TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

VILLE MYLLYNEN

DESIGN AND IMPLEMENTATION OF A MESSAGE STANDARDIZATION TOOL

Master of Science thesis

Examiner:
professor Hannu-Matti Järvinen
Examiner and topic approved by the Faculty Council of the Faculty of Computing and Electrical Engineering on 11th November 2017.

ABSTRACT

VILLE MYLLYNEN: Design and implementation of a message standardization tool
Tampere University of Technology
Master of Science thesis, 54 pages
June 2018
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiner: professor Hannu-Matti Järvinen

Keywords: XML, message standardization, automation system, Qt framework, concurrent saving.

This thesis describes designing and implementing an extension to an existing standardization tool that allows configuring and saving diagnostic messages of an automation system and allows the users to save their changes concurrently with each other. The existing tool is a PC application that has been implemented with the Qt framework and is used to configure and save XML template configurations. The XML configurations contain definitions that are similar between automation system configurations, and which are imported to them, reducing the amount of repetitive work. The standardization tool has limitation with saving the changes to standards, when more than one user tries to save their changes to the same standard version. New saving logic is to be created to allow more than one user to edit the same standard version at the same time.

The thesis starts by introducing the target system and the usage of the tool. In addition, the structure and usage of messages are introduced. Then the goals of the thesis are presented. Following the goals, the concurrency issues are viewed, and current saving logic is presented. Two solutions for improved logic are described and a solution is chosen for further design and implementation. The design of the *Messages* standardization tool and the new saving logic are introduced next and the architecture is presented. Using the designed architecture and solution, implementation is done, and the result is evaluated against the set goals. In addition, further implementation ideas are presented. Last, the conclusion of the thesis is described.

The result application is still under development, but a test version including the new saving logic and initial *Messages* standardization tool has been made available to the customer of Wapice Ltd. that ordered the work. Implementation work will be continued with further features and possible bug fixes as the users use the test version.

TIIVISTELMÄ

VILLE MYLLYNEN: Viestien standardointisysteemin suunnittelu ja toteutus
Tampereen teknillinen yliopisto
Diplomityö, 54 sivua
Kesäkuu 2018
Tietotekniikan koulutusohjelma
Pääaine: Ohjelmistotuotanto
Tarkastaja: professori Hannu-Matti Järvinen

Avainsanat: XML, Viestien standardointi, automaatiojärjestelmä, Qt-sovelluskehys, rinnakkainen tallennus

Tässä työssä kuvataan standardointityökalun laajennuksen suunnittelu ja toteutus. Työkalua käytetään luomaan automaatiojärjestelmien diagnostiikkaviestejä ja tallentamaan tehtyjä standardeja rinnakkain muiden käyttäjien kanssa. Olemassa oleva työkalu on PC-ohjelma, joka on toteutettu Qt-sovelluskehyksen avulla, ja jota käytetään XML-konfiguraatioiden luomiseen ja tallentamiseen. XML-konfiguraatiot sisältävät määritelmiä, jotka ovat samankaltaisia erilaisissa automaatiojärjestelmien konfiguraatioissa, ja jotka voidaan ottaa käyttöön niissä vähentäen manuaalisen työn määrää. Standardointityökalussa on todettu ongelmia tallennuslogiikassa, kun useampi käyttäjä yrittää tallentaa muutoksia samaan standardiversioon. Uusi tallennuslogiikka suunnitellaan ja toteutetaan, jotta voidaan sallia usean käyttäjän tekemä samanaikainen tallentaminen.

Työn alussa esitellään kohdejärjestelmä ja työkalun käyttö. Lisäksi esitellään viestien rakenne ja käyttötarkoitus. Tämän määritellään jälkeen työn tavoitteet. Tavoitteiden jälkeen kuvataan tallennuslogiikka ja sen rinnakkaisuusongelmat. Kaksi ratkaisuvaihtoehtoa rinnakkaisuuteen esitellään ja niistä valitaan toinen jatkosuunnittelua ja -kehittämistä varten. Standardointityökalun ja tallennuslogiikan suunnittelu käydään tarkemmin läpi seuraavaksi, ja esitellään kokonaisuuden arkkitehtuuri. Suunnittelun jälkeen kuvaillaan työn toteutusta. Työn tuotosta verrataan asetettuihin tavoitteisiin, jonka jälkeen esitellään jatkokehitysideat toteutukselle. Lopuksi työn johtopäätökset käydään läpi.

Wapice Oy:n asiakkaalle on toimitettu testiversio, joka sisältää uuden tallennusmekaniikan ja alustavan viestien standardointilaajennoksen. Kehitystyötä jatketaan uusilla ominaisuuksilla ja mahdollisilla korjauksilla, kun käyttäjät kokeilevat testiversiota.

PREFACE

This thesis and the related work was designed and implemented during the years 2017 and 2018 for the Department of Pervasive Computing in the Tampere University of Technology (TUT). The work was designed during the year 2017, and implementation was started during the same year and finished during 2018. The project was moved from active development to maintenance where minor bug fixes and code refactoring was continued after the official project was finished. The initial idea for the tool came from the customer and during the design phase it became clear that the project could be a subject for a thesis.

I would like to thank Otto Bothas (Wapice Ltd.) for guidance with the thesis project and the architectural issues in the project, and my project coworkers Ville Tienvieri and Tuomo Heikkilä for their help in designing and implementing the overall project from which this thesis was made. I would also like to thank Hannu-Matti Järvinen who acted as the examiner of this thesis and provided a considerable help with the thesis process and the thesis itself.

Tampere, 21.05.2018

Ville Myllynen

TABLE OF CONTENTS

LIST OF ABBREVIATIONS AND SYMBOLS

API	Application Programming Interface
DLL	Dynamic Link Library [8]
EAP	Enterprise Application Platform [11]
Git	Version control system [15]
GUI	Graphical User Interface
ID	Identifier
IDE	Integrated Development Environment
JBoss	Application server [11]
LCS	Longest common subsequence [13, 16]
MB	Megabyte
MVC	Model-View-Controller [6]
PC	Personal Computer
PostgreSQL	Object-Relational database management system [18]
QMap	Template class with a red-black-tree-based dictionary in Qt [20]
Qt	Cross-platform application framework for C++ [21]
SGML	Standard Generalized Markup Language [28]
SOAP	Single Object Access Protocol [24]
STT	Standard Template Tool
STTID	Standard Template Tool Identifier
SVN	Apache Subversion, software versioning and revision control system [17]
TUT	Tampere University of Technology
UI	User Interface
UID	Unique identifier
URL	Uniform Resource Locator
VS	Visual Studio
Widget	Small application embedded in the UI
W3C	World Wide Web Consortium [25]
XML	Extensible Markup Language [26]
XPath	XML Path Language [28]
XSD	XML Schema Definition [27]

.

1. INTRODUCTION

Automation systems are complex structures where separate machinery pieces are controlled with electronics and software. These systems are widely used in the industrial field and vary widely from production lines to power production systems. To control these systems with software, they must be represented in some way. The systems could be programmed into the control software directly, but this will tie the systems to specific applications. A more typical approach is to create a configuration representing the system which the software then interprets.

To configure the systems, a dedicated desktop application, hereafter the configuration tool, has been created. There are different kinds of automation systems, which can be represented with it, and their contents vary. Despite the variance in the configurations, they will contain similar data structures to each other: *Parameters*, *Measurement details* and *Communication setup* configurations. These can have different values, but the overall data structures and how they are represented are similar. The *Parameters* contain definition for generic data structures like unit systems (e.g. C Celsius, rpm rounds per minute). One of the goals of creating automation systems, is to improve efficiency, and this goal can be extended to creating the system configurations. To reduce the amount of repetitive work, a standardization tool is used to create XML standards or templates that contain the *Parameters*, *Measurement details* and *Communication setups*. These standards and templates can then be imported to the configuration by using the dedicated configuration tool.

The goal of this thesis is to design and implement a message standardization extension to the existing standardization tool. The messages are a way for different parts of the system to inform the whole of changes in them. The main research topic that this thesis addresses is the concurrent usage with multiple users. The current functionality allows only one user to edit a single standard version at the time. To solve this issue, the new extension must support the concurrent saving of the template files if the edits do not conflict with each other. In addition, the concurrency must be designed and implemented in a way that is extensible to the other standard types of the existing tool.

The chosen method for this thesis is constructive research. First the background of the system is introduced. Then the goals for the solution are presented. Next, the problem, existing solutions and proposed solutions are described. Based on these a design for the solution is created which is then implemented. Finally, the created solution is compared against the set goals and future implementations are presented.

Chapter 1 contains the introduction to this thesis and its structure. The basic idea of the standardization tool is introduced.

Chapter 2 presents the background of the system and the existing application. The purpose of the application and usage is also presented.

Chapter 3 describes the goals of the thesis, basic functionalities, effects on the existing system and the new saving logic.

Chapter 4 describes the concurrency issues the current system has and presents solutions for it. Finally, a solution is chosen for further design and implementation.

Chapter 5 presents the design for the new saving algorithm and the *Messages* standardization tool.

Chapter 6 covers the implementation of the new system and saving logic. In addition, the goals of the thesis are evaluated. The future implementation ideas are described at the end.

Chapter 7 contains the conclusion of this thesis.

2. BACKGROUND AND ENVIRONMENT

The configuration tool of which the standardization tool is an extension to, is used to create and monitor an automation system. The application is used to create the extensible markup language (XML) configuration which represents the system, the modules in it, their connections and parameters [26]. This configuration along with the individual application binaries for each module can be downloaded to the modules with the configuration tool.

In addition to creating and configuring the automation systems, the configuration tool offers monitoring capabilities. The monitoring is enabled by connecting to the system through the gateway access point. The modules can broadcast essential diagnostics details, such as states and alarms, as messages to the configuration tool. The diagnostic data is based on the configuration and the control software running in them. The configuration tool monitors these messages from the system and can translate and display them for the user.

The messages send by the modules to the configuration tool, like other parts of the system, must be defined in some way. To reduce the amount of repetitive work, a standardization tool has been created. The tool is an extension to the configuration tool. If a user has the appropriate access rights to the application, the standardization tool is available. The user can view and modify the different standard types in the standardization tool. The number of users of the standardization tool was several tens of people and usage was from weekly to monthly. The number of users will increase considerably with the addition of the new *Messages* standard type support.

The tool is described in more detail in Section 2.1 for the client-side. XML and its usage are presented in Section 2.2. Section 2.3 contains information about the server-side. Currently, the standardization tool does not support creating standards for the messages. An older standardization tool is currently used to create the messages standards. It is introduced in greater detail in Section 2.4 with reasons why it is being replaced. Section 2.5 presents the message structure and usage.

2.1 Standardization tool

The existing standardization tool is an extension to the configuration tool that is used to create the automation system configurations. The standardization tool is not available to broader audience, but rather to specific users with the proper user account profile. The profiles are currently divided into two different types: developers and viewers. Both have the access rights to login to the standardization tool and read the configured standards in

the application. In addition, the developers have the right to make edits to existing standards and publish new ones.

The standards are saved to a JBoss 6.1 Enterprise application platform (EAP) server which utilizes PostgreSQL version 8.4.20 database for storing the standards [11, 18]. Standardization tool will load the existing standard version from the server, creates a local copy of the data and shows it to the user. The server will be introduced in Section 2.3.

The standardization tool has three distinct views, one for each supported standard type: *Parameters*, *Measurement details* and *Communication setups*. Each view has the selection to view and edit a specific standard version. The version is controlled with a separate drop-down menu located in the lower part of the view. The standard navigation view is shown in Figure 1.

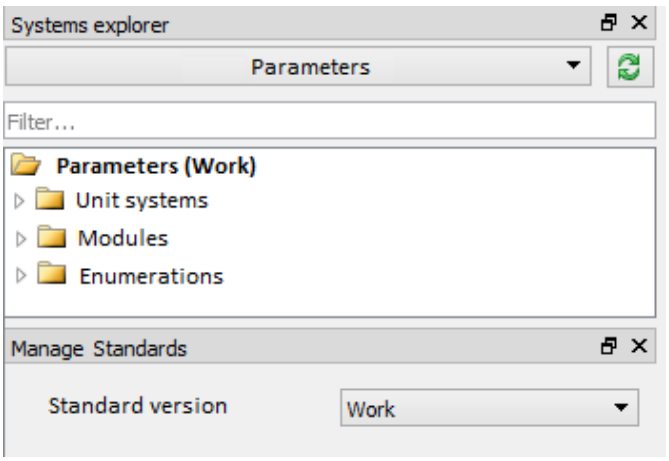

Figure 1: Standard view and version navigation.

