LDU has a mapping for fans to areas so that it can indicate which areas are being adjusted.

After an application for each module in the system has been programmed, configuring of the applications with the PC application can be started. System parameters are configured to set used CAN channels and I/O pins used by modules, IP for Ethernet connection (to the LDU), timeouts and various other parameters that are either optional or mandatory for the AC system to be functional. Application parameters for the MCM are used to configure thresholds for adjusting the speed of the fans. For the LDU, application parameters are used to create the mapping of fans to areas. For the IOM, a time period for reading thermal sensors and a unit for temperature readings is configured. The configuration binary, which is generated from the XML configuration of the system, includes these system and application parameters. After the parameters are configured, the system package can be uploaded with the selected media to the modules. After a successful upload and established connection to the system, a user can start monitoring the configured parameter values and also change or adjust them at runtime. For example, the user is able to change the thresholds, which controls the fans' speed, in the MCM.

When the system is running and the PC application has established a connection to it, the system status view is an easy way to see if the modules are in an operational state and work as configured. The modules also have other states that are discussed thoroughly in Section 4.1. One of the primary goals of this thesis is to enhance and improve the currently used system status view to have more functionality in it.

2.1 Currently used system status view

Figure 2.2 presents the currently used system status view. It can be seen that the system status view is rather simple, as it shows only three columns: name, state and status of a module. Name represents the configured user interface (UI) name for a module, state is

one of the states discussed in Section 4.1 and status shows additional information of the current state, if necessary.

Figure 2.2 Currently used system status view

The system status view does not provide any user interaction except that the size of the view and column sizes can be adjusted. There is no need for more user interaction because the view is used only to monitor the state of a running automation system. The PC application supports multi-system usage. When a user is logged in to multiple systems simultaneously, a separate system status view needs to be opened for each system. There is also a status bar for each system. The purpose of the status bar is to provide information related to the currently active system in the PC application. For example, the status bar shows if the PC application is connected to the active system. The user can only see one status bar at a time: the status bar of the system that is being configured or monitored. The status bar is also used to open the system status view: double-clicking an item that shows "system" in the status bar opens the corresponding view for the system.

To summarize, the currently used status view shows only the basic information related to the modules of the automation system, and there is not much left for the user to do. It is worth mentioning that the user cannot easily recognize the automation system that a status view represents because there is no additional information shown in the view. If the user cannot distinguish the related system from module names, the easiest way is to

reopen the system status view. The functionalities of the existing system status view were partly implemented by the author of this thesis in 2014.

3. GOALS OF THE THESIS

The initial concept for the next generation of the system status view including new visual look and improvements was designed by the author's co-workers. Goals related to the initial concept, which was renewed for this thesis, are shown in Figures 3.1 and 3.2. The figures present the initial UI design for the new system status view. From now on until the end of this chapter, "(x)", where x represents a number is used to reference a number within the referenced figure.

3.1 Basic functionalities

The system status view should be opened by double-clicking a status bar item "system" shown in Figure 3.1 (1). After the view is opened, it should show all the configured modules of the system (2). The name of the currently viewed system should be shown above the modules. This more graphical interpretation of the system status view is referred to as the grid view.

A system can have disabled modules (3), which means that a module is included in the XML configuration, but it is not intended to be used. The purpose of disabling modules is to quickly change the composition of the automation system for testing. The PC application ignores the disabled modules when monitoring the system. Disabled modules should be clearly shown in the system status view with a text, a symbol and a lighter image of the disabled module. The system status view should allow enabling and disabling modules used in the automation system. Because the view is not only showing the status of the automation system, a more suitable name should be used: a system view. This naming is used from now on for the view because it better describes the possibility to not only monitor the automation system, but also to configure parts of it. Modules should be selectable in the system view and after a module is selected, a user should be able to disable or enable the module. It should be also possible to enable and disable all modules at once, and buttons for these actions should be added.

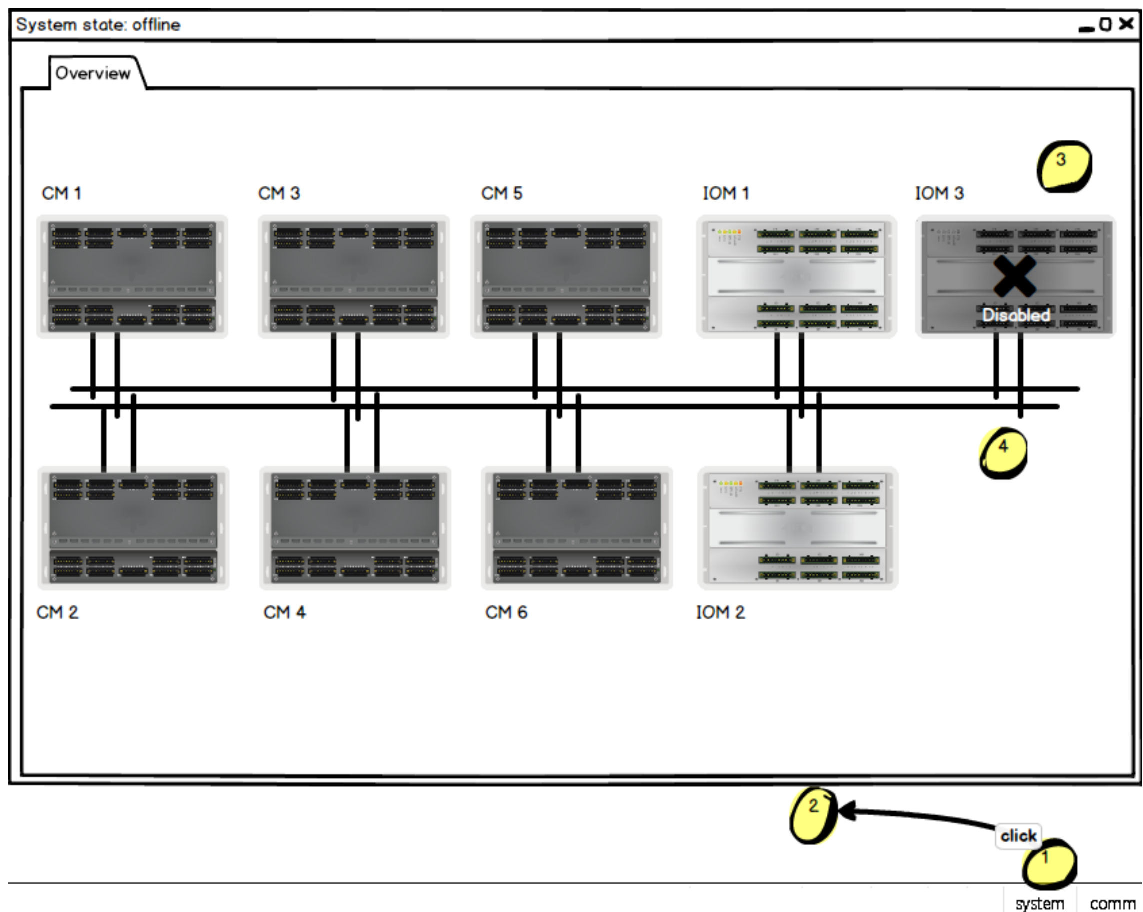

Figure 3.1 Initial UI design for the basic functionalities of the system view, part 1

Chapter 2 explained that only some user levels have access to the configuration side of the PC application. Enabling and disabling modules modifies the XML configuration of the system, so these actions should be restricted and not shown for users who can only access to the monitoring side of the PC application. Also, these users should not be able to select modules in the system view.

If the PC application is not connected to the system, the system view could be used to see an overview of the configured system. In addition to the modules, the configured CAN buses (4) for the system should be shown. The number of CAN buses varies from two to four in different systems, and the number depends on the automation system needs. The maximum number of CAN channel connections for a module is module dependent, and it varies from one to four. If a CAN channel of a module is connected to a CAN bus, the connection should be shown.

CAN is used as a communication media within the automation system, so functional CAN

channels for each module are vital in order for the system to be functional. The cause for a non-functional CAN channel is most likely a broken wire, a short circuit or a loosely attached cable. In Figure 3.2, (4) illustrates a situation where all the CAN channels of the module are not functional in one way or another and the state of the module is “not found.” The state could be also presented as “offline” because the state cannot be known.

A non-functional CAN channel should be indicated in the UI with a different color. When hovering over any non-functional CAN channel of a module, a tooltip should be shown. The tooltip should include information about all the CAN channels that can be configured for a module. If a CAN channel is not configured (it is not connected to any CAN bus in the system), “<channel>: not configured” text should be shown. Alternatively, channels that are not configured could be not shown at all. For a non-functional CAN channel, “<channel>: Bus off / Error” text should be displayed. For a functional CAN channel “<channel>: Normal” text should be shown. Because the number of CAN channels varies in the modules, the CAN channels should be displayed in logical order from left to right regardless of missing a connection to a CAN bus. The first configurable CAN channel should be shown as the leftmost one.

Figure 3.2 presents more basic functionalities. The states of the modules are still to be shown (1, 2, 8), but compared to the currently used system status view described in Section 2.2 and shown in Figure 2.2, the states should be presented more visually: adding an animation for some of the states (7) and a symbol to represent the state.

The current state of the system should be seen (8). The state of the system is composed of all the current states of the modules in the system, and it is discussed more in Section 5.2.1. The current state of the system should also be indicated in the status bar with a color and a tooltip. The state of the system is used to quickly see overall status of the automation system, either from the status bar or from the view, if it is open. If the PC application is connected to the system, the connection should be indicated in the system view so that the user knows if it is possible to start monitoring the system that the system view represents.

In the PC application, there is a separate upload dialog that is active when an upload is ongoing. The upload dialog has a progress bar for each module configured in the system and a lot of related log information on what is currently being uploaded or prepared to be uploaded for a module.

In Figure 3.2, (6) presents an upload progress bar that should be shown in the grid view

for each module. The currently used upload dialog should initially be hidden, but could be opened for a closer inspection of the upload progress, if necessary.

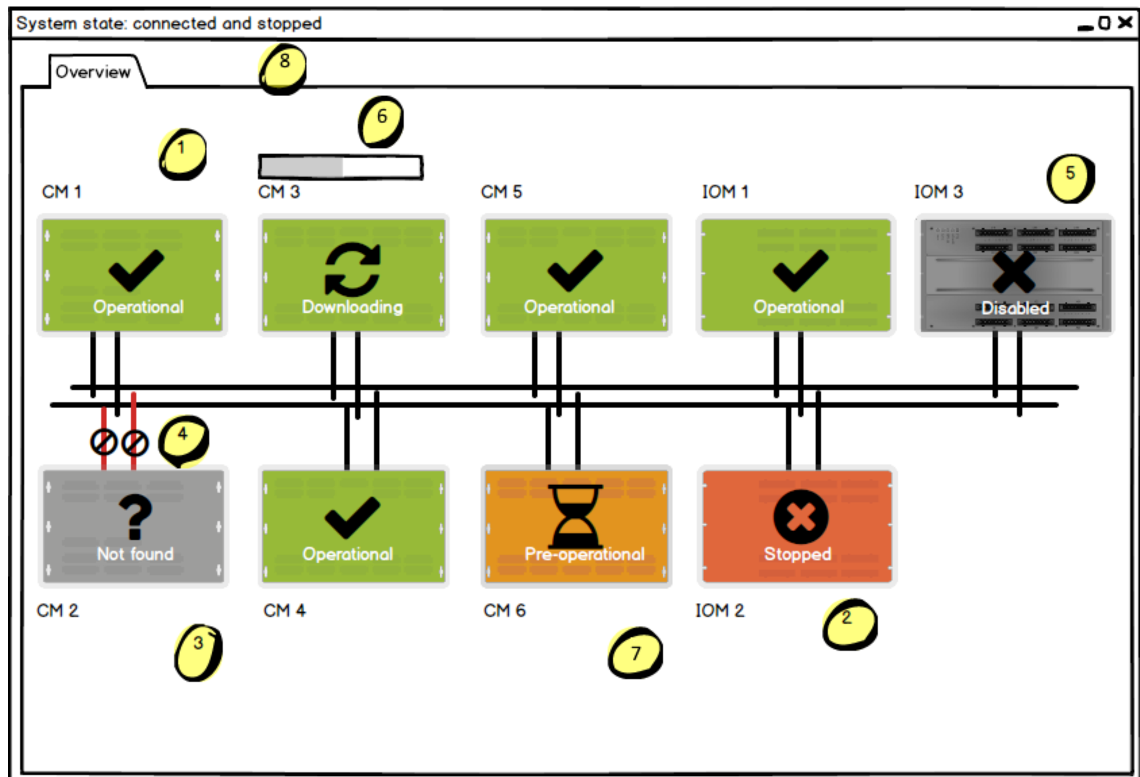

***Figure* 3.2** *Initial UI design for the basic functionalities of the system view, part 2*

3.2 List view and multi-system support

The system view should support simultaneous usage of multiple systems. The view should be optimized for four systems because it is the maximum number of systems monitored with a single instance of the PC application. The limitation is a requirement by the customer because there is no use case where more than four systems are used. However, it should be kept in mind that this limitation may change in the future. All systems, in which a user has logged in, should be easily seen from a single view.

When the system view is opened, the overall status for all logged in systems should be shown. Each system should be presented as a button-like rectangle. To clearly indicate the overall status of a system, the system rectangle should contain at least the following items: whether the PC application connected to the system, the current state of the system and the name of the system. To see more details of a specific system, a user should

be able to navigate into the system by pressing the rectangle that represents the system. When looking at more details, the currently viewed system should be clearly indicated and navigating back to the system level should be possible. These transitions from the system level to the module level and vice versa should be animated to look like zooming in to a system or zooming out from a system in case of navigating from the module level to the system level. The module level of the grid view is presented in Figure 3.2, and it contains all the configured modules for the system.

The system view should provide the possibility to monitor the status of a system from a list view. The currently used system status view is a list view, but as mentioned in Section 2.1 there is no multi-system support. In addition to the multi-system support, the new list view should show more information in it.

The list view should also show the overall information of all logged in systems like described for the grid view earlier. A button should be added to show only the overall information of all logged in systems, and this should hide the information related to modules. In addition, it should be possible to easily switch between the list view and the grid view.