The users can open tree items from the system explorer tree which will open the configurable items of the standard. Some items also have the option in their context menu to add children elements and copy or delete the item itself. When the user opens the tree item by left clicking it, a view is constructed from configured view XML. The view XML defines the view elements which are created. The view is populated with data from the XML configuration. The user can edit the standard items and finally save the edits by saving the data (if editing the *Work* version) or publishing a new standard revision (editing published standard).

Each standard view has its own standard versions which are divided into two categories: published standards and single work version. An example of the standard versions can be seen in Figure 2. The work version is always the newest version and is not a published standard version. The users can publish the work standard, at which point the standardization tool will create a new major standard revision branch from the work version. The users are also able to edit the existing standards and publish a new revision of the standard

branch (e.g. 3.0 → 3.1), but the published standard branches have no dirty or work version. This means that each further edit to an existing standard requires publishing a new revision of the standard. The publishing is limited to the latest revision of each standard branch, meaning that if standard branch 3.x has standards 3.0, 3.1 and 3.2, only 3.2 can be edited and a new standard revision 3.3 can be published from 3.2 version.

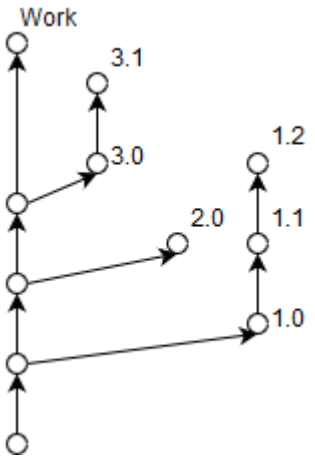

Figure 2: Standard version tree.

The standards can have dependencies between them. The *Measurement details* standard type has a dependency to the *Parameters* standard type. This means that the *Measurement details* standards refer to values configured to some published *Parameters* standard, such as the unit system values. *Communication setup* standard type has a dependency to *Measurement details* standard type, and by extension to *Parameters* standard type. The values configured to the *Measurement details* standard version to which the dependency is to, are available for the configuration in the *Communication setup* standard. The values configured to the *Parameters* standard, to which the *Measurement details* standard has a dependency to, are available as well. The dependencies can be changed by changing the standard version to which the dependency is to. In case of *Communication setup*, the secondary dependency to *Parameters* is changed according to the new dependency to *Measurement details* standard.

Publishing a new standard revision or saving work version sends the local version of the XML standard document to the server where it is saved to the database. Creating a new standard version or saving the work is only possible if the standard has not changed on the server. This means that if *User A* is editing standard 3.0 and tries to publish standard 3.1, but *User B* has already published his changes as standard 3.1, the *User A* cannot publish his changes. The only way the *User A* can make their changes and publish them, is to reload the standard data from the server and in the process, remove all the local changes they have made. Then the *User A* can make the changes again and try to publish them as new standard revision. The same is true for the work version of the standards.

2.2 XML, schemas and XPath queries

The internal data representation of the standardization tool is in the form of XML document. XML is a restricted form of the Standard Generalized Markup Language (SGML). In XML documents, some of the characters in an XML document are elements of the XML and form the structure of the document. Some are the actual data which is stored into the elements or nodes. [26]

The structure of the XML documents can be restricted with XML schema language. The schema of the XML documents is written as its own document and the schema is referred to in the XML document. This way in addition to checking the well-formedness of an XML document, the validity can also be checked against the schema. The schema defines the elements that can be used in the XML document, the hierarchy of the elements restricting the locations where the elements can be used, and to offer documentation that is both human and machine readable. [27]

Elements of the XML document can be accessed by using a XPath query language. XPath defines the elements and/or attributes that should be traversed to reach a node. The language is further expanded by supporting conditions which allow matching and comparison with the queries. XPath models the XML document as a tree of nodes that can be traversed. There are different node types, of which relevant to this thesis are: the element, the attribute and the text nodes. [28]

While XML is meant to be both machine and human readable [2], XML configurations with 10000 to 100000 lines if formatted for humans to read, means that the human readability suffers. To help the user, the standardization tool abstracts the configuration work to creating, editing and deleting items such as unit systems. In addition to the size of the document, the users of the standardization tool may not know the underlying XML, and therefore the tool should abstract the data. This design is extended from the configuration tool, where the users work on higher abstraction level.

The standardization tool abstracts the XML data model from the user, but the underlying logic uses the XML to store the data of the configuration. XPath queries are used to read and write data to the configuration, and schemas are used to create new nodes that are to be added to the configuration.

2.3 Standardization tool server

Standards are stored on a remote server that has a JBoss server application running, which uses PostgreSQL database to store the data [11, 18]. The standardized XML configurations are stored as whole XML documents to the database with relations to other tables where the standard schemas, view IDs, versions and comments are stored [27]. Editing a standard and the server communication is depicted in Figure 3.

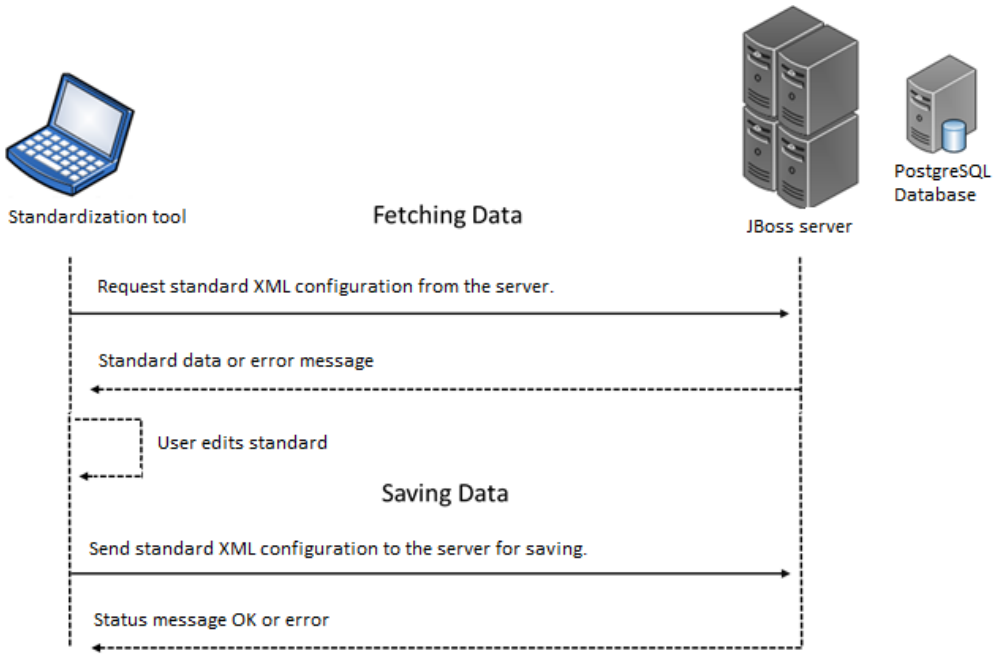

Figure 3: Server communication.

The server is stateless and is considered thin with a fat client application [29]. The client sends requests to the server, such as *save a standard* or *get all standard versions*. The server only parses the request message and reads necessary values from it. For saving a work version of a standard the values are the view ID, the version, the comment and the data. The data will then be written to the database according to the other values given in the request.

The server offers an address to which requests can be made and according to the parameters, the server performs a requested action, such as saving a standard or retrieving a specific schema. A response is then generated and returned to the client. The clients sending requests to the server can be divided into two main groups: 1. standardization tool instances that are loading the data for viewing and editing, and 2. configuration tools that are importing standards to the system configurations.

The server code is legacy code that has not received major upgrades in several years [5]. Over time small modifications have been made, but the Java library dependencies have not been maintained over time. This has caused deprecation in the code base which is an issue to consider when designing and implementing code into the server itself.

The server handles requests concurrently, by creating a separate service thread for each of them. This implementation has a problem that currently there is no proper concurrency control implemented into the server. Because of this, there is a risk that a change written by one user to the work version of a specific standard is overwritten by another user's write, if the save operations are performed near simultaneously. The risk has not realized so that it would have been noticed, because the issue exists only for the Work version of

each standard and the user amounts are so low, that the chance of data overwriting is minimal. Therefore, the issue is not further considered in this thesis.

2.4 The predecessor of the message standardization tool

Currently, the users use a separate Java application to standardize the messages [10]. The application is specific to the messages and uses a separate server and database for data writing and reading. Each different standard type has its own application with which to create and edit the standards. The applications are accessible through a website from which the user can download the applications if they have proper credentials. Some of the applications have been deprecated as the standards have been moved to the new standardization tool which is an extension to the configuration tool (from Section 2.1).

The old tool has the configurable messages as actual database entries and as such the standards are not stored as whole XML documents in the database. The users can create, edit and delete messages in the old standardization tool. To edit or delete existing messages, the user must enter the edit mode for the single message. This allows multiple users to edit the same standard if the edits do not target the same messages. This is a functionality that is currently missing from the newer standardization tool.

The old standardization tools are no longer maintained and are in production use. The new extension designed in this thesis will replace the old *Messages* standardization tool and enable the user to use the same application to edit several different standard types.

The old standardization tools have issues where the created standards are too limited in structure. An example of this is the messages standard, where the maximum group depth is only two, whereas the new implementation will allow the user to configure the group depth limit to each standard. In addition, the work flow for older standards when creating new systems is to export a file where the values are found. To make use of the data, the user had to manually move the file to correct location and import it. The new standardization tool offers greater integration, where the configuration tool can directly import the chosen data from the server, and the integration process is thus more automated. The integration is also visible between the standard types. With the new tool, the user can first create or modify parameters standard. Then they can take that standard into use with measurement details standard they are creating with the same tool. In addition, the old tools have several smaller issues and bugs, which are not being worked on as the tool is no longer maintained.

2.5 The purpose and usage of the messages

Messages are used as a means of communicating the states and changes in the modules to the whole system, which the monitoring application will then show to the user. When user connects with the configuration tool to the modules, the log messages are read from

the system and shown to the user. If a module crashes or encounters an error it cannot resume from, a message about the situation is shown in the log. The log also shows messages that are received when the modules go from operational to booting, to pre-operational and finally operational.

The messages are divided into five different categories: *info*, *safety*, *error*, *event* and *debug*. Each of these types has a different use case. For example, an *info* message can be used to inform the user that a booting of an external module was successful, where as a *debug* message can be used for developing the external modules and applications that are executed in them.

The log works mainly as a diagnostic view when the application is connected to the automation system. The log provides an easy way to determine and debug possible issues in the system. In normal situation, all the modules are aware of all the messages in the system, and there is no reason to pass all the message information to all the modules. In most cases, the modules must know the unique ID of the messages and all the parameters the message requires to be able to pass the correct parameters when sending the messages. The main application will receive the ID and parameters of the message and then store it to the buffer according to the category of the message, and show the user more information in the log, such as the description, recommendation and implication of the message.

3. GOALS OF THE THESIS

The goal of this thesis is to design and implement an extension to an existing standardization tool. The extension allows the users to create and maintain message standards that are XML configurations. The standards can be taken into use in the system configuration tool. The original message standardization tool is an independent Java application which is in customer use. The new standardization tool has replaced some of the old standardization tools and the new extension is to replace the old message tool. The new implementation must allow the users to save their changes concurrently to the database. The goals are divided into three main areas, each described in their own sections: the basic functionality of the new standard type (3.1), effects on the existing system (3.2) and the new concurrent saving logic (3.3).

3.1 The basic functionalities

The extension will be available in the standardization tool. The active view can be changed from the starting view of *Parameters* standards to the messages standards. The XML configuration is loaded at startup and the view will be built when the user changes the active view to the message standardization tool.

The configuration items are presented in the system explorer tree as a tree structure as shown in Figure 4. The tree contains group items which allow the user to group the configured messages in the standard. The groups can contain other groups. The depth of the group structure is limited to the maximum of 10 groups, but the user can set this limit to be lower using a configurable value in the configuration. The groups can also contain the messages and the messages can belong to multiple groups. The message must be configured only a single time to the XML configuration and the UI must support showing the same XML item in multiple positions in the system explorer tree. The reason for this is that in a normal use case the user is interested in only a specific group of items, but the groups can overlap in their contents. Thus, the same items must be present in multiple groups, but the size of the XML configuration is to be kept under control and data duplication must be avoided.

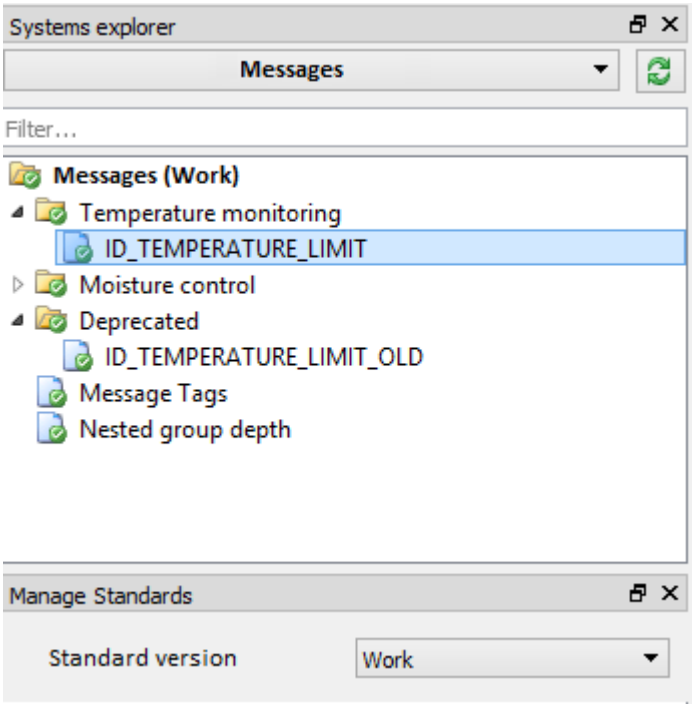

Figure 4: Messages tree explorer.

The tree view must have a context menu. From the context menu of a group item, it is possible to create a new message, create a new child group, and edit or remove the group. The context menu of a message item allows the user to deprecate the message or delete it. Left clicking a message item, opens the configuration view.

The messages should be configurable in multiple languages. Due to the requirement, a custom view handling is required. Initially, the tool must support configuring the messages only in English, but the support in XML level for multiple languages must be taken into consideration. Normally in the application, the views would be constructed from separate XML configurations through a generic view constructor. That is not an option for the messages because of the more complex structures, as the amount of shown data should be limited to improve usability [9]. The item configuration is done in the view by editing the values of the items, such as the description of a message. The tool will then write the change to the XML configuration which can be saved to the database server.

The standards stored to the server must be valid. The validations are done based on the user input, the existing values and the dependencies to other data entries. The errors are shown to the user in event view, along with possible warning and info events. Possible errors that can be created are: a duplicate message, tag, category or group name, an empty mandatory field or an illegal group depth configuration. Most of the errors can be detected when the erroneous situation is created, such as a duplicate group name. An exception to this is the changing of the standard dependency.

The message standards will have a dependency to *Measurement details* standard and by extension to *Parameters* standard. The dependency can be changed from a dependency

change dialog which will update the information to the XML configuration, but the change in dependency can introduce new errors to the configuration. If a configured message uses a specific unit system (e.g. *rpm*) or a code as its parameter, and the unit system or code is no longer available in the standards to which the *Messages* standard has the new dependency to, then new errors have been created in the configuration. The user must then reconfigure the items which have errors before the standard can be saved or published.

3.2 Effects on the existing system

The standardization tool has complex code that in several places can be considered legacy code. Editing the legacy code has a risk that the edits have unexpected side effects, and thus the new extension should be designed and implemented in a way to minimize the risk to the existing standardization views [5].

Chapter 5 describes the current architecture of the system and how it is going to be extended to support the new *Messages* standardization tool. The new extension should not cause considerable regression in the existing code base and possible refactoring work must be done as separate work to ensure there are no unexpected side effects. The work will also contain changes that will be done to the JBoss server and the database where the configurations are stored. A new API function version must be implemented into the JBoss server that the standardization tool will utilize in the new extension. This way the original views can use the current version and the new functionality can be added to the server.

Currently, the standardization tool loads the data for all the views on startup. For the *Parameters* and *Measurement details,* the loading is reasonably fast since they have either no dependencies or a single dependency. By contrast, the current *Communication Addresses* and the new *Messages* standards have a dependency to at least two different standards. The data loading is slower at that point and for the future implementation, a solution should be created to ensure the data loading is reasonably fast to ensure better usability and efficiency. This improvement is not the goal of this thesis, but the refactoring of the startup logic must be considered.

The implementations to the general code of the standardization tool are to be designed in a way to allow the existing views to be extended to use them. The concurrent saving logic introduced in Section 3.3 must be done in a way to allow the existing views to take it into to use in the future.

3.3 Concurrent saving of the standards

The new implementation for the concurrent standard saving logic is to be designed and created for the new extension of the application. Current saving is done by sending the

complete XML document to the server and creating necessary database entries for the document and the data relations to schemas. The current saving results in situations where one user saving their version of the standard to the database prevents other users from saving to the same standard branch. In addition, the users cannot move their changes on top of a fresh version of the saved standard and instead the user must perform a reload action, which clears all the changes they have done.

The new saving logic must allow multiple users to save the edits to standards if the changes do not target the same items. The change requires definitions for the smallest level of items, since the level of single XML element is too precise. For the *Messages* standard, singular items are the messages, the groups, the tags and the categories. The user can add, edit and remove each of the item types. The saving operation must support saving each of the operation types.

In case users edit the same items in a standard, user's changes cannot be written automatically. The user should be informed about the situation, but the user may not be familiar with XML, and therefore all the communication with the user must be done on a higher abstraction level.

For the first edition of the *Messages* standardization tool, the conflicting items are shown to the user and the changes done to those items are discarded. Improvements to this logic are further discussed in Section 6.4 where future implementation ideas are presented. The number of conflicts has been evaluated to be quite low if the application use is done in planned manner, where one message is not being maintained by different people. The conflicting information must be presented in a clear and informative manner for the user, so that they know how to fix the issue.

The implementation must start another save attempt in case the first attempt fails. A reason for such a failure can be that the data has changed on the server during the saving operation.

4. CONCURRENCY ISSUES AND SOLUTIONS

The new standard extension must support concurrent standard saving, which is currently missing from existing standardization tool. The issue has not been critical before. In the future, it may become more prominent because the number of users will increase as the supported functionality of the tool is extended and more standard types are implemented into it. Therefore, a new functionality that allows multiple users to save changes to standards must be implemented. The logic must also be extendable to the existing views if they are converted to use the new logic in the future.

To solve the concurrency issues, the starting point is to create logic to detect exactly what items have been added, edited or removed in the XML configuration. This logic is further discussed in Chapter 5. The level of detection can be line-level like some existing version control systems (e.g. SVN) or more abstract [17]. The more abstract option requires the program to know that what kind of items are considered singular objects.

Currently, the application does not support actions where the local changes are transferred on top of another XML documentation. A solution can be to use rebase which is an action found in Git [1]. The action can be used to allow multiple users to move their changes to the XML configuration on top of a fresh configuration. This allows them to make saves at the same time if the changes do not have the same targets. If the changes have the same target, the changes are in conflict and must be resolved. In Git, this means that the user must view and understand the line level conflict and solve the issue to prevent any unwanted change.

The end users of the standardization tool may not know about XML, so resolving the conflicts on line-level is not user friendly. Item-level conflict resolving is easier for them than the line-level because they have more knowledge on the items that they are editing [9]. This has the effect that the number of conflicts may increase when compared with the line level conflicts. In normal workflow, the users do not edit the same items and thus in theory the conflicts do not increase [14]. This and the fact that the usability of the software increases with lesser granularity as the conflicting changes gain context from the surrounding data of the changed item. The minimum requirement for the program is to allow the user to save all their changes to standard that do not conflict with other changes to the same XML configuration and discard the changes that conflict.

The current standard saving logic is presented in Section 4.1. Existing solutions to rebase or merge XML configurations are presented in Section 4.2. The standardization tool has a modification handler class which keeps track of changes made to the XML. By modifying it, a list of items that have changed can be created. With this information, the rebase

can be done, but whether the rebase is done on the client or the server side must be decided. The solutions are presented in Sections 4.3 and 4.4 and a comparison between them is done in Section 4.5. Based on the chosen option an architecture is designed in Chapter 5.

4.1 Current standard saving

On startup, the tool will determine all the views it supports and requests the latest XML configuration (work version) for each of the views from the server. The XML configurations are loaded to the memory of the program and their version dependencies are checked. If they have a dependency to another standard, the dependency standard is also loaded and then injected to the loaded XML configuration. This will be done until the dependency chain has been executed and each of the dependencies has been loaded and injected to the XML configuration. For the *Messages* standard this means that it first loads the *Messages* XML configuration, then the *Measurement details* standard and finally the *Parameters* standard and combines them into a single XML configuration.

The loaded XML configurations are stored to the memory of the program and are swapped to be the active ones when the user changes the view in the tool. After the data has been loaded and the views have been built, the user can make edits to the standard versions. For the released standards, the user can only edit the latest revision of the major standard branches. When they are satisfied with their changes, they will attempt to save their work. For the work version, the user has two options: *Save to Work* and *Publish version*. The first one is only available if the user is in the work version making the changes, and it is not available when editing published standards. The second one is available in both the work version and the published standards. *Save to Work* saves the changes to the work version on the server and does not affect the published standards. The changes are not publicly available and only the standard editors themselves can import the work version for their system configurations. The action flow of *Save to Work* is presented in Figure 5.

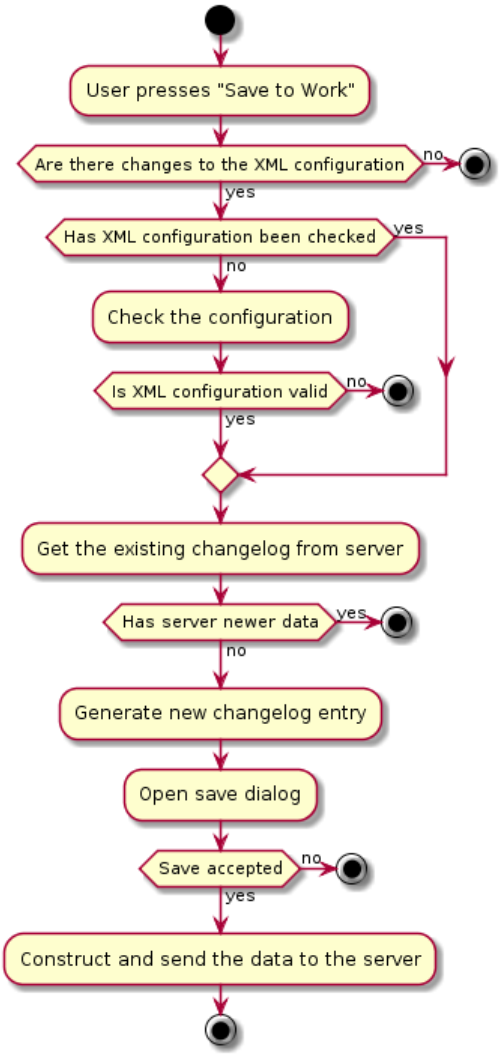

Figure 5: Current save to work.

The *Save to Work* logic is linear with no loop to the previous steps of the logic. If at any point, the save encounters an issue, the operation is terminated and in the case the user did not stop the process themselves, an error or info message is shown to them.

Publish version logic differs when the currently active version is work or a published standard. If the current version is the work version, the changes are first saved to the work version to ensure that it is always on the level with the newest standard branch. After that, a new standard version is created where the major version number has been incremented and minor number is set to 0. Both the edited work version and the new standard version are saved to the database. When the active version is a published standard, the data of the existing standard is not edited, but instead a new version is saved to the database and the minor version number of the standard branch is incremented for the new version.

The current *Publish version* logic causes each edit to an existing standard to require a new standard revision. The multiple standard revisions waste resources if they are truly not used. Allowing users to save changes to a standard version without increasing the version

number reduces the number of unused standards. Editing can be done gradually and when the standard revision is deemed publishable, a new version is created. This is further discussed in the future implementations in Section 6.4.

The standardization tool stores the XML configurations that were loaded from the server and copies them to keep the original data available for comparison. The changes made by the user are made to the stored XML configuration and when the user saves or publishes the standard, the data of the standard extracted from the XML document and sent to the server. The server will then create the necessary database entries and store the received XML configuration to them. Usually, the user does not make changes to all the items in a standard and thus some unnecessary data is sent to the server, especially when saving the work version. Because the whole XML configuration instead of individual changes is sent to the server, the server-side implementation does not monitor the received XML for its validity and stores it as is. The server is lean and the client program (standardization tool) has more of the implementation logic, such as checking the validity of the XML [29].

The side effect of sending the XML configuration to the server is that if more than one user were editing the same standard version, it is not possible to merge their changes into the same XML configuration. Thus, if the second user attempts to save their changes after the first user, the save is stopped, as accidental data overriding would happen if the save was accepted.

The server is used with older versions of the standardization tool that will not support the new extension. This also restricts the new logic to saving and loading data from the server. The old server API must be kept intact if possible and new API functions or function versions must be implemented for the new functionalities.

To solve the concurrency issue, a merge of two different XML configurations must be taken into use. Depending on the chosen implementation type, the result XML configuration will be created either on the client-side or the server-side.