It can be seen from Figure 2.2 that the color scheme of the list view could be improved. The currently used colors are very bright, and there is no grouping for modules by their type or any additional information besides the current states of the modules. The new color scheme should be matched with the grid view.

3.3 Usability and performance

When considering the previously mentioned module level in Section 3.2, it is quite clear that not too many modules with a graphical presentation fit in the system view. Figure 3.2 presents only nine modules, but in a real system there are usually 14–20 modules when CANopen devices, which are introduced in Section 3.4, are added to the view. An easy way to handle more modules is to add more height and width to the system view and fill empty space on top and bottom so there would be a total of four rows instead of two rows for modules. As the grid view contains a lot of graphics, a zoom functionality should be added to the view. When the view is zoomed out, it should fit into a smaller area. While when the view is zoomed in, the modules could be inspected more closely.

Another solution could be grouping the modules and adding one more level to the system-module hierarchy. A single module group consists of all the modules of the same type

in the system. This module group level should be configurable at runtime in a way that the modules that are grouped, can be selected from the view. A module group should have the same functionalities as described for a system in Section 3.2 and it should be presented like the system rectangle. Only the modules that belong to a module group should be shown when the module group is clicked. Transitions to the module level of a module group and from the module level to the module group level, should be animated in the same way as the transitions to/from the system level. The system level should be shown by default in the grid view and whenever a system is clicked the module level or the module group level is shown. The module group level should show grouped modules as system-like rectangles and other modules as presented in Figure 3.2. It should be possible to select whether the module group level or the module level is active. The active level should remain consistent when closing and re-opening the system view.

When a user is switching between the list view and the grid view, the sizes and positions of the windows should be stored and restored. This small adjustment will make the view more intuitive: if the list view is used on a secondary display and the grid view is used on a primary display, switching between the views should always reposition the window on respective display. The grid view is most likely reserving more space so it is important to resize the system view according to the size of the previously used list/grid view.

As far as performance is concerned, the system view should open quickly, navigation between different levels of the grid view and switching from the list view to the grid view and vice versa should be fast. A simple loading animation should be shown while constructing different parts of the system view. The animation indicates that the UI is not frozen and the PC application has not crashed. Memory consumption should be measured and it should be confirmed that there are no memory leaks, which should be a certainty in any application.

3.4 Support for CANopen devices

The automation system provides support for CANopen devices. The CANopen devices implement a CANopen protocol that is a communication network, and it is explained in Section 4.2. The CANopen devices used in the automation system can be regarded as an external system that is functioning on its own. The PC application cannot monitor the CANopen devices directly. However, the CANopen devices can be connected to the same CAN buses as other modules of the automation system. Then, a module is able to query information from a CANopen device to receive the current state of the CANopen device.

After receiving the current state of the CANopen device, the module transmits it to the PC application. The states of the CANopen devices are briefly introduced in Section 4.2.

Support for the CANopen devices should be added to the system view. The CANopen devices or the states of the devices cannot be seen in the currently used status view. A module, which is not a CANopen device, has to be connected to the same CAN bus as the CANopen device in order to receive the current state of the CANopen device. The module is then able to read the status of the CANopen device, and the PC application receives the state information of the CANopen device from the module. The PC application is used to configure the CANopen devices. The following information has to be configured for a CANopen device in order to monitor it: a module that monitors the state of the CANopen device and a Data Container (DC) code. DC codes are explained in Section 4.2, but briefly, the configured DC code for the CANopen device stores the value of the state. The DC code is stored in the module and transmitted to the PC application. More detailed explanation of the procedure is explained in Section 4.2. If the configuration for a CANopen device is missing mandatory information required to monitor it, the system view should indicate the missing information.

4. SYSTEM DIAGNOSTICS

Now that the environment of the automation system is covered in Chapter 2 and the goals of this thesis were set in Chapter 3, system diagnostics and states of the state machine are introduced. The state machine is a functional model that is executed by all the modules in the automation system except CANopen devices. The whole process of sending and receiving CAN messages inside the automation system or how messages are handled in the PC application is not covered in this thesis because such implementation details are beyond the scope of this thesis.

4.1 States of the state machine

A state machine makes it easier to ensure that a system where the state machine is used is working as it should. In an embedded system, a state machine is a logical choice for implementing functionality that changes whenever a mode of the system changes. The state of a state machine is a condition that is related to both inputs and outputs: what kind of input is required to move from State A to State B and what kind of output does the system provide in the State B. [7]

This section covers the state machine running in each individual module. The functionalities of the states are implemented by author's co-workers, and this section and its subsections are based on Laurikainen's and other Wapice empolyees' work [9]. The state machine has a total of five states when excluding bootloader, which is not considered as one of the states. The states are: initialization, pre-operational, operational, stopped and test state. A state of the state machine can be also referred to as a state of a module, as it is more intuitive when considering what kind of output the module provides in the state. After modules are reset, the bootloader starts executing and starts the applications of the modules. The default state of the state machine is the initialization state that is entered immediately after the bootloader. The first three states are entered in respective order, and each of the states enables more services of the module. The stopped state is entered if a module encounters a hardware or a software failure. Failures are detected during the

pre-operational, the operational and the test state. The test state is used to isolate some components of the applications running in the modules for testing, and the state has to be remotely controlled from the PC application.

Transitions between the states and ensuring that transitions between all the modules in the automation system are synchronized is handled by the state machine. The state machine models a life cycle of the automation system: if all the modules of the system are initialized and synchronized without errors, the automation system stays in the operational state. The stopped state is entered in case of an error is detected and the state machine remains in the stopped state until a reset is performed. The PC application receives log messages of the state transitions and error logs if a state could not be entered. As mentioned earlier in Section 2.1 the current system status view shows these states. Figure 4.1 presents the states of the state machine and transitions between them after reset and bootloader phases which are shown with dashed outlines.

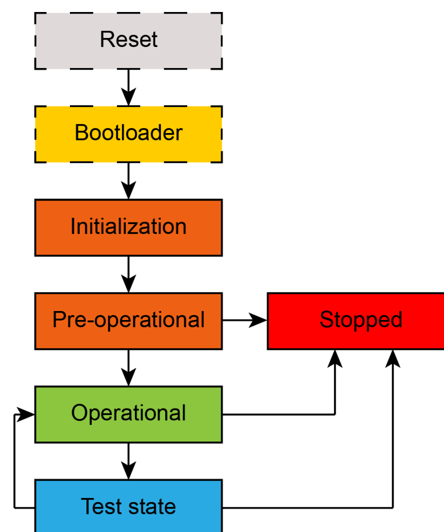

Figure 4.1 *States of the state machine*

4.1.1 Bootloader

In general, a bootloader loads an operating system kernel and its supporting infrastructure into memory and begins kernel's execution. In embedded systems, it is possible that a microchip to be used in a system does not come with a prewritten firmware. Usually the firmware would handle initializing Random Access Memory (RAM) timings in the memory controller circuitry, clearing the processor caches, and setting sane default values to Central Processing Unit (CPU) registers. Programming a suitable bootloader can

be done by porting an existing bootloder to fulfil the microchip and the operating system requirements. There are many bootloaders for embedded systems which could be ported, but U-Boot is the most flexible and most actively developed open source embedded bootloader available [4]. [31]

All the modules of the automation system have two bootloaders. The first one, called bootloader one, is a custom U-Boot ported for a module. The primary purpose of the bootloader one is to flash the second bootloader in a specific memory region of module's RAM. The first bootloader is executed only when the second bootloader needs to be upgraded. It is very unlikely that an automation system in use by the end user requires upgrading the second bootloader. The second bootloader is executed always when the PC application uploads a system package that contains the application and configuration binaries. In order to upload the system package to a module, which is not currently executing the second bootloader, the module has to be first commanded to enter the second bootloader by the PC application. If the binary files are corrupted, the module will remain in the second bootloader and will require another upload to proceed further. An upload is also required after modifying the XML configuration of the system in order to monitor the system.

4.1.2 Initialization state

The initialization state is entered after a module has been reset and the bootloader has started the application. In the initialization state the basic routines of the module are executed. If an error is detected during the initialization state execution, causes for the error are logged by the failing component to the PC application. Initialization statuses of all the initialized components are stored in the state machine component of the module to be used later in the pre-operational state.

The initialization state ensures that all constructors of application and system components are called in addition to performing a post-initialization. The post-initialization is an optional initialization routine that, for example, can be used to check allocated runtime data for an application. These post-initialization functions are called only if normal initialization for all components was successful, and the functions themselves may report an initialization failure for the module. After all the initialization procedures have been completed, the state machine logs the status of the initialization to the PC application and enters the pre-operational state.

4.1.3 Pre-operational state

The pre-operational state is entered automatically after the initialization state. The pre-operational state uses information stored during the initialization state to handle the actual logging to the PC application. During the pre-operational state, the state machine ensures the use of correct software and configuration versions for a module. The CAN handler component, which was introduced in Chapter 2, sets CAN communication to a pre-operational state that allows custom communication with the other modules in the automation system. Because of this, a module whose state machine is in the pre-operational state is also aware of the other modules in the system and their current states. After initializing necessary data with the other modules in the system, the state machine performs a state transition to the operational state if transition conditions are satisfied.

Each module has a hardware ID that is configurable with an ID jumper. A jumper is a conductor used to bond two or more parts of an electrical circuit [10]. Jumper connections are read while executing the second bootloader. Reading the jumper connections almost immediately after a module is powered on allows applications to refer hardware IDs of the modules as numbers. As mentioned in Section 3.3, modules of the same type forms a module group. Every module group has a master module, which is discussed next.

The pre-operational state is divided into four phases: initialized, synchronization, preparation and ready. The initialized phase assigns previously mentioned master modules for module groups in the system if those have not already been assigned. A master module of a module group is responsible for synchronizing application and configuration binaries to the other modules in the same group during the next phase of the pre-operational state. Modules that are part of the system are taken into consideration when determining the master module for a module group. A module with an ID that is not configured to be part of the system is an illegal module. Illegal modules and modules that have failed during the initialization state are not considered to be in the master module selection.

The master module of an LDU group is primarily performing application and configuration synchronization during the next phase of the pre-operational state. The master of an LDU group performs synchronization for all the modules in the system. However, if the system does not have an LDU or it has failed or not powered on, the master module of each module group performs the synchronization for the other modules in the same group. A master module executing synchronization transmits information to all booting up LDUs to prevent them to starting a synchronization procedure. Modules that are waiting for synchronization stay in the synchronization phase until the synchronization is completed by

the master module or a configurable timeout period is expired. If synchronization for a module is not completed before the timeout, the module enters the stopped state and a log message is transmitted. After the synchronization phase, possible failures that were encountered during the initialization state are checked. In case of a failure, the state is immediately transitioned to the stopped state and a log message is transmitted.

All hardware inputs and outputs of a module can be mapped to a specific data source. Mapped data sources can be read or written with the PC application, and applications of the modules can also utilize these data sources. In the preparation phase, all the hardware inputs are read and outputs are written for the first time. Outputs are set to safe values to prevent undefined behaviour in devices that are connected to output channels. In addition, periodic reading of hardware inputs is enabled at this point.

After all the inputs are read and outputs are written, the state machine calls registered callback functions that should be executed in the pre-operational state for the system and application components. The callback functions are configured and registered with the PC application and included in the configuration binary. These callback functions have access to mapped data sources so they can perform more necessary initialization before entering the operational state. The callback functions would return succeeded status if the configured initialization routines were successful. The callback functions are called periodically until all of them return succeeded status or a configurable timeout period is expired. It is configurable if the state machine should enter the stopped state in case of any of the callback functions return failure or the timeout has expired.

After all the previous phases have been completed successfully, a module enters the ready phase. The purpose of the ready phase is to indicate to the other modules in the automation system that everything has been set up for the module and it is ready to enter the operational state. The state machine provides a synchronized boot-up functionality, where a module in the ready phase waits for the other modules to enter the same state and phase. The synchronized boot-up functionality can be enabled or disabled, and a timeout period for waiting the other modules can be set. The XML configuration has a list of mandatory modules that are required by the automation system to be functional. The list is configurable, and if a mandatory module is not ready after the timeout period, all the modules enter the stopped state. If all the modules reach the ready phase, they enter the operational state.

4.1.4 Operational state

The operational state is entered either from the pre-operational or the test state. In the operational state, the state machine provides all the configured services and a module is expected to be fully functional. Applications programmed for a module are executed only when the state machine has entered the operational state. Applications are executed periodically and the period is configurable. After the execution of an application has finished, the application waits for the next execution round.

Modules that store read or written data from hardware inputs or outputs are configurable. Applications also have data that is used by the applications and/or monitored with the PC application. Together, the hardware input/output data and the application data can be called process data. The process data of a module is distributed to all the other modules in the system which are configured to store the data. When entering the operational state, the CAN communication state is set to an operational state in order to transmit and receive CAN template messages. These template messages are used to distribute the process data within the modules in the automation system. In the operational state, the PC application is able to monitor all the process data used by the modules. The process data one wants to monitor with the PC application is selected from the monitoring side of the PC application after the PC application is connected to the system.

Usually, the operational state stays in execution from now on, if all the states before the operational state were executed without errors. There are some cases that might cause the state machine to exit the operational state. If a user wants the system to enter the test state, it is possible only when the state machine is in the operational state. Transition to the test state or the stopped state may also be requested by other modules in certain circumstances. Also, if a module encounters a fatal hardware or a software error the module transitions to the stopped state.

4.1.5 Test state

The test state is used for testing different parts of the system in an isolated environment. Entering the test state has to be explicitly ordered by the state machine. The state machine receives commands from the PC application, and the transition to the test state is only possible if the state machine is in the operational state.

In the test state, all the tests selected by a user are executed and a transition back to

the operational state is requested. Executed tests are selected from the PC application by selecting pre-programmed tests for a specific module. During a test execution, it is possible that a critical system component is disabled or modified in a way that a module cannot execute its tasks normally afterwards. In these cases, the state of a module is treated as a non-recoverable state. If the non-recoverable state is entered, a transition back to the operational state cannot be performed after all the tests are executed. Instead, a transition to the stopped state is performed.