4.2 Existing solutions

There are already existing merge implementations that can be used to merge two XML configurations into one. Multiple version control systems can do it, of which Git is used as an example in Section 4.2.1. Existing merge algorithms are also studied to introduce the different solution types for the merging.

4.2.1 The Git rebase

Git is a distributed version control system that can be used to track and manage changes to files and handle combining different changes (commits) into a single history of changes

[1]. The functionality of Git can be described as storing a snapshot of the files it tracks and creating a history of snapshots [1]. Git supports branching which means that from the history of a single branch (often called *Master*) a new branch is created. The branches have the same history until that point. Anything newer than the point of branching in either branch is not part of the common history unless the user combines them.

There are several ways for Git to add the changes introduced in one branch on top of another. The rebase operation of Git is one of them. A rebase of *Feature* branch on top of *Master* branch is depicted in Figure 6.

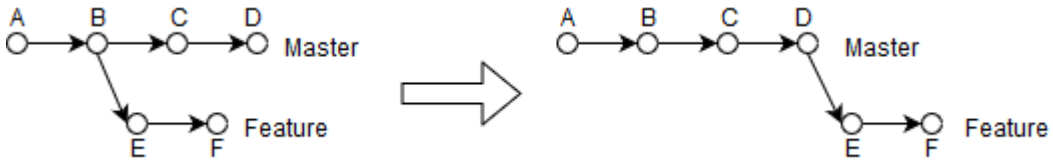

Figure 6: Git rebase [1].

First there is a *Master* branch. The user wants to implement a feature and to do that, they create a new branch *Feature* from the *Master* branch. Later, a critical issue is detected in the software and a hotfix is quickly implemented and added to the *Master* branch as commits *C* and *D*. The issue affects the development of the feature, but the hotfix is not available in the *Feature* branch because it was created from *Master* branch before the hotfix was added to the *Master*. Instead of manually writing the same solution implemented in the hotfix to the *Feature* branch, the user can perform a rebase operation on their development branch. During the rebase operation, Git determines the common point of history between the two branches, then stores the changes done after the point of common history to the branch being rebased. The history of the branch is then replaced with the history from the branch from which the rebase is done on. The stored changes are then applied on top of the new history in order of creation.

In ideal situation, the rebase operation does not encounter issues and the process is performed automatically. However, sometimes the operation notices a conflict between the changes in the new history and the changes being applied on top of it. If the hotfix and the *Feature* had changes to the same lines, a conflict situation is created. The Git is unable to determine the correct result and the user must resolve the conflict. The conflict location shows the different versions of the location and the user must then combine the versions to resolve the situation. Once all the conflicts have been fixed, the rebase operation can be continued.

The Git rebase logic can be used as an example for solving the concurrency issue when saving the standard XML configurations to the server. The XML configuration stored to the server can be thought as equivalent to the *Master* branch from Git and the local versions are different branches that have been created from the *Master* branch. When the user wishes to save their changes to the server, the changes are stored and applied on top

of the latest version of the XML configuration from the server. In case the XML configuration of the server has new changes done to the same items that were changed locally, a conflict situation is created. The minimum requirement defined in Section 3.3 states that at this point, the user is informed about the conflicting items and the local changes are removed and must be done again on top of a fresh XML configuration. Further improvements are described in Section 6.4. The client application could allow the user to resolve the conflict by showing the conflicting item and the changes done to it. The user could at that point make the required changes to create the final version of the item that would be saved with the rest of the changes.

Because the conflict level has been decided to be at item-level instead of line, there is no need to compare the entire XML configurations. Instead, a node from the change location is fetched from the original local and the server XML configurations. The nodes are then compared with each other. If the nodes have no differences, the local change can be safely applied to create the final version of the document. This process is repeated for each change. The different ways to solve the rebase logic are shown in the Sections 4.3 and 4.4.

4.2.2 XML merge with versioned tree and identifiers

C. Thao and E. Munson describe an algorithm for three-way merging of the XML documents [16]. According to them, the benefits of their algorithm are the speed and memory usage when comparing with other three-way XML merging algorithms.

Three-way merge of files requires three different versions of the file. The base line and two modified ones, both of which were created from the baseline by making changes to it. The algorithm creates delta from the base line and the changed versions and then using these, creates a document where the changes from both files are present. [16]

The authors use an existing versioned data structures to represent different versions of the XML configurations [16]. For the rest of this section, the word '*node'* refers to the node in the versioned data structure which represents an element in the XML. It should not be confused with the node in XML document.

Nodes in the data structure contain information such as the version of the document, references to other versions and nodes, and a node value. Each node also has a reference to the older versions of the node. A ChangeRecord object is created to listen to changes in a single node of the versioned data structure. The object knows if the node has changed, but not the type of the change. [16]

The authors define following merge operations: addition, deletion, update, update where both versions target the same node, but the changes are disjointed, and move. Conversely, they also define conflict rules: 1. One node is moved to a specific location in one version,

and another node is moved to the same location in the other version, so the order of nodes cannot be determined. 2. A node is moved in both versions but to different locations. 3. A node is moved or updated in one version and deleted in the other. 4. An attribute of the node is updated to have a different value in the different versions. 5. The node is deleted in one version and it or its descendant is updated in the other version. [16]

Every XML element is required to have a unique identifier (UID) in the authors' proposal. The UID is added to each element when the document version is added to the version control system. When an element is added to the configuration, the UID field is left empty. This allows the detection of addition in the merge algorithm. This UID is used to create a hash table from the versioned tree of the baseline where the UID is mapped to an element. Then a parser reads the first modified document version and compares each item with UID to the existing node in the base lines hash table. If the node has changed, the change is added to a delta that is constructed from the differences between the baseline and the edited version. Each node without UID is an addition and added to the delta. The delta for the second edited version created in similar manner as the one for the first edited version. [16]

A new third version of the document is created from the second edited version. Then each changed item in the first version is parsed. For each change, the longest common sequence of nodes is calculated in each version (baseline, first edited and second edited). Then an existing diff3 algorithm is used to compute a new sequence for the node. [16]

The proposed algorithm cannot be used directly in the standardization tool, as the algorithm handles the changes with too fine granularity. The algorithm presented by the authors works on the XML element level where the elements can be added, updated, moved and deleted. On the other hand, the concept of a singular item presented in Section 3.3 handles the items on the level of real world items the XML elements define, such as a message or a group. Also, worth considering is their approach that XML configuration is ordered [16]. This is only partly true for the standards, because the order of messages or groups is not as critical, if the depth of the item does not change. Thus, the conflicts presented by the authors are only partially applicable to the XML merging in the standardization tool.

The solution proposed by the authors has concepts that are also found in the standardization tool. The usage of UID is like another identifier which is used in the tool. This identifier and its usage is further discussed in Chapter 5. The ChangeRecord object has similarities to a modification handler already found in the standardization tool but has a differing use. Finally, the three-way merge algorithms use three different versions of a single file to merge the changes: A baseline and two edited versions which were created from the baseline version. The standardization tool has the baseline as well as one of the edited versions already in the program memory when the rebase operation is performed, so the

existing data can be used in the client-side rebase operation, which is presented in more detail in Section 4.3.

4.2.3 XML diff with context fingerprints

S. Rönnau and U. Borghoff present a diff algorithm for XML documents in their article [13]. In their approach, they calculate deltas which contain a list of edit operations done to the XML configuration. One version of the document can be created by applying a delta to the other version.

Issue when comparing the XML documents for merging is the non-persistent paths in the XML. As XML documents are often ordered tree structures, changes to the element order by insert or delete changes the tree structure. This may cause other changes to be written to incorrect locations if not taken into notice. [13]

C. Thao and E. Munson used UID to determine the node identity and thus allow differencing a node that has been changed from others [16]. S. Rönnau and U. Borghoff present a concept of context fingerprints [13]. To support different node types found in the XML (text, element and attributes), they present that each node has calculated a value. The value is a hash value which is calculated differently for different kinds of nodes. For text nodes, the value is their contents. For elements, the node name and attributes are represented as normalized. If two nodes have the same hash value, they are considered equal.

Their approach is to use the document order. The hash values of the immediate neighbors of the node are used to create a sequence of their hashes. This acts as the fingerprint. Furthermore, they define two concepts: the depth and the height of a node. The depth is the step count from the node to the root node. The height is the longest path from the node to its descendant. [13]

They define four different change types: insert, delete, move and update. Of the operations, the first three modify the tree, its subtree or tree sequence. This knowledge is used in their algorithm to determine the changed items. [13]

First, they process the leaf nodes of the document. For each leaf node, the XML versions are compared, and the node is considered matching if it has the same value and depth. All matched noted are added to a list. Next, they check the parents of the matched nodes. Matching nodes are added to the list and mismatches have been updated. The tree is then traversed bottom-up. Since the leaves that matched had the same depth, possible changes must have kept the structure of the tree the same. Insert and delete operations were thus captured in the first step. An exception to this detection is if the leaf node was moved from one parent to another with the same depth. This situation can be detected here as the parent nodes are compared and possible mismatch is detected. Lastly, the non-matched nodes and their insert and delete operations must be identified. For those nodes, the tree

is traversed upwards until already matched node is encountered. This way, larger subtree changes are also detected. [13]

The diff and merge of XML documents in the standardization tool is much more limited in scope. As the XML elements are abstracted to a higher concept such as messages, it is important to consider this in the design. In addition, using this algorithm means that a lot of the existing code base cannot be used for the rebase logic. An example of such logic is the log of changes the application keeps during the user's session. Each change is logged and available during the rebase operation. This is further compounded, because of the concept of a singular item. An edited item in the standardization tool may have had its internal XML structure changed considerably with all the different change types S. Rönnau and U. Borghoff define. Furthermore, the order of items in the singular items is not important in most cases. Because of this, the actual node value instead of the order of nodes is more important.

4.3 Rebase on the client side

The client-side rebase solution performs the rebase in the client application and sends the updated XML configuration to the server for saving. The server implementation is currently very lean and does not perform complex operations or validations on the data it receives from the clients. The client side rebase builds on top of the existing logic by keeping the server-side implementation simple and handling the synchronization and data manipulation actions on the client side. This means that the client-side implementation will become more complex as it must solve the following issues: how to guarantee no data is overwritten without user confirmation, how server data is modified in synchronous manner and amount of data being sent. The client-side rebase logic is displayed in Figure 7.

Figure 7: Save to work (client side rebase).

As shown in the figure, the logic is mostly done on the client-side and only at the end, the execution is passed to the server, which checks that is the save ok to do and responds to the client.

The saving algorithm

In this solution, the saving logic is started by requesting the latest XML configuration from the server. The client compares its own original XML configuration with the received one on the items that were changed locally. At this point, the logic branches depending on whether there are conflicts or not. If there are no conflicts, the client will finish the rebase operation and send the whole XML document to the server for saving. If there are conflicts, the user is informed about the conflicting items. The local changes targeting conflicting items are discarded and the rest are applied on top of the new XML configuration.

First, the client will create two new XML configuration instances, both of which will be identical. The first one will be kept as is and is used with the local original XML configuration to compare client and server XML configurations. The second XML configuration will be modified when the changes done to the active XML configuration of the client are copied to it.

The client compares each change location in the loaded XML configuration with the local original configuration. If the XML configurations have a difference, the change is in conflict. If a change is in conflict, the user is informed about the situation. If all the changes are in conflict, the saving process is interrupted. Otherwise the user has an option to continue the saving process or cancel it. If the user continues the save, only the non-conflict changes are saved. This implementation also allows the extension of the logic where a conflict resolution wizard can be implemented. The wizard is shown after all the changes have been checked and a conflict is found. This is further discussed in Section 6.4.

Next, each non-conflict change is written to the second loaded XML configuration. After the changes have been copied to the new XML configuration, it is checked for validity. For the standard types that use XML files to define the view contents, the checks are also defined by the XML. For the standards (such as *Messages*), the validations are programmed into the standardization tool. If the validation fails and errors are found, the saving process is interrupted, and the user must fix the issues before saving their changes. Otherwise, the saving process is continued, and the saving dialog is opened for the user.

The saving dialog is shown in Figure 8. All the changes the end users do to the standards must be traceable to a documented issue and the issue ID field allows the user to give the issue number. A list of links is created from the given issue numbers which allow easy navigation to the issue documentation. Second field is partly pre-filled. The application automatically generates the change log entry, based on the changes that were done. The changes are documented into this field, but the user can edit them and add more information if they want. Above the input fields, the change log of the standard being edited is shown. When the user is satisfied with the new change log entry to be created, they can

press the save button which will start the process of sending the data to the server for processing.

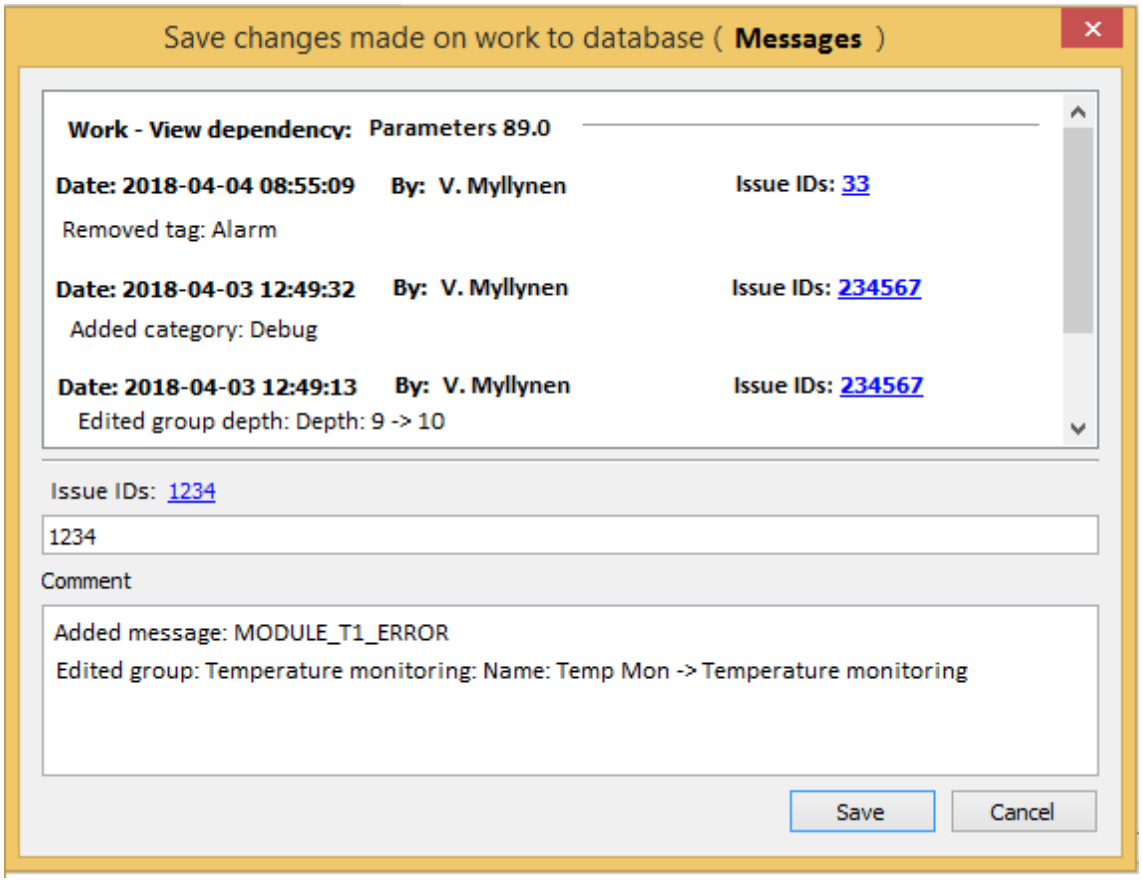

Figure 8: Save dialog.

When the standard data is loaded from the server at the beginning of the saving, the change log of the standard is loaded as well. The change log is shown when the user saves the standard and gives a new entry to the log. Second usage for the log is to use it to determine if the server data has been changed, since a new entry is always added to the log when the XML configuration is saved. This can be utilized when sending the rebased XML configuration to the server for saving. When the data is being sent to the server, the client passes the change log it received from the server when it loaded the data at the beginning of the saving operation. The server then compares the received change log with the one that is stored in the database. If there are no changes, the client operated on the most recent XML configuration and the server performs the save operation. If there are changes in the change log, someone has saved their changes during the rebase operation done on the client-side. The server does not save the received data and instead replies to the client that the data has changed. The client can then start a new rebase iteration and save attempt.

The server-side implementation is lean and existing functionalities can mostly be used for the new saving operations. The major difference is the addition and handling of the

standard change log that will be passed along the save request. The server uses the received change log to determine if the server contains newer data than the one on top of which the client rebased its changes. Because a change must be made to the save request, a new API function version must be implemented into the server. The server only validates the request was built correctly and contains all the necessary data. It does not consider the actual contents of the XML configuration that is to be saved to the database. The server reads the received change log and the data, and then it compares the change log with the log stored in the database. In case the logs match, the saving operation is safe to be continued.

4.4 Rebase on the server side

The rebase operation can be done on the server-side. The client will send the changes to the server, which will load the target XML configuration from the database and write the changes to it.

Currently, the clients will send whole XML documents to the server. That means that in most cases, a lot of unnecessary information is transported between the client and the server to save changes to a few items on the configuration. A solution for this issue could be to move parts of the XML parsing to the server side. The client would create data package containing the necessary information to inject the changes to the XML configuration on the server-side. The server implementation would still have to guarantee that two simultaneous save attempts to the same configuration do not result in a loss of data. The standard XML configuration must be locked for the duration of the parsing, and the competing clients must attempt to resend the information in case they are unable to obtain the lock to the standard. The saving logic is shown in Figure 9.

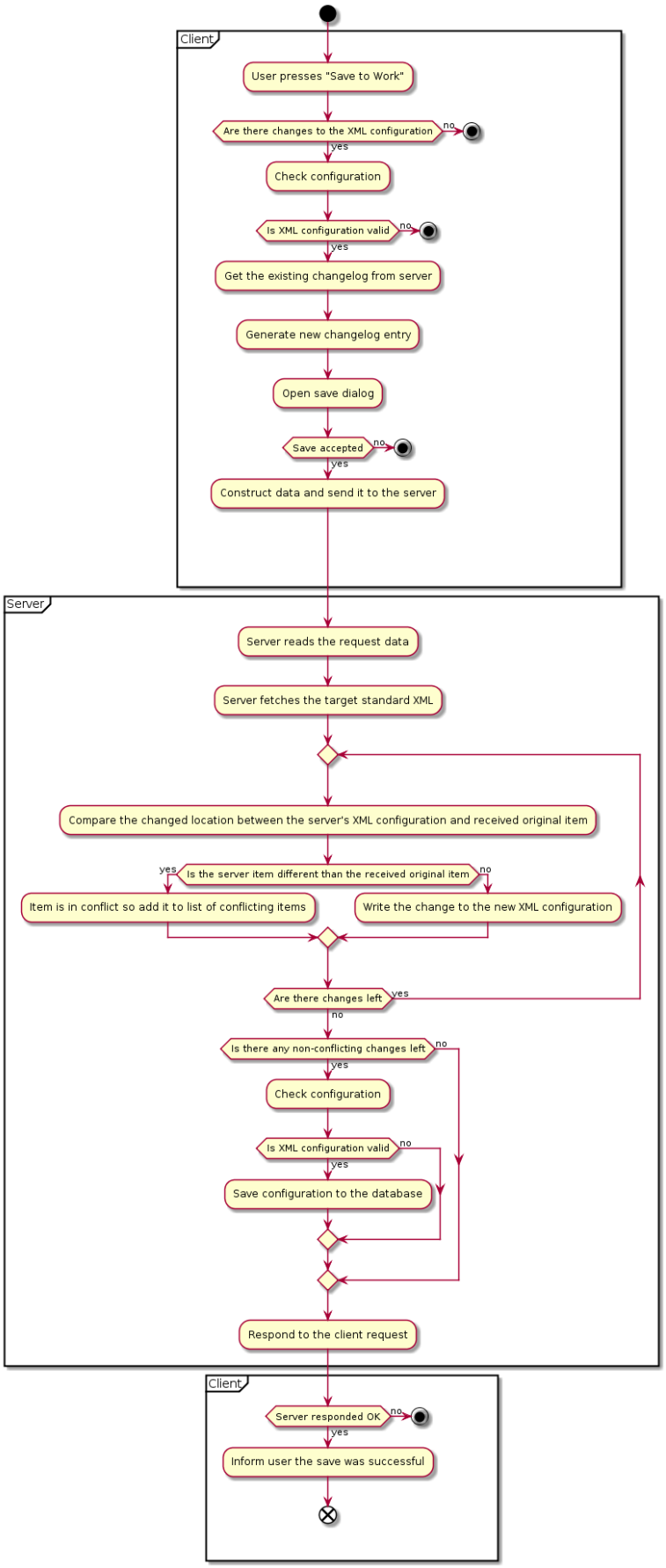

Figure 9: Save to work (server side rebase).

Major issue with the server-side implementation is the validation. The XML validations are done on the client side, either basing them on the UI configurations or being directly programmed to the application itself. If the XML merge is done on the server side, the standard must still be kept in a valid state. This problem could be partially solved by implementing a standard work described in Section 6.4. The server would allow invalid standards to be saved if they are valid against the specified schema, but this option requires larger change in the way the users use the application. If this option is chosen, the standard version numbering system must be changed so that instead of having just one work version and several released standards, each standard branch would have to have its own work version. The work version would be allowed to be invalid and multiple saves can be done to it. When the work version is published, all remaining validity issues must be resolved before publishing can be done.

The saving algorithm

In this solution, the saving logic is started with validating the configuration on the client side, because if there are errors, they must be fixed before communicating with the server. After the validation has been done, the change log is fetched from the server so that a save dialog described in Section 4.3 can be opened and the log can be shown. After the user presses the save button in the dialog, the application begins to construct the data for sending to the server. Instead of sending the whole XML document containing the standard specific data to the server, more granular data is constructed. Each change action is iterated and for each change the original node value and the new value is paired. In addition, a unique XPath query must be provided for each change [28]. The XPath query is necessary to allow the server side to write the change to correct location in the XML configuration.

Once the data has been constructed and sent to the server, the client will wait for the server to respond. The expected responses are: success, success with conflicts, failure as all changes in conflict, configuration no longer valid and standard locked for editing. The server must first determine the targeted standard version and if it is already locked for editing by another request. If it is already locked, the server should immediately respond to the client, so the user can be informed about the delay and a new save attempt can be made later. If the standard is not locked for editing, the XML configuration is fetched from the database and the parsing is started. The server must compare the original data that was received from the client in each changed location with the one found in the XML configuration of the server. If there are differences, the item is in conflict as someone has changed the same value as the user. The conflicting item is added to a list to be shown to the user when the server responds. The items that are not in conflict can be written to the XML.

The writing depends on the type of change: add, edit or remove. The items that are to be added to the configuration are not yet present, so the unique XPath query that is provided

by the client must point to the parent of the item. This is different than the edit and removal items as their XPath queries point to the item itself. The server must then create a new node to the XML configuration. To do this, the server must load the correct schema version from the database.

Removing items only requires the server to locate the correct XML node using the XPath query and then removing it from the document. In the new messages standard type, the removal operation is easy for messages, but for other types such as groups and tags, the operation becomes more complex because the validity of the XML must be maintained. The groups are singular items, but they can contain other groups, creating a treelike structure. The removal of leaf group (one that has no child groups or messages) is allowed, but if it has either groups under it or messages referring to it, the operation becomes more complex. In case of child groups, the removal should not be allowed at all and the change should be in conflict. If messages belong to the group, all the references to the group must be removed from the XML configuration. If a message would be left without a group, a reference to special group called *Deprecated* must be added. Tags do not contain other singular items, but they have similar design as the groups containing messages. On the XML level, the tags are referenced by messages and therefore if a tag is to be removed, all the references to the tag must be removed. Edit items can be fetched from the XML using the provided XPath query and a new value can be written if the change is not in conflict.

After all the changes have been either deemed to be in conflict or written to the XML, the server must then perform validation. This step can be made optional as stated earlier if the work version can be invalid, but otherwise the server must be programmed to know all the validation conditions. If the XML configuration contains errors, the saving process is stopped, and the server informs the client of errors in the result XML configuration. The user must then make local changes to save their changes. If there are no errors, the configuration can be saved to the database and the server responds success to the client. If the save was successful with conflicts, the list of conflicting items is shown to the user. The user must make said changes again and make a second save attempt to save them to the server.

If the client receives a failure response from the server, the logic will differ according to the response. If the reason for the failure is that all the changes are in conflict or the configuration is no longer valid, the user must perform additional actions before the saving can be done successfully later. If the response is that the standard was locked for editing by another save attempt, the client can wait and automatically start a second save attempt after reasonable amount of time.

4.5 Choosing solution for design and implementation

A concurrency solution must be chosen because beyond the editing of the standard in the client application, the architecture of both the client and the server will be different between the solution options. The main points for the selection are presented in the Table 1.