4.1.6 Stopped state

The stopped state is a failure tolerant state where only the critical system services are enabled. Services such as reading hardware inputs, writing hardware outputs and application execution are disabled. A module in the stopped state can communicate with the other modules in the system, and the process data can be accessed from the PC application. The values of the process data are left in a state just before disabling the corresponding service for modifying a single data element.

If the previous state before entering the stopped state is the test state, software can reboot a module. If the stopped state is entered from any other state, the state machine remains in the stopped state as long as a manual reboot is done or an upload is performed.

4.2 Diagnostics of CANopen devices

CAN is currently the leading serial bus[1] protocol for embedded systems [32]. Originally CAN was designed to be used in automobiles as introduced in 1986. After several years, in 1992, industrial manufacturers and users established the CAN in Automation (CiA) association. During its first years the association created a specification for CAN Application Layer (CAL) which enabled easier usage of the CAN protocol for industrial applications. However, CAL was impractical because as an application layer it required other users to design a new profile to be able to use it. This led Bosch GmbH and various organizations to design a prototype for a new profile, which would become known as CANopen, based on CAL to provide better embedded networking in production cells. In 1995, CiA had completely revised the prototype and released the CANopen communications profile. The CANopen profile defines device, interface and application profiles,

[1] In a serial bus data is transferred bit-by-bit or more generally from one point to another [1]

in addition to a framework for programmable systems. An application profile is used to specify all device interfaces used in a specific application. For example, lift control systems and light railways were one of the first CANopen application profiles. [15]

The difference between CANopen and CAN protocols can be explained with the ISO-OSI model. The CAN protocol covers the bottom two levels of the model: the physical layer and the data link layer. The physical layer defines ranges of voltages and currents, the types of cables and connectors that should be used and bit encoding. The data link layer defines conditions for received messages that should be accepted by a node, mechanisms for recovery management and flow management and CAN as a frame-based (message-based) protocol. The CANopen protocol covers the top five layers (bottom-up): the network, the transport, the session, the presentation and the application layers. The network layer defines addresses for the nodes and routing between them. A delivered message in the network layer might not be reliable, and the transport layer takes care of end-to-end reliability. The session layer controls connections between nodes. The presentation layer ensures that the data is encoded and represented as specified in the standard. The application layer defines how CANopen devices are configured, transferred and synchronized. Figure 4.2 presents the CAN and the CANopen protocols with the ISO-OSI model. [12] [13]

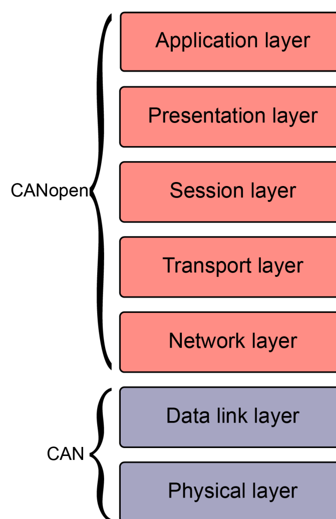

Figure 4.2 *CAN and CANopen protocols presented with the ISO-OSI model*

The CANopen protocol specifies a Network Management (NMT) state machine that each CANopen device must implement. Figure 4.3 presents the states of the NMT state machine. After a CANopen device is reset, the state machine enters the initialization state. CAN and CANopen interfaces and the communication network are initialized in the ini-

tialization state. After that the pre-operational state, which is the first state of the three major states, pre-operational, operational and stopped, is entered. State transitions between all the major states are possible. The details of the services provided by the state machine in each state are omitted because CANopen devices are not in a crucial role in the automation system. The states can be considered as the states of the other modules described in Section 4.1: the operational state provides all the services described in the CANopen protocol, the pre-operational state provides less services than the operational state and the stopped state provides only the critical services. [15]

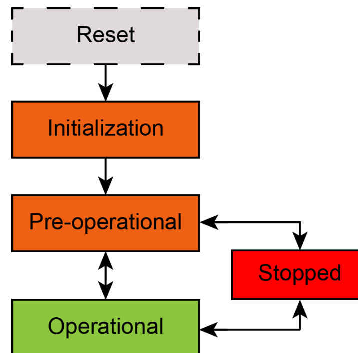

Figure 4.3 *States of the NMT state machine*

One of the critical services that are available in all the three major states is the heartbeat service. Heartbeat is a low-priority status message transmitted periodically by CANopen devices to CAN buses to which the CANopen devices are connected. The heartbeat message contains the current state of the NMT state machine running in a CANopen device. It was mentioned in Section 3.4 that the PC application is used to configure a module that monitors a CANopen device. The monitoring module receives these heartbeat messages from the CANopen device and stores the information. [15] The PC application is used to select a Data Container code that is used for storing the information. DC codes are an internal data storage mechanism to configure and monitor the mutual data of the PC application and the automation system. A DC code is represented as an integer value of four bytes where the first two bytes represent a code index. The code index is the index to the vector where the data related to the DC code is stored. The third byte represents an attribute ID that is the type of the value in the vector, for example, *int, float* and *short int*. The fourth byte represents an offset that is used to retrieve data related to the attribute ID. For example, *int* and *float* values are stored with different attribute IDs and the offset is used to read the correct type of the value from the memory. New DC codes can be created with the PC application. DC codes are made available for a module by including them in the configuration binary used by the module. The PC application generates and adds

DC codes to the configuration binary. Now the PC application and the modules are aware of the DC codes and the PC application is able to monitor and adjust the values of the DC codes. Whenever the value of the DC code, which represents a state of a CANopen device, stored in the module changes, the value is transmitted to the PC application. The value is transmitted only if the monitoring module is in the operational state as described in Section 4.1.4.

The CANopen devices used in the automation system implement the NMT state machine. The PC application is able to receive the states of the CANopen devices from the modules that are monitoring the CANopen devices. Heartbeat monitoring has to be enabled for a CANopen device from the PC application by modifying the XML configuration on the configuration side of the PC application. After that, a module that stores a DC code that represents the state of a CANopen device has to be selected. In the PC application, new DC codes can be added to the XML configuration, and the DC codes can be monitored when the PC application is connected to the system and the module that stores the monitored DC code is in the operational state. In addition to this, both the module that monitors a CANopen device and the CANopen device have to be connected to the same CAN bus.

4.3 CAN channel failure handling

The CAN protocol specifies a fault confinement mechanism to detect defective nodes in a network. If a faulty node is detected, the node is set in a passive or an off state to prevent it affecting the CAN buses to which the node is connected. The protocol specifies three states that are used to represent the current CAN state of a node: error active, error passive and bus off. The CAN communication states mentioned in Section 4.1 are used internally by the applications executed by the modules, and the states specified in the CAN protocol are used by a CAN controller. All the modules of the system come with a hardware CAN controller that is a separate microchip implementing the CAN protocol. [12]

Figure 4.4 presents the error states and the state transitions specified in the CAN protocol. Nodes in the error active state are functioning properly, and they can transmit and receive messages on the bus normally. Nodes in the error passive state are still able to transmit and receive messages, but they might have a faulty behaviour on either the transmission or the reception side. If a node in the error passive state transmits a message at the same time as a node in the error active state, the message transmitted by the node in the error passive state is delayed. This means that messages transmitted by a node in the error active state

have higher priority than messages transmitted by a node in the error passive state. Nodes in the bus off state are most likely corrupted and cannot transmit or receive messages on the bus. A state transition is performed according to the value of two integer counters: *transmit error count* and *receive error count*. The counters represent the likelihood of a faulty behaviour either on the transmission or the reception side. There are many rules specified in the CAN protocol for incrementing and decrementing the counters, which are not covered in this thesis, but as an example, the transmit error count is decreased by one if a message is transmitted successfully. [12]

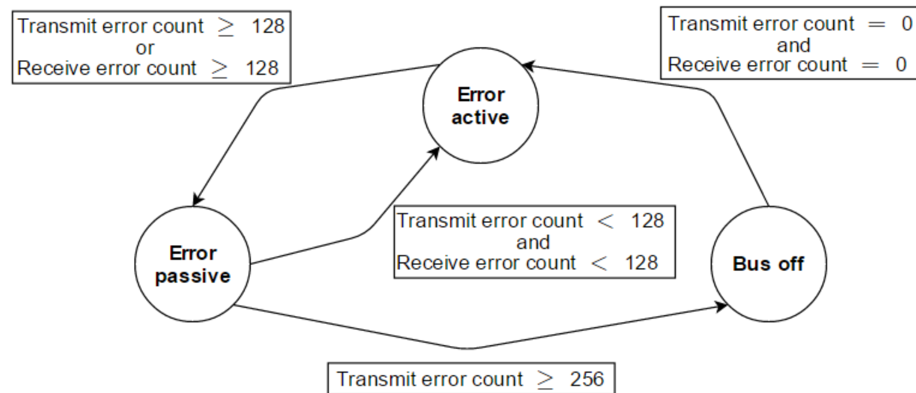

Figure 4.4 *Error states and state transitions in CAN protocol*

The system parameters of the CAN handler component have DC codes that are used the same way with the CAN error states as the DC codes with the CANopen devices. A DC code is selected for all the configured CAN channels of a module and the DC code is included in the configuration binary used by the module. The application running in a module can access the data related to the CAN error states of a CAN controller through a Hardware Abstraction Layer (HAL)[2]. The error state of each CAN channel is periodically read from the CAN controller through HAL, and the value is stored in the DC code configured for that CAN channel. If the value is changed, the module transmits the new value to the PC application. Because the PC application can be connected only to one CAN bus at a time, the values have to be distributed to the other modules in the system in order to always receive the information of a CAN error state transition. For example, if the PC application is connected to CAN bus one and CAN channel one of an MCM goes into the bus off state, CAN channel two of the MCM is used to distribute the value to the other modules in the system. The PC application then receives the information of the state transition from any other module transmitting it on CAN bus one.

[2]HAL is commonly used as an interface for applications that requires an access to a microcontroller and its peripherals. The HAL encapsulates the underlying hardware in a way that the applications using the HAL are still functional if the interface remains the same but the hardware is changed. [14]

4.4 Transmitting states of the modules to the PC application

The PC application receives and handles information of the state transitions in two different ways depending on the communication type. Before the PC application can connect to the system and start monitoring the system, the communication media has been selected. An initial connection to the automation system is established after the communication media is selected. The initial connection is referred to as a pre-connection to the automation system. Only the state information of the automation system is received when the pre-connection is established. The PC application receives CAN messages from the modules when CAN communication is used. A CAN message in the pre-connection state contains the following information: an ID of the module (transmitter), current state of the module and a timestamp of transmission. The CAN message is transmitted by each module periodically every 100 milliseconds (ms), and it is called an alive message. The PC application stores the information, and the timestamp is used to detect a module failure. If a module that has been transmitting alive messages is not transmitting another alive message within three seconds, it is treated as a failed module. If the state of the module remains the same for two or more consecutive times, only the timestamp is updated. Otherwise the state is further processed in the system view and it is discussed in Section 6.3.

All the states of the modules are transmitted as a single package when the PC application uses Ethernet as a communication media. Each package is wrapped in a custom profile that describes the contents of the package transmitted via Ethernet. This makes the handling of the packages easier because both the PC application and the module are always aware of the contents of the package. In addition, the PC application may be used together with a system that is not upgraded to the same version as the PC application. The PC application then queries from the system (in this case from the LDU), what profiles are supported by the system, and it uses the information to enable or disable profile-related services in the PC application.

The profile for transmitting status information of the modules consists of IDs and the current states of the modules. In addition, the profile contains header information to make the transmitted package unique so it can be detected in the PC application. An LDU has data structure for the states of all the other modules in the system, and it updates the structure according to received alive messages. Every 500 ms, an LDU checks if one of the states have changed. If a state has changed, the LDU transmits the states of all the modules in the system to all listening clients. A client is a running software connected to

the LDU via Ethernet, like the PC application.

There are some reasons why an LDU transmits all the states in one package to all connected clients in such long intervals. First, the size of the package is relatively small since both the ID and the state of a module are encoded with one byte. If considering a system that consists of 20 modules, the size of the package would be 40 bytes + the size of the header. If only one state would be transmitted, the size of the package would be two bytes + the size of the header. By adding only 38 bytes more, which is a relatively small amount of data in Ethernet communication, the state information of the whole system is provided for a client. Second, if a new client connects to the LDU, it does not have the current states of the modules. When sending the first package to the new client, all the state information is provided. If another type of PC application would implement the same profile for the package in question, it is desirable to provide all the information as one package without necessity to know how the information is used. Third, 500 ms time period is quite infrequent, but it is frequent enough to update the UI appropriately. A synchronous updating of a Graphical User Interface (GUI) and preventing flickering in the GUI is achieved easily by using such a time period.

The custom profile for transmitting the states of the modules via Ethernet, and its related functionalities were partly implemented by the author of this thesis. Also some modifications to the PC application were programmed to handle alive messages received via CAN.

5. ARCHITECTURE AND DESIGN OF THE SYSTEM VIEW

This chapter discusses the architecture and the design of the system view. As mentioned in Chapter 2, the PC application is based on Qt framework. Qt is a cross-platform application development and user interface framework. Qt framework comes with Qt Creator, which is a cross-platform Integrated Development Environment (IDE) used for application and UI development [18]. Qt is implemented with C++ programming language [16], and C++ is used by default for developing applications with the framework. C++ binding is provided by the Qt Company, and other bindings such as Java, Lua, Python and Ruby are provided by third parties [5]. Qt makes application development easier by providing pre-built UI components, over 1000 high-level C++ classes with well documented Application Programming Interfaces (APIs) and interactive UI designer as a part of Qt Creator [25].

Because the PC application is already utilizing Qt framework, using the framework for the new system view would be a natural choice. In addition, the currently used system status view is implemented with Qt framework, and the view is embedded as part of the PC application. However, when designing a next generation implementation for the system view, it should not be taken for granted to rely on the previous implementation. The design process should rather be started from scratch.