Table 1. *Comparison of solution designs.*

Criteria	Client-side rebase	Server-side rebase
Sent data amount		X
Code reusability	X	
Work amount	X	
More granular communication		X
Simplicity	X	

The main reason for choosing the server-side rebase is to reduce the amount data that is being sent and making the server communication more granular. As stated before, currently the whole XML configuration of a single view is sent to the server for storage. The file can be 4 MB in size, but if instead only the changes to a single message are sent to the server, the changes are at most kilobytes in size. In addition, if the XML parsing is moved or copied to the server side, the communication between the client and server can be more granular in the future if the program is modified to become more of a cloud-based service instead of a simple data storage in the server and a complex client. However, such a change would require considerable change on the users' way of working, as the standards would not necessarily be saved as a single large operation as is currently done.

Current save implementation sends the whole XML configuration to the server. The client-side rebase would still function in similar manner and thus the existing communication code can be utilized. The server API function call for saving the standards would need to be modified slightly as the server would have to be able to compare the age of the received XML configuration with the one in the database and respond accordingly if the age is the same or not. Beyond this change to the original API function, the server implementation can be used as is. As the amount of existing code that can be used is higher in the client-side rebase than the server-side rebase, the amount of work required to implement the feature is considerably less. This is directly reflected in the cost of the software and the development time.

Choosing the client-side rebase has the effect that instead of reducing the amount of logic on the client-side and moving it to the server, the complexity of the client is increased. The complex client makes adapting to changing requirements and environments more difficult, as the implementation done to the client is made available to the end users with

a release [12]. The client software must be modified, tested and redistributed to the users. If the client would be thinner, changes could be made to the server and tested there. Once the server is tested, the release can be made, and the effect will be visible to all users [12].

When comparing the activity diagrams in Figure 7 and Figure 9, the client-side rebase has a long feedback loop in case the save operation fails on the first try, but the overall execution path is linear. The server-side rebase on the other hand is more complex because the server is changed from being lean, to handling more complex operations and consider the actual data that is being stored. The server has multiple possible return points depending on the execution path that is being taken and the client must be aware of the different ways the execution is returned to it. The matter is further complicated if the conflict resolution wizard is implemented in the future. The server must then detect each conflicting item, inform client about them and the client must be able to allow the user to fix the conflicts. This means that instead of having simple requests to the server in the saving logic, the communication becomes more like a discussion. The execution is switched between the client and the server multiple times during a single save attempt.

Based on the solution proposals and their comparison, the client-side rebase is chosen as the one for further design and implementation. The most relevant reason for the choice is that if the server-side implementation were chosen, a greater redesign of the application should be done. Choosing the client-side rebase allows making general refactoring and code improvements as the implementation requires less work than the server-side implementation. Chapter 5 discusses the design and implementation of the implementation.

5. ARCHITECTURE AND DESIGN

This chapter discusses the architecture and design of the standardization tool and the new extension for messages standards. The application has been created with Qt framework. Qt is a cross-platform application framework that allows writing applications to different software and hardware platforms [21]. Qt is implemented with C++ and provides ready to use UI elements and C++ libraries, and allows the users to create both GUI and non-GUI implementations. The framework offers GUI-elements, but it is possible to write and create terminal or server console applications as well. Qt offers its own IDE called Qt Creator, but the framework can be used in other IDEs as well, such as Visual Studio for which Qt offers Qt VS Tools add-ins [22]. Qt also offers bindings to other programming languages such as C# and Java [3].

The existing application is divided into multiple plugins [8]. Some plugins are mandatory and are present in all system types and in normal use can always be assumed to be available, such as Core and GUI. The plugins are loaded on run time when they are needed. Using the dynamic link library (DLL) structure, the application can be made modular and the plugins that are not needed for a specific use case are not loaded in program memory. The creation of UI is handled in one plugin and the actual standardization tool logic is implemented in its own plugin. From here on, the plugin of the standardization tool is referred to as *STT* or Standard Template Tool. In addition to STT, a communication plugin called STTCommunication is necessary for the standardization tool to work. The STTCommunication plugin offers the API functions for contacting the server to request and send data to.

The architecture of the standardization tool is presented in Section 5.1. Section 5.2 describes the design for messages standard support in the tool. The new saving logic and its design is presented in Sections 5.3 and 5.4.

5.1 Standardization tool

On system startup, a login dialog is opened. The user can at this point choose to login to an empty system (for creating a new system), existing system (for editing it), or to STT system. If the user chooses the STT system, the STT.dll is loaded and the data is fetched from the server. Because the system architecture uses an AddinManager class to manage the plugin loading, the plugins act like singleton classes [4, 6]. The overall structure of the current STT plugin is depicted in Figure 10.

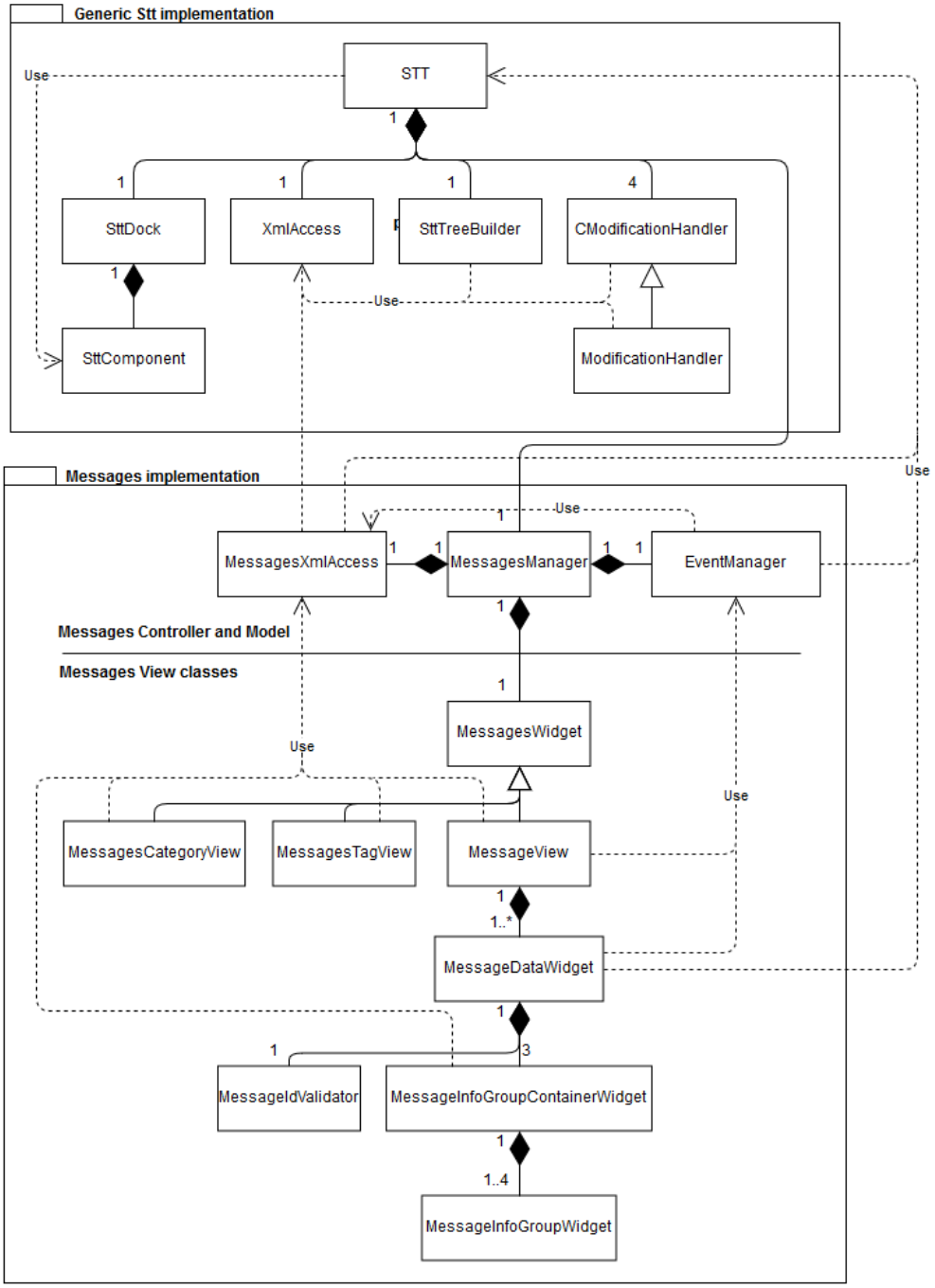

Figure 10: STT class diagram.

STT has a single interface through which requests to the plugin are made. Class ISTT is the interface class and the actual STT class is inherited from it. STT acts as the main class of the plugin and holds ownership and memory management for most of the high-level objects in the plugin. Examples of such high-level objects are the modification handlers and the actual XML configuration objects which contain the data for each different type of standard that is currently active (often work versions). *Parameters* and *Measurement details* standards are implemented mostly using generic implementation provided by the

GUI plugin and the STT, which manages the XML, logs changes and offers some standard type specific functionality. For *Communication setup* the class acts as an entry point and the actual functionality is handled in its own classes. This separation of concern between the generic STT class and standard type specific functionality will be used in the new *Messages* standard type.

The STT class and the basic functionalities of the plugin itself are quite old and can reasonably be called legacy code. Thus, a better separation of concern is an overarching goal for the long-term health of the project and the plugin [5]. The technological debt is evident in the pervasiveness of XML handling in the plugin and more broadly in the whole system. In the ideal case, the XML handling would be abstracted, and only lower level functionalities would handle it. This is not the case, as the classes use the XML class which allows for fetching and writing values to the XML document using document node notation and XPath requests. As preparative work for the subject of this thesis, a new class XmlAccess was implemented. The class will offer STT specific XML functionalities which can be used across the whole plugin. Even though the XML functionalities still leak to the higher levels of the plugin, the new XML helper class can be used so that the callers only pass along the return values of the functions without parsing the XPath requests or nodes.

A set of unit tests were also created for the XmlAccess class. This way any future implementation to the class will benefit from quicker feedback to the changes, and the overall quality of the plugin is improved [5].

5.2 Messages standard support

The STT class is already very large. It is better to separate the new messages standard support from the generic functionality of the STT class. Model-view-controller (MVC) design pattern can be used as a starting point for the design [4, 6].

5.2.1 Classes of the message standard support

The MessagesManager class acts as the controller of the design pattern. The manager is the high-level object and owns the other classes related to the messages standard. The manager is owned by the STT class. When a function call related to the messages standards is passed to the STT class, it passes the request along to MessagesManager.

A model class is needed to handle the actual data and how it is stored. Actual data is XML configuration that is loaded from the server. The existing XmlAccess class can be used as an example for a model class. The new MessagesXmlAccess class abstracts functionality such as creating a new message so that the rest of the plugin does not need to know the structure of the XML. When the XML configuration is modified, MessagesXmlAccess also informs the modification handler about the change. A generic function in

XmlAccess can be used to create a unique XPath to the target node. MessagesXmlAccess requests the unique XPath from XmlAccess and passes it to the modification handler.

View part of the MVC design pattern is divided into multiple classes. The classes derived from or owned by the MessagesWidget class act as the view part. The main application uses widget-based user interfaces of Qt [23]. The view classes are all inherited directly or indirectly from QWidget class [4]. These classes only contain logic related to showing the data to the user and interpreting the user's actions.

The Manager class owns a pointer to a view widget which can be one of the three available types: a category, a message or a tag widget. The pointed object will be replaced if the type is changed, by user selecting another item, and a new view is constructed. MessageDataWidget contains the logic for showing the data of the message in the UI. Part of the UI is further defined in the MessageInfoGroupContainerWidget and MessageInfoGroupWidget classes, which show specific fields of the message.

The data fields of messages, tags and categories can be edited. When user edits a field, the view class executes the function connected to the Qt edit signal [19]. The function then reads the data from the UI and passes it MessagesXmlAccess for storing to XML. A view of a message is shown in Figure 11. Each field is a GUI class of Qt or inherited from one.