5.1 Qt Quick, Qt WebEngine and Qt Widgets

This section compares the user interface technologies provided by Qt. User interfaces can be programmed not only with C++, but also with Qt Meta-object Language (QML) [16]. QML is a markup language, similar to HTML, and it is composed of elements enclosed with curly brackets (e.g. Rectangle{...}). A QML document is a file that contains one or more of these QML elements that are also known as QML types. QML types can be extended with JavaScript (JS), which is a dynamic scripting language used widely in Web [30] [3]. In QML, user interfaces are built in a declarative way: a programmer

writes a QML document/type and describes what it should do instead of how it should be done. Implementation of user interface benefits from the simplicity of compounding QML documents and types together for desired functionality [28].

Qt Quick is a software module that supplies pre-written QML types for creating user interfaces such as a rectangle with its own coordinate system and rendering engine. The rectangle could have different states and contain some other QML types that could be animated. Transition effects between the states and animations are a first-class concept in Qt Quick, and it has been made easy for a developer to adopt using of them. To make user interfaces even better, visual effects can be supplemented through specialized components for particle and shader effects by applying them to QML types. [28]

In desktop environments, traditional user interfaces are implemented with Qt Widgets. The widgets are GUI components, such as a button or a check box, that independently interact with a user. The widgets provide native look and feel on different operating systems, and the components are mostly used for static user interfaces. The widgets do not adapt as well as Qt Quick for touch screens and highly animated modern user interfaces. Traditional desktop-centric user interfaces, such as office applications, could be approached with the Qt Widgets. [28]

Qt WebEngine is a Chromium-based layout engine. WebEngine makes it easy to embed content from the Web, which would be normally viewed with a Web browser, into a Qt-application on a platform which does not natively provide or support a Web engine. Qt provides another software module, called Qt WebView, for showing Web content without including a full web browser stack. WebView uses native APIs of platforms instead of WebEngine for showing Web content. [26] [28]

Table 5.1 summarizes three previously mentioned user interface technologies. Table 5.1 is based on the table shown in the official Qt documentation Web site [28]. In Table 5.1, X on a row means that a corresponding feature or component is provided by the respective user interface technology, and (X) means that it is provided partially.

The currently used system status view is implemented with Qt Widgets. Table 5.1 is used to decide whether or not the new system view should be implemented with Qt Widgets. Qt WebEngine is excluded from decision making because it not only provides the least features, but also there is no existing server from which to fetch data required by the system view. It is possible to create Web content and store data locally, but it would serve only the PC application and Web content usually implements the client-server model. If

Table 5.1 *Comparison of user interface technologies provided by Qt*

Feature	**Qt Quick**	**Qt Widgets**	**Qt WebEngine**	**Comments**
Used language(s)	QML JS	C++	HTML CSS JS	
Native look and feel	X	X		Qt Quick and Qt Widgets provide native look and feel on Windows, Linux and OS X.
Custom look and feel	X		(X)	Qt Quick is better for UIs that do not aim to look native.
Touch screen	X		X	Qt Widgets usually require a mouse cursor for good interaction, whereas Qt Quick is designed with touch interaction in mind. WebView component in QWebEngine provides multi-touch gestures for web content.
Standard industry widgets		X		Qt Widgets have been developed over two decades. A lot of different kinds of widgets are provided.
Rapid UI development	X		(X)	With Qt Quick, UI prototyping and development is fast.
Graphical effects	X			Qt Quick with particle system and shader effects is very flexible. Qt Widgets have very little to offer in this area.

the client-server model with Qt WebEngine would be implemented to show related data as Web content, it would require considerably more work: the PC application should be extended to communicate with a server that would store designated data for the system view. On the other hand, such a server would be rather useful if it would also provide data used by other components, like the trend and monitor views in the PC application. By using an external server to fetch required data for a view, the implementation of the GUI components would not be limited to use Qt framework at all: a Web client could be used to retrieve data from the server and show related information with HTML, Cascading Style Sheets (CSS) and JS.

The first thing to consider, when comparing Qt Quick and Qt Widgets, is native and custom look and feel features. Both of the technologies provide native look and feel in Windows environment, but Qt Quick offers easier implementation for custom look and feel. Chapter 3 discussed that a more graphical UI should be designed including custom drawing, for instance, for CAN channels. In addition, if the new system view would be adapted to any touch screen device, Qt Quick already provides support for these devices. With Qt Widgets it would probably be necessary to re-factor the programmed code base to separate the handling of touch events and mouse events, which would not be necessary with Qt Quick.

When considering standard industry widgets, Qt Widgets have much to offer. However, as mentioned in Chapter 3, the new system view is supposed to be a next generation view where standard industry widgets are not required: more graphical interpretation of the underlying automation system and user interaction should be provided. Sections 3.1 and 3.3 discussed that some of the UI elements and navigation in the system view should be animated. As mentioned earlier, Qt Quick is more suitable for highly animated user interfaces. Qt Quick also offers much more when it comes into graphical effects that could be used to enhance the UI. Furthermore, rapid UI development encourages the use of Qt Quick for fast development and UI prototyping. The UI is expected to change many times during the implementation of the new view, so the faster it is to apply and test suggestions based on internal or customer feedback, the better.

In conclusion, Qt Quick seems to offer more than Qt Widgets when it comes to fast development, more graphical and highly animated modern UIs. Thus, Qt Quick is selected as a user interface technology for the new system view.

5.2 Design of the system view

The system view component is realized as a dynamic link library (DLL). By using a DLL, a program can be modularized: unused parts of the program are not loaded into memory at all. This makes launching the application faster, the program consumes less memory and the architecture of the program can be modularized. DLLs are loaded into memory at runtime and multiple programs can use the same DLLs. The integration of the system view component to the PC application is described next. [11]

The PC application uses Qt's plugin architecture. In a Windows environment, these plugins are created as DLLs and they can be thought of as software components that extend a

related application. The plugin architecture provided by Qt is rather simple. For extending an application with a plugin, the following steps are needed [19]:

1. Define an interface (in C++, a class with only pure virtual functions)
2. Use Q_DECLARE_INTERFACE macro to tell Qt's MOS about the interface
3. Use QPluginLoader in the application for loading the plugin
4. Use qobject_cast for testing if the loaded plugin implements the interface

When the PC application is extended with the system view plugin, the previously described procedure is followed. QPluginLoader is used to load the system view plugin whenever it is required. For example, when the system view is opened for the first time, the DLL that provides the system view functionalities is loaded. For other software components that would use the functionalities provided by the system view plugin, the plugin offers an ISystemViewController interface. The interface and its functionalities are discussed in Section 6.1.

To program a plugin that implements an interface the following steps are needed:

1. Declare a plugin that inherits from QObject and from the interface(s) the plugin provides
2. Use Q_INTERFACES() macro to tell Qt's MOS about the provided interfaces by the plugin
3. Export the plugin using the Q_PLUGIN_METADATA() macro
4. Build the plugin and create a DLL of it

These steps are used when programming the actual implementation of the system view plugin. Another benefit to using a plugin architecture is that the implementation of a plugin can be re-factored or changed as long as the interface for it remains the same. [19]

5.2.1 Basic functionalities

The system view is opened by double-clicking a status bar item "system" shown previously in Figure 3.1. Figure 5.1 presents the module level of the grid view described

in Section 3.2. Figure 5.1 presents the module level in a situation where the communication media is selected from the PC application and the pre-connection is established to the automation system. It shows all the configured modules in the system. “View by modules” button with an orange background indicates that the module level is active. It is worth mentioning that the customer provided support with the graphics shown in the design figures.

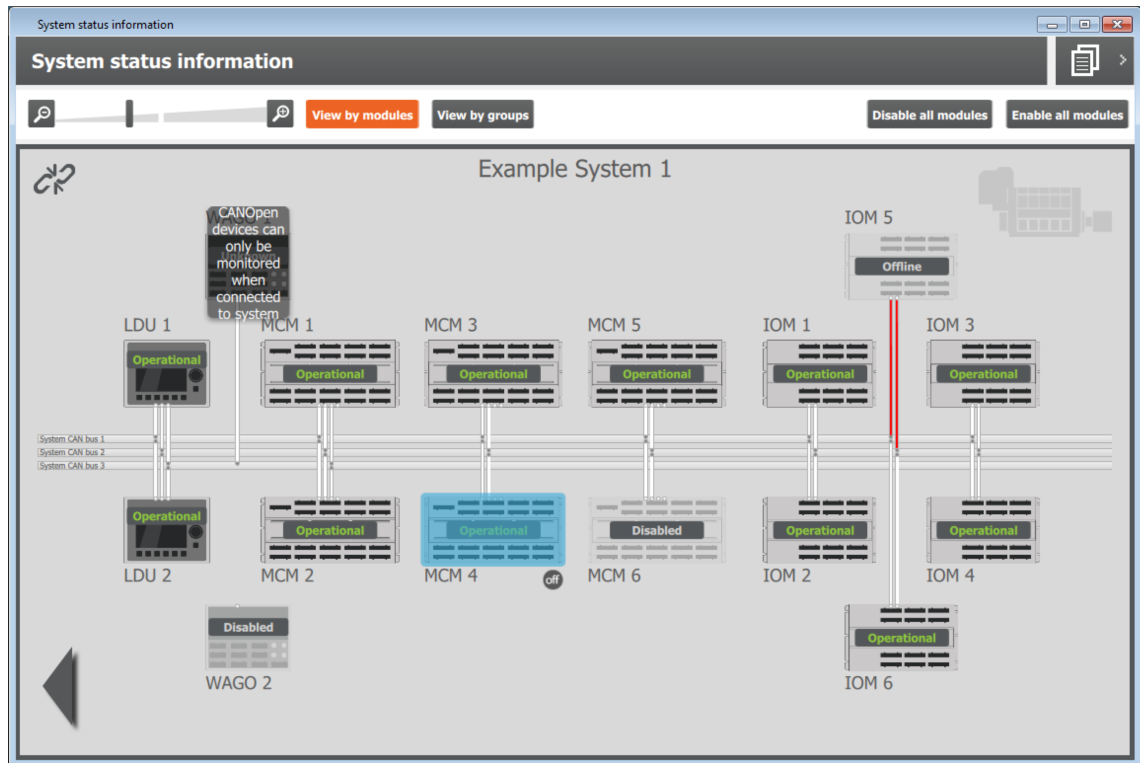

***Figure 5.1** Module level presentation of the grid view*

Disabled modules are shown with a lighter image of the respective modules and with a “Disabled” text. In Figure 5.1, CANopen device WAGO 2 and MCM 6 are disabled. MCM 4 presents a selected module. The selected module is shown with a transparent blue color and there is an off button that disables the selected module. If a disabled module is selected, the button shows “on” and pressing it enables the module. At top right, there are buttons for disabling and enabling all configured modules in the system. If the PC application is connected to the system, modules cannot be selected and the buttons ignore mouse events. If a user has only access to the monitoring side, modules cannot be selected and the buttons are hidden because the user does not have access right to modify the XML configuration of the system, which enabling and disabling a module would do.

The configured CAN buses of the system are shown in the middle. A CAN bus shows a

user configured name on it so that it is easier to recognize the bus. All the connections from the CAN channels of the modules to the CAN buses of the system are drawn with a narrow white line. MCM 1 has three CAN channels that are connected to a bus and the fourth configurable, but no connected channel is shown as a small white circle. The circle is barely seen on the module, but it is clearly seen on a full sized window. Both of the CAN channels of IOM 5 are in an error state. If the mouse cursor is hovered over one of the CAN channels, the following tooltip is shown: “CAN(1): Off state\n CAN(2): Off state”, where \n is a line break. The state of the module is set to offline because the state cannot be received by any other module or the PC application. If MCM 2 would have the first two CAN channels in an error state, the first two lines of the tooltip would be the same, but the following lines would be added to the shown tooltip: “CAN(3): Normal \n CAN(4): Not connected”. Now the real state of the module would be shown because there is still a working connection to a CAN bus and state information can be received. The PC application cannot receive the state information of the CAN channels after the pre-connection is established, and the figure was modified in order to describe the design for the CAN channel errors at this point.

All the tooltips are animated when mouse cursor is moved on top of an element with a tooltip. The tooltip animation is a simple scale and out elastic animation: the tooltip begins to expand from the middle of the element, and when it reaches its full size, it expands a bit more and then bounces back to full size. Whenever a mouse cursor is set outside the element, the tooltip is hidden with an animation that scales the tooltip back to zero in the middle of the element. An example tooltip is shown in full size where the mouse cursor is in the figure. The same background is used for all shown tooltips, and it always covers at least the element where it is shown. If a text does not fit in the background of a tooltip, the background is expanded to fit the text inside it.

The current state of a module is shown as a text in the middle of the module image. Section 3.1 discussed that some of the states should be animated and a symbol to represent a state should be added. As it can be seen, there is no room for such elements in a module figure. Several static and animated symbols in different locations were tested, but this just cluttered the module figure.

The state of a system is not shown in the module level, because there is no need for it. The reason is that the module level is entered via system level where this information is already shown. However, it could have been shown next to the name of the system or next to the title. The status bar shows the current state of the system. Figure 5.2 presents the

possible states of the system and their respective tooltips. The tooltip in the status bar is not animated, because it is part of the existing status bar that uses Qt Widgets as a user interface technology. The state of a system is composed of the states of the modules used in the system. Modules that are disabled or in the offline state are not taken into account when composing the system state. The state of the system is offline if all the modules are disabled, the modules are in the offline state or the PC application has not established the pre-connection to the automation system. If any of the modules in the system is currently executing bootloader, the state of the system is bootloader. The state of the system is stopped if at least one of the modules is in the stopped state and none of the modules is executing bootloader. The starting up state is used if at least one of the modules is in the initialization state or in the pre-operational state and none of the modules is in the stopped state or executing bootloader. The state of the system is test state if none of the modules are in the previously mentioned states and at least one of the modules is in the test state. The state of the system is operational only if all the modules in the system are in the operational state.

	Tooltip
system	System status: Offline
system	System status: Bootloader
system	System status: Starting up
system	System status: Operational
system	System status: Stopped
Test state	System status: Test state

***Figure 5.2** States of the system in the status bar and their respective tooltips*

If the PC application is connected to the system, the connection is shown in the top left corner as a symbol. The symbol shown in Figure 5.1 represents a disconnected symbol. When it is hovered, tooltip shows “Disconnected” or “Connected” depending on the connected status. The symbol for connected status is shown in Figure 5.3.

If an upload is ongoing, instead of an upload progress bar discussed in Section 3.1 and shown in Figure 3.2, an animated symbol is shown next to the state text that shows “Uploading” in this case. The upload symbol and the text can be seen in Figure 5.5. The animation is done by moving the orange triangle from top to bottom and repeating that infinitely. The currently used upload dialog is not hidden when the uploading begins, and the system view cannot be used to open the dialog. It was decided that indicating an ongoing upload in the system view is sufficient because the upload progress would still be viewed from the currently used dialog.

5.2.2 List view and multi-system support

The list view is the default view when opening the system view. Figure 5.3 presents the list view with two systems. Modules are grouped in the list view and module group name separates different types of modules. The same three columns are shown as in the currently used system view, and they represent the same things as described in Section 2.1. The color scheme used matches with the grid view. Above the list of modules, there is a rectangle that shows if the system is connected, the name of the system and the current state of the system. There is no room to display the state of a system as a text, so a tooltip is shown when hovering the state symbol. The border of the rectangle uses the same color as the connected or disconnected symbol to improve user perception of whether the PC application is connected to the system.

***Figure* 5.3** *List view presentation of the system view*

The “Show only headers” button is used to resize the list view. When it is pressed, all the module information is hidden by resizing the view to show only the rectangles presenting the systems. Height of the window is always changed to a fixed value, but the width depends on the number of logged in systems. The button goes to an active state once it is pressed, and its background is changed to the same color as used in the connected symbol. If a user resizes the view manually, the state is deactivated automatically. Another way to deactivate the state is to press the button again which restores the size of the window to the size before pressing the button.

The grid view can be activated from the top right corner. To activate the list view from the grid view, the grid view has a button at the same location. System level is shown as the default view in the grid view. Figure 5.4 presents the system level with four systems, as it is the maximum number of simultaneously used systems, as described in Section 3.2.

Figure 5.4 *System level presentation of the grid view*

The system level shows the basic information of all systems to which a user has logged in. The name of a system is shown in the top part of the system rectangle. The connection

status and the current state of the system are shown with a text in the bottom part inside another rectangle. The symbols for states are the same as in the list view, but hovering over a symbol does not show a tooltip, because it is already shown next to the symbol as a text. Other states that are not shown in Figure 5.4 and their respective symbols are shown in Figure 5.5. An extra outline is added into a system rectangle if the PC application has established a connection to the system. Example system 3 shows “Starting up” state, which represents the starting up state described in Section 5.2.1. The symbol next to the state is animated. The animation is a rotation animation where the orange segment is rotated around. This indicates that the system is not ready and there are some initialization, synchronization or other tasks to be performed in the automation system.

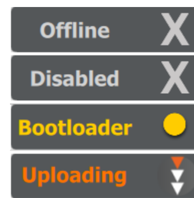

Figure 5.5 *Additional states and their respective symbols used in system and module group levels*

When a system is hovered, a shadow is used to make the rectangle look like a clickable item. Example System 3 shows the used shadow. The shadow is shown as the innermost border of the system borders. When a system is clicked, a transition to the currently active level is performed. The transition animation expands the clicked system to match the system view size and hides other systems at the same time. The module level is shown by default. The module group level is shown if it has been previously selected for the system. In Figure 5.1 there is a button at the bottom left of the view, and when it is pressed, a transition back to the system level is performed. The transition animation resizes the system back to the original size and then shows the other systems in the system level.

5.2.3 Usability and performance

Section 3.3 discussed the handling of a large number of modules in different ways. In Figure 5.1, some of the modules are placed above and some below the other modules. The minimum width and height of the module level are calculated to fit 22 modules. It was mentioned that in a real system the module count varies from 14 to 20, so some extra modules can be added without extending the minimum size of the system view. On the module level, modules are organized according to module groups. In the figure,

this grouping can be seen on the right side, where all IOMs are close to each other. An alternative approach would be to set IOM 5 above MCM 1 and MCM 3, and IOM 6 below MCM 2 and MCM 4 but it is easier to view the same types of modules close to each other.

Zoom control buttons and a slider are shown at top left corner in the figure. A zoom-out action is performed if the magnifying glass button with a minus sign is pressed. The other button with a plus sign performs a zoom-in action. These buttons increment or decrement the scale of the view by a fixed number. The slider is used to scale the view with a floating number. If the scale of the view causes the view to exceed the size of the window, horizontal and/or vertical scrollbar(s) appear. Another way to view different content inside the view is to press and hold the mouse cursor in the background area and then move the cursor. This is called *flicking*, and it is a widely used technique with touch screens.

Figure 5.6 presents the module group level described in Section 3.3. “View by groups” button is active and next to it, a button that opens a view used to select modules that should be grouped. Below the module group selection view, there are two buttons that are used to deselect grouped modules or to select all modules to be grouped.

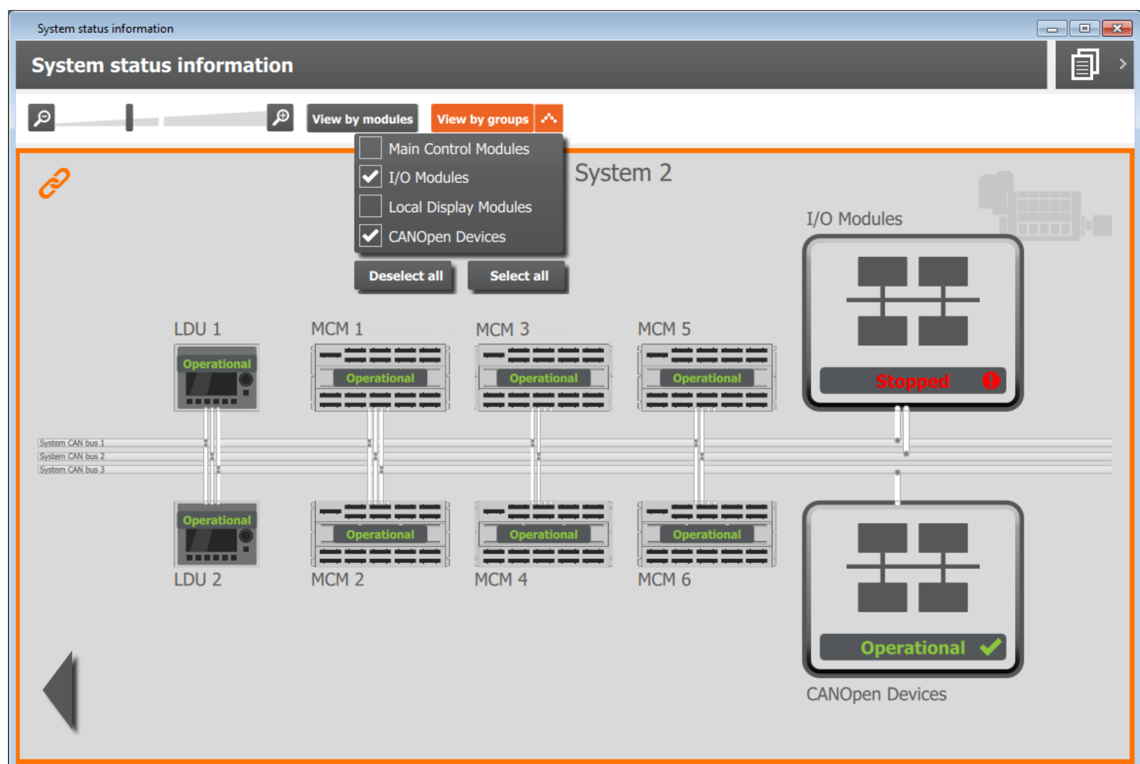

***Figure* 5.6** *Module group level presentation of the grid view*

Module groups are presented almost exactly like systems on the system level: only the symbol for connected status is not shown, because it is not related to a module group. The grid view enters the module level of the module group when a module group is clicked. The transition is animated in the same way as the transition from the system level to the module level described in Section 5.2.1. However, modules and other module groups are hidden instead of other systems. The same button used for navigating back from the module level to the system level is used to navigate back from the module level of the module group. The transition back to the module group level is also like the transition back to the system level, except modules and module groups are shown instead of systems. Figure 5.7 summarizes the navigation in the system view. It should be noticed that switching between the module level and the module group level is not possible when the user has navigated onto the module level of a module group. This is prevented by hiding the "View by modules" and "View by groups" buttons.

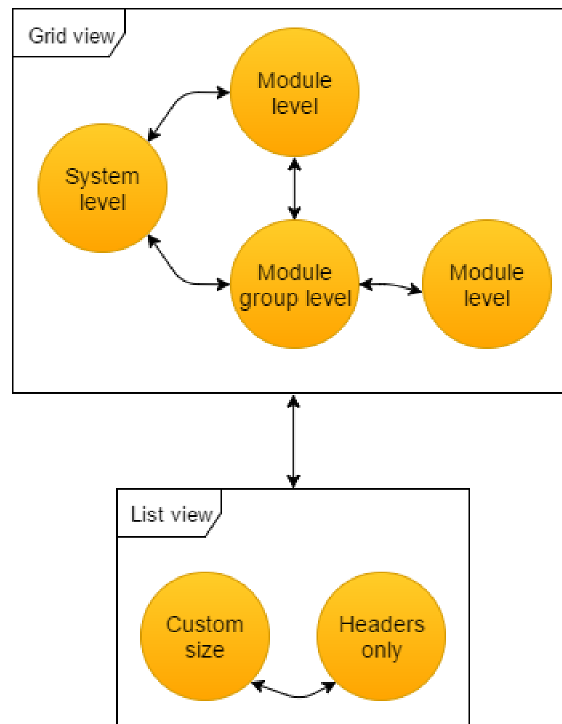

***Figure* 5.7** *Navigation in the system view*

All the content that a user has not previously seen is asynchronously created when the user navigates in the system view. A loading animation is shown in the center of the system view while the new content is being constructed. The loading animation is done by rotating an image in GUI thread while the constructing of a new view is executed in another thread. The loading image is shown in Figure 5.8.

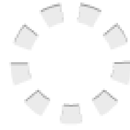

Figure 5.8 *Loading image used for animation when views are being constructed*

The window positions of the grid view and the list view are stored in a single QML document. The document remains persistent in memory as long as the PC application is running. When switching from the list view to the grid view, the size and position of the list view is stored and a new size and position for the window are set according to the previous values of the grid view. The same procedure is performed also when switching from the grid view to the list view.

The performance of the system view is measured with profiling tools provided by Qt Creator. If some functions in QML documents consume relatively too much time, these functions are taken into closer inspection to find the cause. Execution times of the critical functions are measured with QTimer class in C++ parts of the system view. To open the system view fast, the system view plugin is loaded immediately after a user first logs in to a system. Memory consumption is measured with the resource monitor provided by the Windows operating system.

5.2.4 Support for CANopen devices

Figure 5.1 presents two CANopen devices, WAGO 1 and WAGO 2. WAGO 1 can be partially seen behind the tooltip. If the PC application is not connected to a system, a tooltip for a CANopen device shows “CANOpen devices can only be monitored when connected to system.”

Section 4.3 explained that to be able to monitor a CANopen device, another module has to be configured to monitor the heartbeat of the CANopen device. Also, a DC code that stores the state of the CANopen device in a module has to be selected. When the PC application is connecting to a system, the configuration of CANopen devices is checked. If a module is not selected to monitor the heartbeat of a CANopen device, the tooltip shows “Heartbeat monitoring is not enabled in configuration.” If a DC code is not selected, the tooltip shows “Error code for heartbeat monitoring is not enabled in configuration.”

If a module that monitors the heartbeat of a CANopen device is not connected to the same bus as the CANopen device, the CAN channel of the module is not in the error

active state or the module is not in the operational state, the tooltip shows “Monitoring module is in a non-operational state or disconnected from CAN bus.” If a CANopen device is configured properly and the state of the CANopen device is received by the PC application, the tooltip shows “Node in <state> state”, where <state> is one of the following: stopped, pre-operational or operational. All the tooltips that are related to CANopen devices or modules, are shown in the “Status” column in the list view.

5.3 Architecture of the system view

The system view is now designed at a general level. In the project, C++ classes use letter C prefix before a class name, and interfaces use letter I prefix before an interface name. All the related C++ classes and the interface of the system view are in Plugins::GUI:SystemView namespace. Object-oriented programming (OOP) is used along in Qt framework, and it is already used elsewhere in the project, so the methods described in OOP concept are also used for the system view implementation.