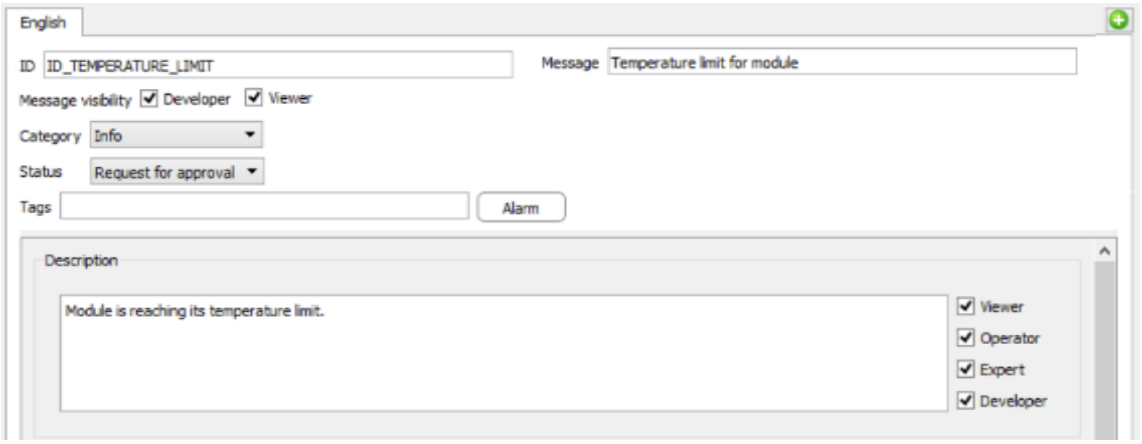

Figure 11: New messages view

Validation functionality can be added to MessagesXmlAccess. The class operates on the XML, so data checking should be done there. However, the class should not care about the results of the checks. A new class called EventManager does that. The event manager keeps track of errors in the configuration by calling the check functions of MessagesXmlAccess. If an error is found in some item, the event manager calculates a unique ID for the error, based on the item itself and which field in it has the error. The error manager can then create a new error event that is shown to the user. The usage of error manager can be divided into two: 1. checking a single item and 2. checking the whole

configuration. The first is needed when the user is modifying the standard. The second is needed when saving the standard.

ModificationHandler of the STT plugin is used to keep track and store information about the changes that have been done to the XML configuration. The standards created with the standardization tool have an XML attribute STTID (Standard Template Tool Identifier). STTID is guaranteed to be unique by maintaining a sequence on the server-side. To define the singular items (e.g. messages), each item is given the STTID attribute. Using the STTID as a condition in the XPath queries allows the program to define a single item in the configuration and is used in the modification handler. The use of STTID to define a singular item is similar in use to the UID presented by C. Thao and E. Munson in their study [16].

The new modification handler is inherited from the old one. The old handler stored each change to the same data structure with the full XPath to the edited field. This caused issues when the item has siblings with the same name. An index is added to XPath, but the index is no longer valid if the user makes an order changing edit.

A new modification handler uses the concept of a singular item as its basis. The handler has separate data structures for different change types (add, edit and remove). Each data structure contains XPaths to singular items that have changed. Since each item has a unique STTID, the index is not needed in XPath. If the change is an edit, the additional relative XPath to the edited field itself is added as additional information to XPath to the singular item.

5.2.2 Functionality of the messages standard

The system tree is shown in Figure 12. The user can open items from it by left clicking them. Some actions are found in the context menus which are opened by right clicking the items in the tree.

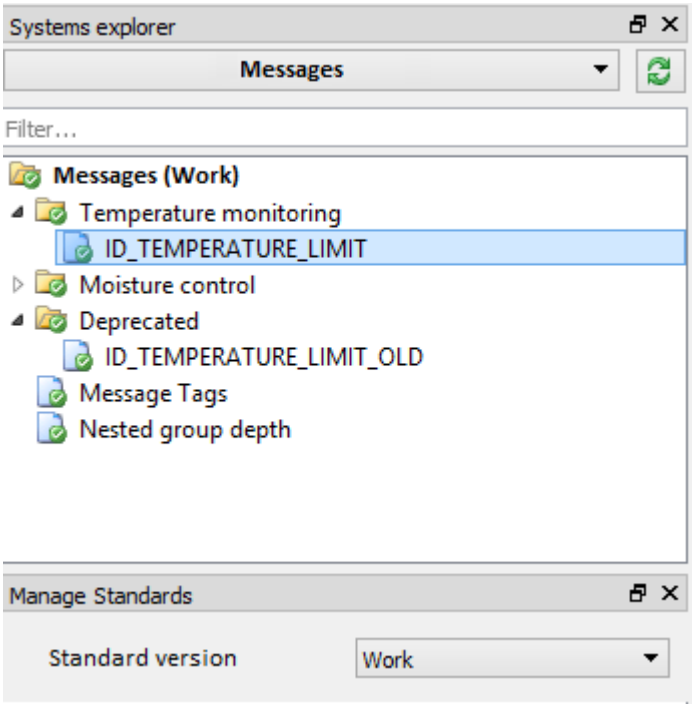

Figure 12: The messages tree explorer.

In the configuration tool, the contents of the context menu are defined in separate tree XML configurations. This functionality can be reused in the messages standards and the handler plugin for the action can be set to be STT. GUI plugin registers the action and passes the handling to STT. The execution can be seen in Figure 13. An example of creating a message is shown in the figure, but the sequence is similar with other actions.

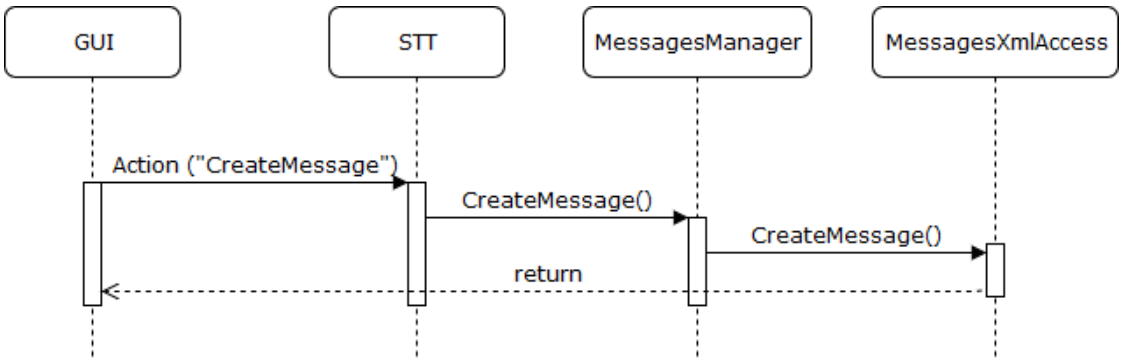

Figure 13: Sequence diagram for context menu actions.

The STT class receives the action and checks its name. If the name matches a predefined action, the execution is passed to the correct class, such as MessagesManager. This allows the functionality relevant to the messages standards to be separated into its own classes. The manager then calls the appropriate functionality, such as *create a message* or *delete a group*.

It should be possible, that a message can belong to multiple groups. The actual XML data should not be duplicated. Either a group must reference the messages that belong to it or

a message must reference groups it belongs to. Message referencing groups was chosen as the design. An example of the XML of a message can be seen in Figure 14. The BelongsToGroups structure defines the groups to which the message belongs to.

```
<Message STTID="90">
  <ID> ID_TEMPERATURE_LIMIT</ID>
  <BelongsToGroups>
    <ReferenceId>1</ReferenceId>
    <ReferenceId>2</ReferenceId>
  </BelongsToGroups>
  <MessageText>
    <Value Language="English">Temperature limit for module</Value>
  </MessageText>
  <Category>100</Category>
  <Descriptions>
    <Description>
      <Value Language="English">The module is reaching its temperature limit.</Value>
    </Description>
  </Descriptions>
  <Implications>
    <Implication>
      <Value Language="English">The temperature is too high.</Value>
    </Implication>
  </Implications>
  <Recommendations>
    <Recommendation>
      <Value Language="English">Lower the temperature.</Value>
    </Recommendation>
  </Recommendations>
  <Parameters>
    <Parameter>
      <Datatype>UnitSystem<Datatype>
      <Unit int="C" disp="C" />
      <Description>
        <Value Language="English">The current temperature</ Value>
      </Description>
    </Parameter>
  </Parameters>
  <Approval>REQUEST</Approval>
  <Tags>
    <Tag>Alarm</Tag>
  </Tags>
</Message>
```

Figure 14: Messages standard XML.

In the view building, when a message is read from the XML, the groups it belongs to are read as well. A tree item is then added to the tree to the referenced groups. The tree items can be traced back to the XML nodes they were created from. This allows fetching the specific XML node that is a target of an action, such as *delete group*.

The fields of a message that have a *Languate* attribute can be configured in multiple languages. The attribute references a known language, in this case the language is English. A new language version of the message can be added by adding a second entry to all the locations that have the language attribute. This way the multilanguage support is configured.

The messages standard has a dependency to the *Measurement details* standard. The values from that standard are used in the parameters of the message. The parameter has some unit system configured to it. The unit system can be configured directly from *Parameters* standard or taken into use from an entry in *Measurement details* standard. Changing the version dependency can be done the same way as for the other standards. The existing code is almost completely reusable and only the UI elements of the version dependency dialog must be updated. If a unit system used in a message is no longer available, the XML left as is, and an error is created from the situation (missing unit system).

5.3 Rebasing data

Rebase is separate functionality which can be called directly as a menu action or be executed as part of the new saving logic. Algorithm for rebase is presented in Figure 15. The code has been abstracted into a chart to show only relevant parts of the program and more specific functionality of it is presented in text.

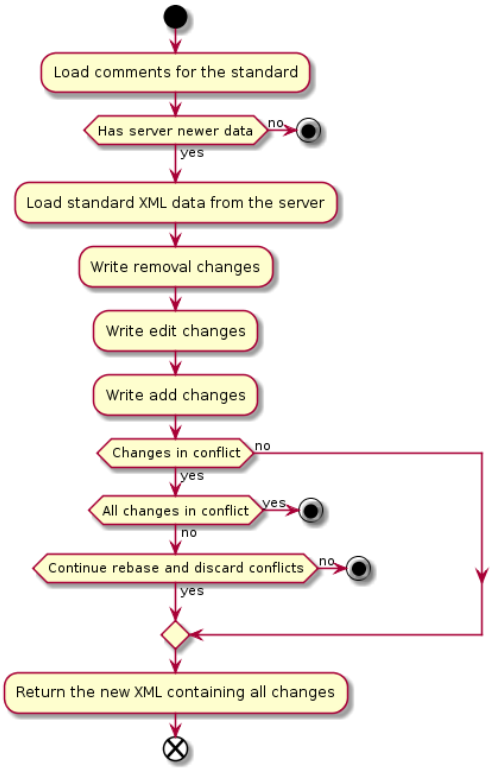

Figure 15: The rebase algorithm.

Before loading actual standard data, the client application will request the change log of the standard. If the received log does not differ from the one the client has in memory from when the standard data was loaded originally, the standard data has not changed and there is no need for a rebase operation. Otherwise the standard data is loaded from the server and two XML instances are created from it with identical contents at first. The

second copy is not needed for the rebase algorithm, but rather for the functionality afterwards. The client keeps two instances of XML in memory: 1. XML with all the local changes and 2. XML that is unchanged. The second copy will become the new unchanged XML for the client if the rebase is successful.

5.3.1 Writing changes to XML

After the XML instances have been constructed, the changes are written to one of the loaded XML copies, which will eventually be the result XML of the rebase. The writing algorithm is presented in Figure 16. Some changes may be in conflict if the same item has been changed between the data of the server and the client. If all the changes are in conflict, there is no reason to continue the rebase logic. If only some of the changes are in conflict, the user is informed about the conflicting items and asked whether they wish to continue. If the user continues, the conflicting changes will be discarded. Otherwise the execution path ends.

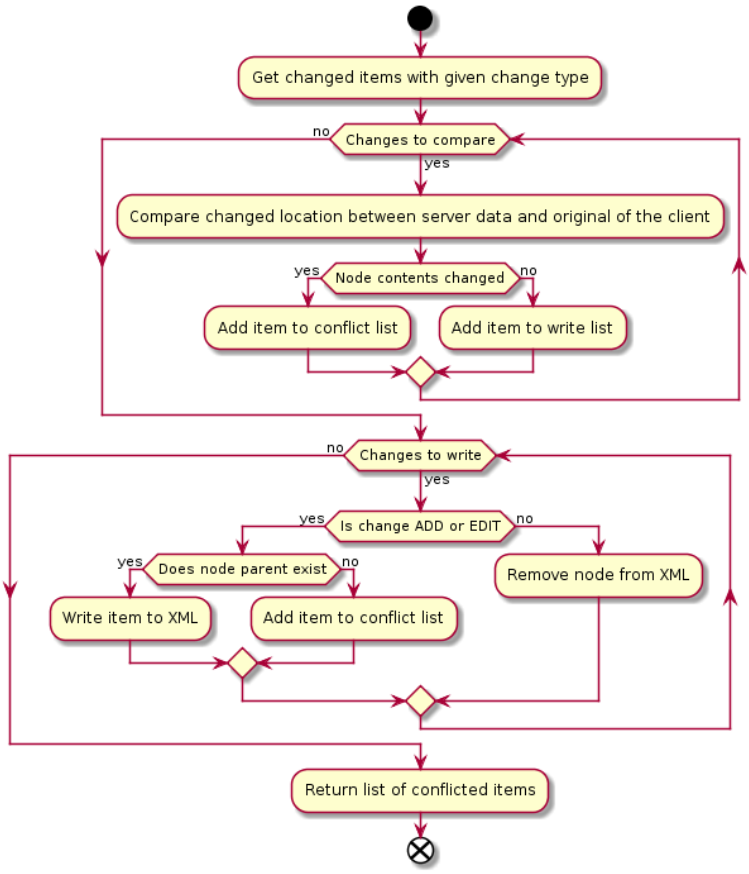

Figure 16: Write changes.