To understand the architecture of the system view better, a well-known design pattern, Model-View-Controller (MVC), that Qt framework utilizes is introduced. Model is used to maintain the data, view is used to display the data maintained by the model and controller is used to handle events that are usually triggered from the view and impact the model in one way or another. The main idea in the MVC pattern is to keep the code related to each of the components as separate as possible. [6]

In Qt framework and Qt Quick, a modified version of this traditional MVC pattern is introduced. The pattern is called Model-View-Delegate (MVD), where the model and the view parts are the same as described for MVC pattern. The delegate simply describes how the data should appear in the view, and it can access both the model and the view directly. [21] In a Qt application, communication between the different parts of the MVD pattern is done with signal and handler event system of Qt. For example, if a view has a button and it is clicked, a *signal* is emitted, and it is responded to through a *signal handler* that could be in a related model or a delegate. The emitted signal triggers the corresponding signal handler that is connected to the signal. This communication between objects is made possible by Qt's Meta-Object System (MOS). MOS also provides run-time type information and a dynamic property system, and it is discussed more in Section 6.1. [20] [27]

Figure 5.9 presents the architecture diagram of the system view. Four different pack-

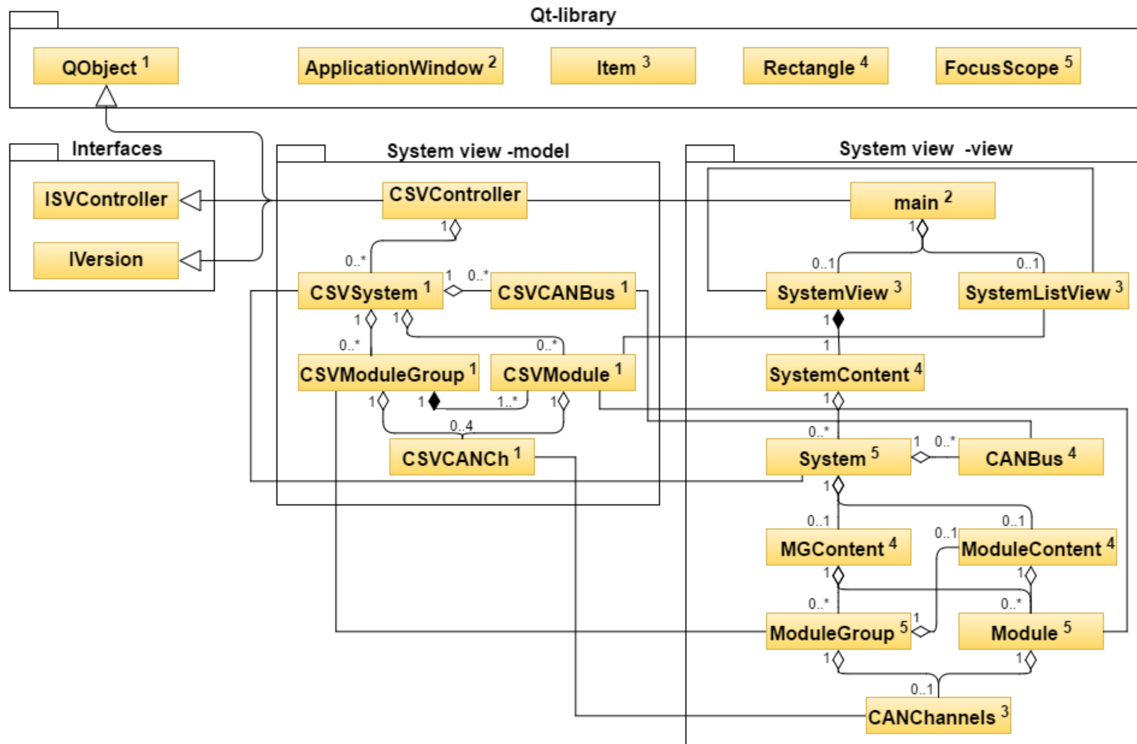

Figure 5.9 Architecture diagram of the system view

ages are shown: Qt-library, Interfaces, System view -model and System view -view. The Qt-library package has one C++ class, QObject, and four QML types: ApplicationWindow, Item, Rectangle and FocusScope. C++ classes that inherit QObject class are marked with a superscript "1" and QML documents, which inherit any of the previously mentioned QML documents, with corresponding superscripts "2", "3", "4" and "5". In the diagram, the abbreviation SV stands for SystemView and abbreviation MG stands for ModuleGroup. The interfaces package has two interfaces: ISVController and IVersion. **ISVController** is the interface provided to other components in the PC application, and it is the only way to access the system view plugin outside of the plugin itself. **IVersion** interface is used to store version related information of a plugin that inherits it.

The "System view -model" package has five C++ classes: CSVController, CSVSystem, CSVModuleGroup, CSVModule, CSVCANBus and CSVCANCh. **CSVController** class is the main class that provides the ISVController interface to other components in the PC application. **CSVSystem** class stores the data of a single logged in system required by the view. **CSVCANBus** stores the data of a single CAN bus configured for the system required by the view. **CSVModuleGroup** class contains the data of a single module group composed of CSVModule classes, and **CSVModule** contains the data of a single

module of the system. **CSVCANCh** class stores the data of a configured CAN channel for a module.

Only the most important QML documents are included in the "System view -view" package. Qt Quick applications require a main document that is loaded first [3]. "main" document is the main QML document, and it is loaded at first. **SystemListView** is the main document for the list view, and it represents all logged in systems with other QML documents, which are responsible for user interaction described in Section 5.2.2. SystemListView uses CSVSystem classes received from CSVController class as a model.

SystemView is the main document for the grid view. **SystemContent** document is responsible for constructing the plain layout of the grid view and handling of different events triggered via SystemView. **System** document presents the UI of a single logged in system by using CSVSystem class as a model. **CANBus** document visualizes a single configured CAN bus of the system by using CSVCANBus class as a model. If the module group level described in Section 5.2.2 is active, **MGContent** document is loaded, and if the module level is active, **ModuleContent** document is loaded. Both documents handle their respective presentation of the levels. MGContent document uses ModuleGroup and Module documents depending on the selected module groups to be shown. **ModuleGroup** document presents a single module group that consists of the same type of modules by using CSVModuleGroup class as a model. **Module** document handles user interaction with a module and presentation of a single module by using CSVModule class as a model. **CANChannels** document presents all the configured CAN channels of a module or a module group. CANChannels document uses CSVCanCh classes as a model. It should be noted that a ModuleGroup document aggregates also of a ModuleContent document. ModuleContent is loaded for ModuleGroup if a user navigates into the module level of a module group.

6. IMPLEMENTATION OF THE SYSTEM VIEW

This chapter discusses the technical implementation of the system view. The implementation of the system view is divided into three parts: C++ classes, QML documents and JavaScript files. The C++ classes provide the interface and its implementation used by the other components in the PC application, and models for the QML documents. The QML documents are using these models to declare the currently logged in systems and their status information in the system view. In addition, the QML documents declare delegates that are responsible for user interaction. The JavaScript files are used by QML documents in order to asynchronously construct different parts of the system view, and to modify the current layout of the system view.

Table 6.1 presents the distribution of the code base of the system view implementation as lines of codes counted with the cloc program. Comment and code columns show the number of lines programmed rounded to the nearest tenth. As can be seen on the table, most of the code base consists of QML and JavaScript code. This is because the related C++ code is providing only the models that are used by the QML/JavaScript code, and the functionality of the system view is more focused in the UI. The program code programmed for other plugins besides the system view plugin is omitted from the table.

Table 6.1 *Distribution of lines of program code of the system view implementation*

Language	Files	Comment	Code
C++ Header	7	850	520
C++	6	200	1550
QML	41	840	3530
JavaScript	4	130	610
Summary	58	2020	6210

Microsoft Visual Studio (MVS) is used by default as a development environment in the project, and its compiler, MSBuild, produces the necessary DLL files for the PC application. Even with Qt add-in for MVS, MVS does not handle QML files very well. Because

QML files are programmed for the system view, Qt Creator is used for development. Another reason to use Qt Creator is that it also provides debugging and profiling of QML documents. Profiling simply means analyzing QML and JS execution in QML document [22]. The QML documents and the JS files related to the system view were profiled several times, and the profiling helped to detect performance problems in the program code and the problems were fixed immediately. Qt Creator uses MinGW compiler by default, but with some configuring, custom builds can be used. Qt Creator was configured to use MSBuild with the same build flags and options as in the MVS environment.

The memory consumption of the system view was measured with the resource monitor provided by the Windows operating system. The memory consumption tests were comprehensive, including stress tests and long-term tests. Some memory leaks, which were related to the ownerships of instances of Qt classes, were found and fixed.

6.1 Descriptions of C++ classes

This section describes the responsibilities of the C++ classes related to the system view. All the C++ classes, except ISystemViewController interface and CSystemViewController class, represent a model that is used by QML documents. The models implement the model described in the MVD pattern.

ISystemViewController interface

The **ISystemViewController** interface provides the public interface used by the other software components in the PC application. The following main functionalities are provided by the interface: opening the system view, adding/removing a system to/from the system view, updating the current state of a module, retrieving the current state of a module/system and emitting a signal if the state of a system is changed. CSVController implements the interface, and the plugin is loaded with a static *LoadExtensionObject* function call provided by the AddinManager class. **AddinManager** keeps track of loaded plugins, and if the plugin is already loaded, it returns a pointer to it. Otherwise, a new instance of the loaded plugin is created. This procedure is very close to the Singleton design pattern, in which only one instance of a class exists in an application and it can be accessed globally [6].

The system view is opened by double-clicking the corresponding status bar item. An instance of CGUI class, which implements IGUI interface, receives the mouse event and

calls the function that opens the system view. Currently, there is no use case for opening the system view without user interaction, but one possible scenario could be opening the view automatically when an upload is started.

When a user logs in to a system, the system is added in CSVController. All added systems, which are pointers to an interface ISystem, are stored in a QHash class member variable. ISystems are used as keys for corresponding model classes, CSVSystem classes, which are initialized at the same time. ISystem is used as a key in the hash to partly implement the multi-system support described in Section 5.2.2. Because of the multi-system support, all the functions declared in the ISVController interface, except opening the view, require ISystem as a parameter. When the user logs out from the system, the system is removed from CSVController, and if the system view is open when the user logs out from the last system, the view is closed.

The ISVController interface declares an enumeration class, named *STATE*, that represents the possible states of a module. The values of the enumeration class are used in the system view plugin, and in other plugins that are interested in the current states of the modules. Another enumeration class, named *SYSTEM_STATE*, is declared to represent the possible states of a system. A value of the enumeration is used as a function parameter when CSVController emits the state change of the system, or as a return value when other plugins explicitly query the state of a system.

CSystemViewController class

The **CSystemViewController** class inherits and implements ISVController and IVersion interfaces. The implementation of the IVersion interface provides only version information related to the system view plugin. In addition to functionalities described previously in the ISVController interface, CSVController is responsible for creating instances of QQmlEngine and QQuickWindow classes. **QQmlEngine** is a C++ class that is responsible for interpreting QML documents and JavaScript files in run time. **QQuickWindow** is a C++ class that is used as a base window for the system view. Another interesting class that is used by QQmlEngine is QQmlIncubationController class. **QQmlIncubationController** is a C++ class that is responsible for creating QML objects asynchronously in run time. QQmlIncubationController can be inherited in order to create a custom controller, or the default QQmlIncubationController provided by QQmlEngine can be used. Normally, when a QML document is constructed, contents of the document are built in the caller thread, but with the QQmlIncubationController, QML documents are built in another thread to prevent the GUI from freezing. [3]

When CSVController receives information about a state change of a module, it converts the state to *STATE* enumeration value. The function that converts the state returns also a tooltip as QString, to be used with the state. The tooltip and the enumeration values are then processed in the corresponding CSVSystem instance where the module belongs.

As has been discussed, the PC application is used also for configuring the underlying automation system and its modules. The configuration of the system is stored as an XML file, and to update the configuration, the user has to trigger a saving procedure which does the actual modifications to the XML configuration file. To update the system view accordingly, a signal from the plugin that is responsible for doing the actual saving of the XML configuration is connected to CSVController. When this signal is received, CSVController deletes the corresponding model, CSVSystem, which is used by QML documents and creates a new one. This is a coarse way to update the view, but it is much easier to implement because CSVSystem initializes modules, CAN buses and other configurable parts related to the system from scratch. The initialization procedure is described in CSVSystem description. A more fine-grained way of updating an instance of CSVSystem would be to signal only the modified parts of the configuration during the saving procedure, but it would require more complex implementation. Because all the views used in the PC application and the configuration are based on XML, it is not trivial to parse the changed parts. Execution time on reconstructing an instance of CSVSystem was measured with QTimer: in debug without compiler optimizations, reconstruction was executed in less than 50 ms, which is relatively fast in comparison with the saving process that is executed in five to seven seconds.

CSystemViewSystem class

The **CSystemViewSystem** class represents a model that is used by QML documents. When an instance of CSVSystem is constructed, it retrieves the information of the related system from the current XML configuration of the system by using XPaths[1]. During the initialization process, modules, CANopen devices, CAN buses and the CAN channel connections of the modules are read from the XML configuration with XPath queries. The CAN buses of the system are initialized first: the name and the baud rate of the bus are read from the XML configuration. For each CAN bus read from the XML configuration,

[1]XPath makes accessing an XML document easier. In practice, XPath provides a tool for a programmer to locate XML content within an XML document. When considering locating data in an XML document, XPath functions perform faster than functions that are reading the document sequentially because the XPath functions know about the nodes in the document. The nodes are chunks of information encapsulated within the XML document. [17]

an instance of CSystemViewCANBus class, which represents a configured CAN bus, is created. After this, the modules of the system are initialized. During a module initialization, the name, the maximum number of CAN channels for the module and the data that represents whether the module is enabled or disabled are read. All the CAN channels that are configured for a module and connected to any CAN bus of the system are initialized at the same time. The connection of the CAN channel to any CAN bus is checked with an XPath query that contains the IDs of the configured CAN channel and the configured CAN buses.

Qt provides a dynamic property system that is based on Qt's Meta-Object System. In the property system, Q_PROPERTY macro is used to declare variables in a class that inherits QObject class. The variables behave like class data members, but they have additional features accessible through the Meta-Object System. As an example property, one of the properties declared in CSVModule class is presented next. [24]

```
{
   Q_PROPERTY(bool enabled READ enabled WRITE setEnabled
              NOTIFY enabledChanged)
signals:
   void enabledChanged();
private:
   bool m_bEnabled;
}
```

The **READ** accessor function, named *enabled*, is used to read the current value of the property named *enabled*. The **WRITE** accessor function, named *setEnabled*, is used to set the property value, and the **NOTIFY** signal, named *setEnabled*, is emitted whenever the value of the property changes. The signal has to be emitted explicitly if the value changes but if a **MEMBER** type is declared in Q_PROPERTY macro, the signal is emitted automatically by MOS. The property system also declares many other types that could be used with the Q_PROPERTY macro, but the types are optional and not used in the implementation of the system view. [24] The value of the *enabled* property is changed from the enabled/disabled buttons, as described in Section 5.2.1, by clicking the buttons. When CSVModule instance receives the signal from Module document about the change, CSVModule emits its own signal. This signal is received by the CSVSystem instance where the CSVModule belongs. The CSVSystem object then modifies the XML configuration

of the system to change the value of the node that represents the enabled/disabled state of a module.

The CSVSystem class declares many properties that are used mainly by the System QML document. For example, a boolean property declared in a similar way as the *enabled* property, named *configAccess*, is used to represent the value of the user access right to the configuration side of the PC application. The value is constant, and it is received from another plugin of the PC application. The value of the property is used to hide, for example, the buttons that represent enabling and disabling modules.

CSystemViewMG class

The **CSystemViewMG** class implements the model used by ModuleGroup QML document. It represents a module group model, and it stores only the same type of configured modules that the CSVModule class represents. While initializing an instance of the CSVMG class, the CAN channels for the instance are constructed. The CAN channels of CSVMG are composed of all the CAN channels of the CSVModule instances that are part of the CSVMG instance in a way in which the first CAN channel of CSVMG represents all the first CAN channels of the CSVModule instances.

For each CAN channel constructed for CSVMG instance, an instance of CSystemView-CANCh class is created. **CSVCANCh** class represents a CAN channel for both the CSVMG and the CSVModule classes. If an instance of CSVCANCh is used by an instance of CSVMG, the instance of CSVCANCh stores the CAN channels of the CSV-Module instances that represent the same CAN channel, for example, the first CAN channel. The most important properties declared in the CSVCANCh class are *tooltip* and *state* properties. The value of the tooltip property, which is QString type, represents a single row shown in the UI for the CAN channels, as described in Section 5.2.1 and shown in Figure 5.1. The value of the tooltip of a CSVCANCh instance is set to non-empty string if the value of the state property, which is an enumeration class with CH_OK and CH_OFF values, is changed. For example, if an instance of CSVMG represents an MCM module group and one of the first CAN channels of MCMs transitions to the bus off state, the value of the state property of corresponding CSVCANCh is changed to the CH_OFF value. Then, CSVCANCh emits a *stateChanged* signal that is received by another instance of CSVCANCh, which represents the CAN channels of CSVMG, and the value of the first CAN channel of CSVMG is also set to the CH_OFF value. The CSVMG class and the CSVModule class have a property named *canChTooltip*, which represents the tooltip of all the configured CAN channels of CSVMG and CSVModule instances,

shown in the UI. The tooltip is composed of all the tooltips of the respective CSVCANCh instances, and if the tooltip of CSVMG or CSVModule is changed, *canChTooltipChanged* signal is emitted to notify the UI about the change.

Figure 6.1 presents the sequence diagram when the state of a CAN channel of a module changes to the bus off state. Before the value of the state is received by a CSVModule instance, the PC application has to be connected to the system. In addition, the CSVSystem instance has set the values of the CAN channels to be observed by calling *setPeriodicUpdate* function of the communication plugin. The communication plugin commands the automation system to transmit the values of the CAN channel states of a module, in order to emit a signal to the CSVModule instance whenever the state of a CAN channel is changed. The instance of CSVCANCh updates the corresponding tooltip of the CAN channel, and emits the *stateChanged* signal that changes the color of the CAN channel in the UI to red, as seen in Figure 5.1.

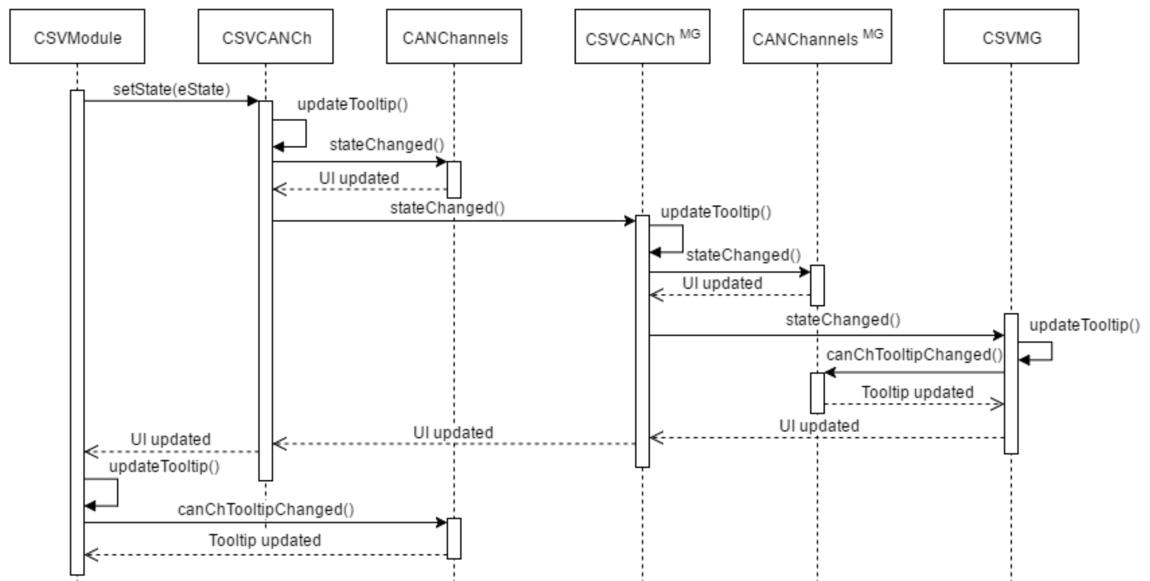

Figure 6.1 *Sequence diagram of the state transition of a CAN channel in the system view*

Another important property, named *showInView*, represents whether the module group should be shown as a group or as individual modules in the UI. The value of the property is changed from the UI, and it changes the layout of the UI, as discussed in Section 6.2.

CSystemViewModule class

The **CSystemViewModule** class represents a model used for CANopen devices and other modules in the automation system. The most important properties declared in the class are

name, moduleState, type and tooltip properties. *Name* represents the configured UI name of the module, *moduleState* represents the current state of the module, *type* represents the type of the module and *tooltip* represents the tooltip, which shows additional information about the current state, shown in the UI.

6.2 Descriptions of QML documents

This section describes the QML documents of the system view. The QML documents implement the view described in the MVD pattern. Only the most relevant QML documents related to the system view are briefly described.

main document

The **main** document declares the switching functionality between the list view and the grid view. In QML documents, properties are declared in a different way than in C++ header files. For example, the property that is presented next, named *gListViewActive*, represents whether the list view or the grid view should be shown in the system view. If the value of a property changes, Qt's Meta-Object System emits a signal about the change to components, which are connected to listen to the change. The prefix g before the property names means that the properties can be accessed globally from other QML documents that are included in the main QML document. In QML, all the properties declared in a parent QML document are accessible from the QML documents that are included in the parent QML document as QML types. [3]

```
{
   ...
   property bool gListViewActive: true
   property var gController: controller
   property var gSystems: gController.systems
   ...
}
```

The previously presented *controller* property is called a context property, and it is stored in the main document's property named *gController*. An instance of QQmlEngine has a root context that stores, for example, all the context properties set for it. The context properties in the root context are available in all the QML documents that are executed

by the QQmlEngine. The context properties provide an easy access to C++ objects from QML documents, but before a custom C++ object can be used in QML documents, the object has to be declared with a *qmlRegisterType* function call to the Qt's MOS. The context property, *controller*, is stored in the root context of the QQmlEngine used in the system view with, *engine->rootContext()->setContextProperty("controller", this)*, function call. The first parameter is the name that is used to refer to the context property in QML documents, and the second parameter is the object to which it refers. [3]

The *gSystems* property introduces another core feature used in QML documents, called property bindings. A property binding specifies a relationship between different object properties. The *gSystems* property is bound to the *gController.systems* property, which is declared in the header file of CSVController as a QVariantList type property. Whenever the *systems* property changes in value, meaning a new system is added or an old system is removed from the QVariantList type member variable container, an instance of CSVController emits a signal about the change. Then, the QML engine re-evaluates the binding expression and applies the new result to the *gSystems* property. If a property is bound to a property that is declared in a QML document, the QML engine automatically re-evaluates the binding expression without the explicit emission of a signal because the QML engine monitors the property's dependencies. [23]

The main document declares the functionality for the loading animation described in Section 5.2.3. Program 6.1 presents the QML component that declares the loading animation shown in the UI, while asynchronously constructing different parts of the system view. The property *visible* of the **Image** type and the property *running* of the **RotationAnimation** type are bound to a Boolean property, named *gLoading*, which is declared in the main document. Whenever a QML component begins creating content asynchronously for the system view, it changes the value of the property to true. Then, the image used in the animation starts rotating and is visible. Whenever the creation of the content is finished, the value of the property is set to false and the image is hidden and the rotation animation stopped.

```
{
    ...
    BusyIndicator {
        anchors.centerIn: parent
        style: BusyIndicatorStyle {
            indicator: Image {
```

```
                visible: gLoading
                source: "qrc:/SystemView/icons/loading"
                RotationAnimation on rotation {
                    running: gLoading
                    loops: Animation.Infinite
                    duration: 2000
                    from: 0; to: 360
                }
            }
        }
    }
    ...
}
```

Program 6.1. *QML declaration of the loading animation shown in the UI*

SystemView and SystemContent documents

The **SystemView** document implements the zooming functionality in the grid view and navigating back from the module level or the module group level. The zooming functionality is simply implemented with the **ScrollView** QML type that provides the scrollbars used in the view. In addition, it includes the SystemContent document and modifies its *scale* property by a fixed or a floating number depending on how the zooming is performed, as described in Section 5.2.3. The navigation is implemented by declaring a property named *viewNavigateBackHandler* that is a reference to a function that does the actual hiding and revealing of the different UI elements in the view. When activating the module level or the module group level, the value of the property is changed to the corresponding function that hides and reveals the UI elements when navigating back from the module level or the module group level. When the back navigation button is pressed, the function that the property currently refers to is called.

The **SystemContent** document is responsible for constructing the **System** documents that present the UI of currently logged in systems in the grid view. The instantiation of System documents is executed in the *system.js* JavaScript file where an instance is created by calling *systemComponent.incubateObject* function. The systemComponent is a **Component** type variable created with a *Qt.createComponent("System.qml")* function call, and it represents a template used to instantiate System documents asynchronously. The other functions in the *system.js* file are used to control the layout of the system level view.

System, ModuleContent, Module, ModuleGroupContent and ModuleGroup documents

The **System** document declares the UI of a single logged in system shown in the system view. It uses an instance of CSVSystem as a model and uses its declared properties to show, for example, if the PC application is connected to the system and what is the current state of the system. The System document has two Loader QML types: one for declaring the module level of the system and one for declaring the module group level of the system. The **Loader** QML type is the easiest way to dynamically load content in run time. It serves as a placeholder for the content that can be created when, for example, an event is triggered by a user via the UI. [3] When a user clicks on a system in the grid view, the Loader for the module level is activated, and it loads the ModuleContent document. The **ModuleContent** document asynchronously instantiates a Module document for each module of the system in the same way as the SystemContent document instantiates the System documents. The JavaScript file related to the ModuleContent document is named *module.js*, and in addition to creating the instances asynchronously, it handles the layout for the module level presentation. The **Module** document declares a single module of the system, and it shows the properties declared in the CSVModule class in the UI.

When the "View by groups" button, which was described in Section 5.2.3 and shown in Figure 5.6, is clicked, the Loader for the module group level is activated and it loads the ModuleGroupContent document. The **ModuleGroupContent** document asynchronously instantiates Module and ModuleGroup documents in the same way as ModuleContent instantiates Module documents. The JavaScript file related to the ModuleGroupContent document is named *moduleGroup.js*, and in addition to instantiating the documents asynchronously, it handles the layout for the module group level presentation. By default, if a module group consists of more than two modules, it is shown as a group, and an instance of ModuleGroup document is created. If name of the module group or the checkbox next to it, shown in Figure 5.6, is clicked, an instance of the ModuleGroup document is created, if it did not already exist. The **ModuleGroup** document declares the UI of a single module group in the same way as the System document declares a system. The ModuleGroup document has a Loader that loads a ModuleContent document if the module group is clicked.

SystemViewListView document

The **SystemViewListView** document declares the list view that is shown in the system view. The document declares a **Row** QML type and fills the row by using a **Repeater** QML type that is used to create similar items. The Repeater takes the number of cur-

rently logged in systems as a model and uses a **ColumnLayout** QML type to show the systems next to each other as presented in Figure 5.3. A single system is declared with a ListViewSystemHeader document and a TableView QML type. The **ListViewSystem-Header** document uses other smaller documents that declare a single UI element, like the connected/disconnected element, in the system header shown in the Figure 5.3. The *model* property of the **TableView** is bound to the *gSystems[index].modules* property that is a QVariantList type member variable declared in the CSVSystem class. The instance of TableView declares all the modules of a system as a list, and the rows shown in the UI are styled with delegate documents, bound to *delegate* properties of the QML types used in the TableView.

6.3 Showing the state transition of a module in the system view

The system view is mainly used to monitor the current state of the automation system. The process of a state transition of a module and transmitting the state information to the PC application were described in Chapter 4. Figure 6.2 presents a sequence diagram of processing the state transition of a module in the system view. The instance of the CGUI class is connected to a communication plugin that emits a signal if the state of a module is changed. The instance of CGUI calls *updateStatusInfo(ISystem *sys, quint16 id, quint8 state)* function provided by the ISVController plugin, and the instance of CSVController class converts the *state* parameter to a value of *STATE* enumeration, which was described in Section 6.1. The function that converts the value returns a tooltip as QString to be used with the state. Then, the CSVController finds the instance of CSVSystem from the QHash with the *sys* parameter, where the module belongs.

After the corresponding CSVSystem instance is found, CSVController calls *setModuleState(quint8 id, STATE state, const QString &tooltip)* function of the CSVSystem instance. The CSVSystem class has a QHash member variable for all the modules in a system, and the CSVSystem instance finds the instance of CSVModule from the QHash by *id* parameter and calls its function *setState(STATE state, const QString &tooltip)*. If the value of the *m_eState*, which represents the current state of the corresponding system module, member variable of the CSVModule instance is changed, it updates the member variables *m_eState* and *m_sTooltip*. First, the CSVModule instance emits the *stateChanged* signal. The Qt's MOS calls all the functions that are connected to the signal, and the QQmlEngine instance re-evaluates the values of the properties bound to the

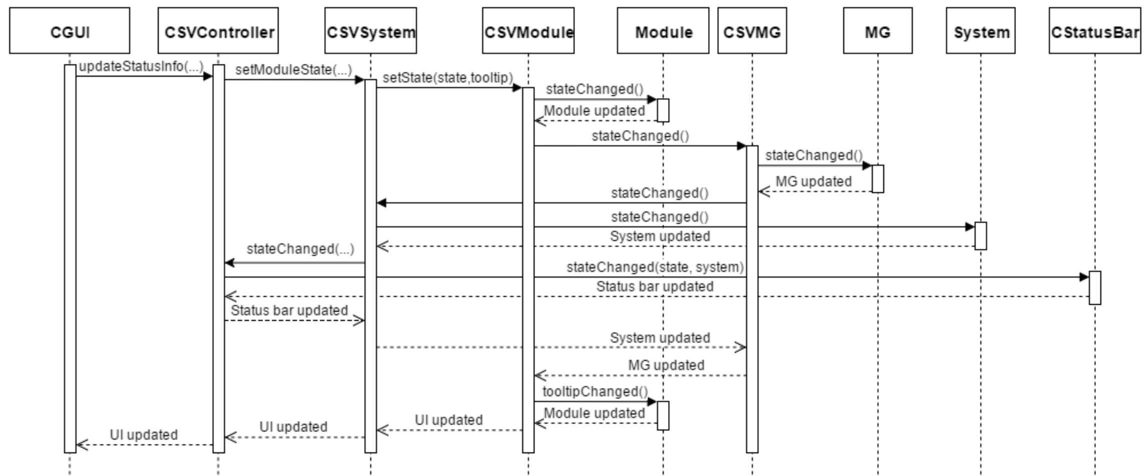

Figure 6.2 *Sequence diagram of the state transition of a module in the system view*

moduleState property, which is the property declared in the CSVModule representing the current state of the module.

An instance of the Module document uses the *moduleState* property in *states* property, which is a pre-defined QML property. The *states* property, which is declared in the Module document, contains all the possible states for the module shown in the UI, wrapped in *State* QML types. As an example, declaration of the bootloader state in the Module document is shown in Program 6.2. Whenever the value of the *moduleState* property is changed to *ISystemViewController.BOOTLOADER* value, the text and the color of the text shown on the module in the UI are changed by the **PropertyChanges** objects. The state of the Module document remains the same until the value of the *moduleState* property is changed. The System and the ModuleGroup documents declare the *states* property similarly to the Module document. The instances of the the ModuleGroup and the System documents updates their respective UI elements once the QQmlEngine has re-evaluated the properties changed in the instances of CSVMG and CSVSystem classes. The instance of CSVMG is connected to the *stateChanged* signal emitted by CSVModule, and the instance of CSVSystem is connected to the *stateChanged* signal emitted by CSVMG.

```
{
    ...
    states: [
        State {
            name: "bootloader"; when: moduleObject.moduleState
                                        === ISVController.BOOTLOADER
```

```
            PropertyChanges{target: centerTxt; color: blColor}
            PropertyChanges{target: centerTxt; text: blStr}
        },
        ...
    ]
    ...
}
```

Program 6.2. *Declaration of the bootloader state in the Module document*

When the value of the *m_eState* member variable, declared in CSVSystem, is changed, CSVSystem also emits *stateChanged(SYSTEM_STATE state, ISystem*)* signal, which is connected to CSVController instance, and the CSVController directly emits its own signal of the state change that matches with the declaration in the ISVController interface. The signal emitted by CSVController is used by other plugins, such as the status bar, to receive the changed state of a system.

7. EVALUATION

This chapter evaluates the set and realized goals of the thesis and discusses future improvements to the system view. Qt Quick user interface technology was chosen to implement the next generation of the system status view, and it proved to be very mature technology in terms of rapid UI development, ease of use and modern UIs. The customer was satisfied with the outcome of this thesis, and internal feedback was very positive.

7.1 Realized goals

All the goals that were set in Chapter 3 were realized. During the development, the current work was presented a few times for the customer to receive feedback, and the customer also had different variations of the system view in internal use before the final release. The current version of the system view was taken into use in December 2015. However, minor bug fixes and functionalities were implemented until March 2016. Wapice's employees had the system view in internal use for the whole implementation time, and a lot of advisable feedback and suggestions were given by the co-workers in the same project. Table 7.1 summarizes the set and realized goals of the thesis. The numbers shown in the table reference the section where the goals were described.

Table 7.1 *Set and realized goals of the thesis*

Number	Goal	Realized
3.1	Basic functionalities	Yes
3.2	List view and multi-system support	Yes
3.3	Usability and performance	Yes
3.4	Support for CANopen devices	Yes

Some functionalities described in Chapter 3 were implemented a bit differently from what was originally intended. The upload progress bar was replaced with an upload animation just to indicate an ongoing upload in the system view, and the currently used upload dialog

is still shown to indicate the detailed upload progress. Also the connection status and the state of a system shown in Figure 3.2 were separated as individual UI elements.

7.2 Future improvements

Currently, CPU and RAM usage are monitored with monitor windows that read the values of the DC codes mapped to the related system parameters. Values of the CPU and RAM usage of a module could easily be fetched by the system view in the same way as data is fetched for CANopen devices. Interpretation of CPU and RAM usage could be as simple as a percentage number, but also more advanced figures such as gauges could be used.

Section 5.2.1 explained that the system view can also be used to do some configuring of the system. Enabling and disabling modules and the possibility to select a module were implemented as an example to introduce the possible concept of using the system view also for configuring the system. Because the system view already takes into account different user access rights by disabling provided functionalities related to the configuration side of the PC application, it is quite straightforward to extend the configuration side of the system view. The system view could be used to add, remove and configure modules, to configure the CAN buses of the system and the CAN channel connections between modules and CAN buses. There could be an extensible side bar shown when a module is selected, and it would show all the configurable information of that specific module that normally would be configured from the XML based views in the PC application. A selected module could be deleted with a button that could be added next to the enable/disable button of a module. Modules could be added by right-clicking the canvas where the modules are currently shown. Making the CAN connections for a module could be as easy as a drag-and-drop event from a non-existing CAN channel to a CAN bus.

The system view could be used to start the actual upload process to the system. As mentioned in Section 7.1, the system view just shows an upload animation when the upload progress is ongoing. The upload functionality could be re-factored and enhanced to use the system view. Then the progress bars above/below modules would be a better option than the currently shown animation, and if a module was clicked, it could show the detailed information about the upload progress for the module.

When using the system view for monitoring the status of a system, it does not provide enough information to do detailed diagnostics of modules that are not functional. Chapter 4 discussed that if a module encounters a failure from any source, a log message is sent to

the PC application. The log window is still the main window used to see why the system is not working as configured. To improve the system view, some kind of middleware component between the log window and a message handler could be implemented. This component could parse diagnostics information and utilize the system view to show the information directly as a module related information in the view. The log window can easily be flooded with all the other log messages sent by the system, so it would be necessary to parse only the critical information that informs the end user what is wrong in the system.

Another interesting improvement could be to utilize 3D graphics. A system could be presented as a 3D model that could show the actual automation system. This would remove the need for the 2D navigation system that was implemented. A 3D model could be viewed in different angles and the modules could be placed on the actual location in the automation system, which would help the end user to locate a failed module immediately.

8. CONCLUSION

This thesis described a high level structure of an industrial automation system and explained the automation system diagnostics with a few detailed examples. The design and implementation of the system view are comprehensively discussed and the realized system view has been put to use by the customer.

The system view was designed and implemented during the autumn of 2015. The budget set by the customer translates to around 600 hours of work, and the main purpose was to make the basic functionalities work flawlessly and to provide an easily maintainable code base for future work. One of the most challenging parts of the actual implementation was to integrate the system view plugin with other plugins in the PC application. During the development, several problems caused by using multiple QQmlEngines in one project were encountered and solved.

8.1 Thesis process

The thesis process was started with the design and implementation of the system view. The initial concept described in Chapter 3 was extended in the autumn of 2015, and a proposal to the customer was presented. The customer set the budget, and after that, designing and prototyping the system view was started. I saw this as a great opportunity to improve my competence in C++/Qt/QML area, and I was really excited about the work. It could be thought of as a mini-project required by the customer, and I could work independently for the whole autumn. Obviously, my co-workers advised me in many things in the early phases, but it was nice to realize that at the end of the implementation I could share my knowledge of the things learned during the whole implementation process.

The writing process started in March 2016, and it set quite a strict schedule for the writing part of this thesis. This schedule helped me to write the thesis intensively, but perhaps there could have been some comparison to other existing automation systems that use some sort of status monitoring. The most problematic part of the writing process was to

limit the scope and to discuss only the relevant parts of the automation system. In the end, the thesis process was entirely successful: the design and implementation of the system view is thoroughly described, and the diagnostics of the related automation system and background of it are comprehensively discussed.

BIBLIOGRAPHY

[1] P. Aksoy and L. Denardis, *Introduction to information technology in theory*. Boston, Mass, 2008, p. 80, ISBN: 9781423901402.

[2] B. Benz and J. Durant, *XML Programming Bible*. Wiley Publishing, Inc, 2003, pp. 3–7, ISBN: 0764538292.

[3] J. Bocklage-Ryannel and J. Thelin, *Meet Qt 5*, © 2012–2014, Available: http://qmlbook.github.io (accessed on 25.4.2016).

[4] DENX Software Engineering, *Das U-Boot – the Universal Boot Loader*, © 2002–2016, Available: http://www.denx.de/wiki/U-Boot (accessed on 4.4.2016).

[5] Digia Plc, *Qt Language Bindings*, © 2016, Available: https://wiki.qt.io/Category:LanguageBindings (accessed on 25.4.2016).

[6] A. Ezust and P. Ezust, *Introduction to Design Patterns in C++ with Qt 4*. Prentice Hall, 2007, p. 361, 392, ISBN: 0131879057.

[7] J. Ganssle [editor], S. Ball *et al.*, *Embedded Systems: World Class Designs*. Newnes, 2007, pp. 247–249, ISBN: 9780750686259.

[8] Kvaser, *CAN Protocol Tutorial*, © 2014, Available: https://www.kvaser.com/can-protocol-tutorial/ (accessed on 17.4.2016).

[9] J. Laurikainen *et al.*, *State Machine, Technical Specification*, Wapice, 2010–2016, Wapice's internal material.

[10] R. Loyd, *Electrical Raceways & Other Wiring Methods*. Delmar Cengage Learning, 2004, p. 51, ISBN: 9781401851835.

[11] Microsoft, *What is a DLL?*, © 2016, Available: https://support.microsoft.com/en-us/kb/815065 (accessed on 27.4.2016).

[12] M. Natale, H. Zeng, P. Giusto, and A. Ghosall, *Understanding and Using the Controller Area Network Communication Protocol*. Springer-Verlag, 2012, pp. 1–13, 22–25, ISBN: 9781461403135.

[13] National Instruments, *The Basics of CANopen*, © 2016, Available: http://www.ni.com/white-paper/14162/en/ (accessed on 18.4.2016).

[14] R. Oshana and M. Kraeling, *Software Engineering for Embedded Systems*. Newnes, 2013, pp. 26–27, ISBN: 9780124159174.

[15] O. Pfeiffer, A. Ayre, and C. Keydel, *Embedded Networking with CAN and CANopen*. Copperhill Media Corporation, 2008, pp. xv–xviii, 30, 83–85, 101, ISBN: 9780976511625.

[16] R. Rischpater, *Application Development with Qt Creator*. Packt Publishing, 2013, pp. 7–9, 45, ISBN: 9781783282319.

[17] J. Simpson, *XPath and XPointer*. O'Reilly & Associates, Inc., 2002, pp. vii, 11–12, ISBN: 0596002912.

[18] The Qt Company, *The Framework & Tools*, © 2016, Available: http://www.qt.io/qt-framework/ (accessed on 25.4.2016).

[19] The Qt Company, *How to Create Qt Plugins*, © 2016, Available: http://doc.qt.io/qt-5/qtwebengine-overview.html (accessed on 27.4.2016).

[20] The Qt Company, *The Meta-Object System*, © 2016, Available: http://doc.qt.io/qt-5/metaobjects.html (accessed on 5.5.2016).

[21] The Qt Company, *Models and Views in Qt Quick*, © 2016, Available: http://doc.qt.io/qt-5/qtquick-modelviewsdata-modelview.htmll (accessed on 5.5.2016).

[22] The Qt Company, *Profiling QML Applications*, © 2016, Available: http://doc.qt.io/qtcreator/creator-qml-performance-monitor.html (accessed on 27.4.2016).

[23] The Qt Company, *Property Binding*, © 2016, Available: http://doc.qt.io/qt-5/qtqml-syntax-propertybinding.html (accessed on 12.5.2016).

[24] The Qt Company, *The Property System*, © 2016, Available: http://doc.qt.io/qt-5/properties.html (accessed on 10.5.2016).

[25] The Qt Company, *Qt Application Development*, © 2016, Available: https://www.qt.io/qt-for-application-development/ (accessed on 25.4.2016).

[26] The Qt Company, *Qt WebEngine Overview*, © 2016, Available: http://doc.qt.io/qt-5/qtwebengine-overview.html (accessed on 25.4.2016).

[27] The Qt Company, *Signal and Handler Event System*, © 2016, Available: http://doc.qt.io/qt-5/qtqml-syntax-signals.html (accessed on 5.5.2016).

[28] The Qt Company, *User Interfaces*, © 2016, Available: http://doc.qt.io/qt-5/topics-ui.html (accessed on 25.4.2016).

[29] W3C, *HTML & CSS*, © 2016, Available: https://www.w3.org/standards/webdesign/htmlcss.html (accessed on 26.4.2016).

[30] W3C, *JavaScript Web Apis*, © 2016, Available: https://www.w3.org/standards/webdesign/script.html (accessed on 26.4.2016).

[31] K. Yaghmour, J. Masters, G. Ben-Yossef, and P. Gerum, *Building Embedded Linux Systems: Edition 2*. O'Reilly Media, Inc., 2008, pp. 273–277, ISBN: 9780596555054.

[32] R. Zurawski [editor], A. Valenzano, and G. Cena, *Networked Embedded Systems*. CRC Press, 2009, pp. 15-1–15-38, ISBN: 9781439807613.