WriteChanges function writes all changes of one type at a time. The function will request all changes of given type from ModificationHandler. Using the retrieved changes and specifically the XPaths to the singular items in them, the function compares the node with

the original XML instance of the client and the data from the server. If the node is unchanged, the change is safe to write to the XML from the server. The node is retrieved from the modified XML of the client and copied to the new XML. If the node has changed, it is instead added to a list of conflicting items which will be returned to the rebase function. The compare logic is further described in Section 5.3.2.

5.3.2 Comparing nodes

Before writing the changes to the newly created XML configuration, they must be checked if the changes are in conflict. The edit and remove XPaths must be used to compare the XML configuration that was received from the server with the local original version. The compare and writing logic in the rebase implementation has similarities with three-way merging [13, 16]. The standardization tool has the original and modified version before the saving is started. Once the saving is started, the newest data is loaded from the server. This results in three different document instances. The original version the standardization tool has is the baseline, and the new data from the server and the modified version in the tool are the differing versions.

Comparison function is generic and is used elsewhere as well, but for the thesis the relevant call location is the writing of changes in rebase. CompareNode function is shown in Figure 17.

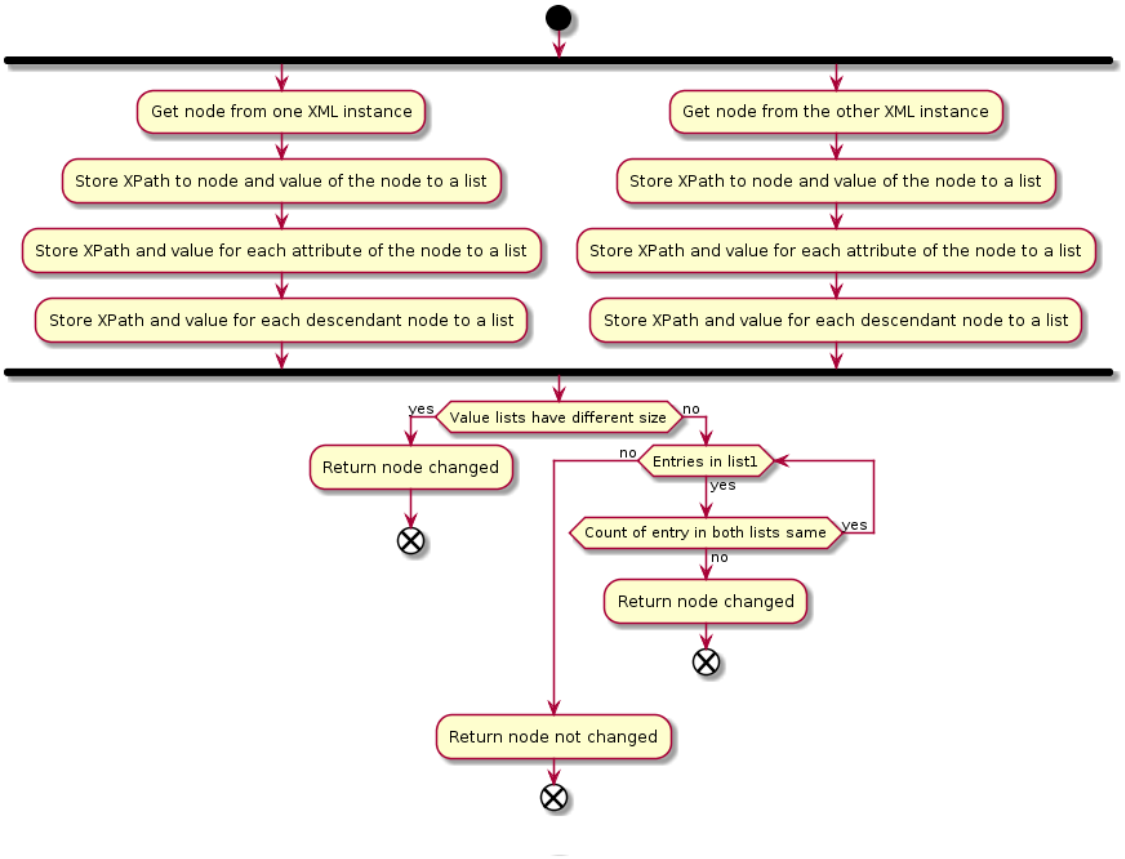

Figure 17: Compare a node in two XMLs.

First the node itself is retrieved from the XML. Then the node and its contents are parsed, and a list of its contents is created. Each list entry contains XPath to the node and the value of the node. If a node has descendants, its value is set to empty. This operation is also performed to the other XML instance, thus creating two lists of node contents.

Then these lists are compared with each other. To skip unnecessary checking in case the nodes differ in size, the size of the lists is compared and if it is different, there must be some change in the node and the function can return immediately. Otherwise the contents of the first list are looped and for each entry, the number of instances in one list must match the number in the other list. If at any point the numbers do not match, the node must have changed.

The presented comparison of node contents is not enough, because the singular items can be in a hierarchical structure where one group owns other groups. If a child group of another group has been modified, the parent group should not be considered having changed, unless the parent group is being removed. Thus, the fetching of node contents is made aware of whether other singular items that are descendants of the node that is being compared, should also be compared. If the change operation is REMOVE, the children must be compared, but for other changes the singular descendants are skipped. An example of the XML data is shown in Figure 18.

```
<Groups>
        <Group STTID="1" Name="Module responsiveness"/>
        <Group STTID ="2" Name="Safety">
                <Group STTID ="3" Name="Warnings">
                        <Group STTID ="4" Name="Module 1 warning"/>
                </Group>
        </Group>
        <Group STTID ="13" Name="Deprecated"/>
</Groups>
```

Figure 18: Message groups XML.

The side effect of the XML structure is that when comparing a group like the *Safety* shown in the figure, change to it such as its name, should not conflict with changes done to the name of another group such as *Warnings*. However, if the change targeting the higher-level item is a removal, the group should not be removable. Such a situation may be created if as a starting point the group *Safety* has no child groups and two users start to simultaneously edit the standard. One wishes to remove the group, and the other wishes to add a child group to it. The situation should not be encountered in normal application usage, but the sanity of the system and usability should still be guaranteed. If the user who is adding the child group saves their changes first, the other user should not be allowed to remove the group, as the group now has a child group. First, they must explicitly remove the child group and only then are they allowed to remove the parent. If the user who is removing the group makes the save first, the addition of child group must fail.

Before starting data construction for sending to the server, the result XML configuration from the rebase operation must be validated. Changes done to it during the operation may have caused issues such as duplicate IDs. For *Parameters* and *Measurement details,* the checking is mostly done based on the view configurations, but *Communication setup* and the new *Messages* are instead programmed to the application itself, because the views are custom built instead of being built from separate view configurations. If validation errors are found, the saving process is stopped, and the user is informed that there are errors that must be fixed. The stopping of the saving process and even the whole validation can be removed in the future if the proposal for allowing erroneous work in progress standards will be implemented. In case there are no validation errors, the saving process continues to the server communication.

5.4 Sending data to the server

When the user starts a saving operation, the STT class first performs a rebase operation. If the operation is successful, a save dialog is opened for the user.

After the user confirms the save in the dialog, the construction of a message to the server is begun. XML configurations that have been loaded to the client system contain more

than just the XML configuration of the standard. In addition, they contain basic definitions and possibly their dependency data. An example of the highest XML levels for the *Messages* standard document can be seen in Figure 19.

```
<Root>
    <Profiles>
        <Profile>XSP</Profile>
    </Profiles>
    <XSP>
        <BasicDefinitions>
            <!-- Basic definitions data -->
        </BasicDefinitions>
        <MeasurementDetails>
            <!—Measurement details standard data -->
        </MeasurementDetails>
        <Messages>
            <!-- Messages standard data -->
        </Messages>
        <Parameters>
            <!-- Parameters standard data -->
        </Parameters>
    </XSP>
</Root>
```

Figure 19: A high level Messages standard XML configuration.

The extra information in the XML configuration is necessary for the client application to operate, but there is no reason to store duplicate data to the server in case the standard has dependencies to other standards, like in Figure 19 the *Messages* standard XML configuration also contains the data from *Measurement details* and *Parameters* standards. After the data has been validated after the rebase operation, the node containing the actual standard data is extracted from the document. Then it and necessary version information is passed to the communication plugin for sending to the server. In addition, the current change log history of the standard must be passed along as well so that the server can determine is the received data acceptable or too old.

Communication is done using SOAP protocol and the standard data and the change log are stored as part of the message [24]. The communication plugin will also read the communication configuration file that is part of the application. The file defines the URL to which the data is sent to, timeout limits and the authentication information. Using the read values, the request message is sent to the server.

The functionality of the server is presented in Figure 20. The code is split into multiple functions which are called, but for clarity, the logic is presented with higher abstraction.

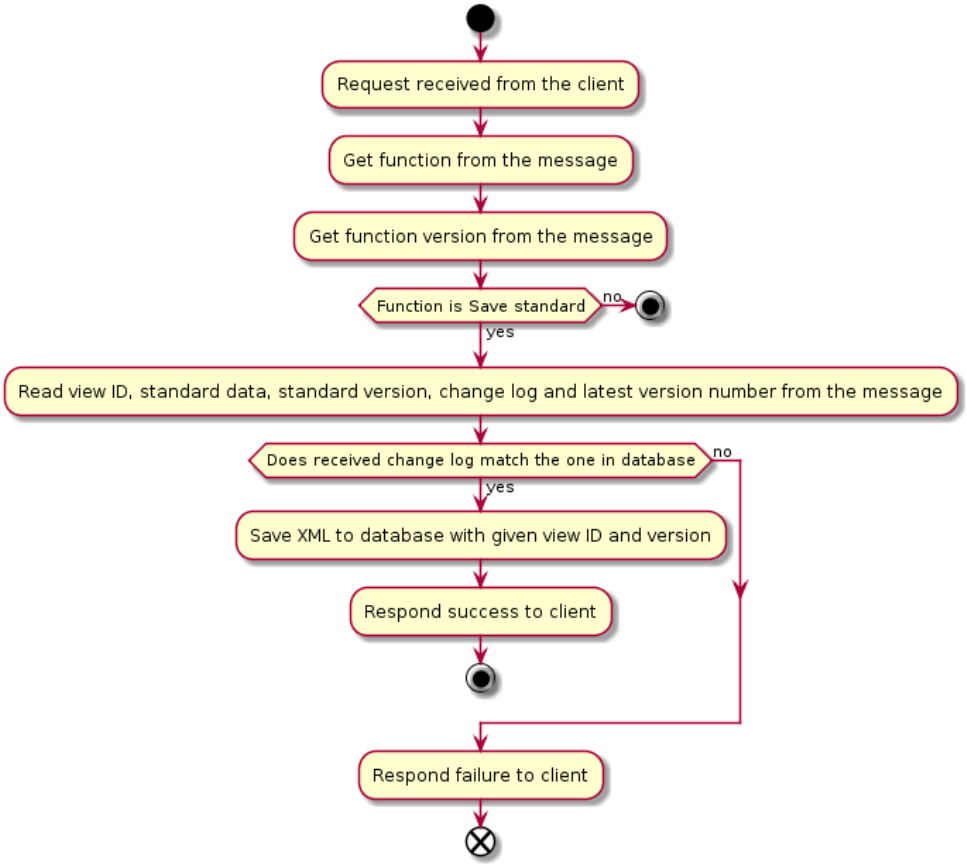

Figure 20: Functionality of the saving on the server-side.

The server will receive the request and create a service instance for handling it. The service will parse the request message and check if it contains all the necessary data. Then the API function to perform is read from the message and the functionality is passed to proper handling function. For simplicity, only the execution path of saving standard is depicted in Figure 20.

The server will then retrieve the change log of the standard version from the database and compare it with the one received from the client. The change log is used to determine if the standard data has changed during the saving operation as described in Section 4.3. The function will parse the actual standard data from the message (it has its own field) and store it to the database. The new log entry is added to the change log and saved to the database. Afterwards, the server responds to the client that the operation was successful.

The server can respond with two different responses: 1) success, and 2) standard data changed. In case the first response is received, the client will inform the user the save operation was finished successfully. If the second response is received, a new save attempt is made.

For the new saving logic, the saving function needs to be modified to allow the new logic to work. Reason for this is to support older standardization tool versions, so the server must offer the older version of the function for requests from those applications.

6. IMPLEMENTATION AND EVALUATION

The implementation was done during 2017 and 2018. A test version including the new messages standardization tool and the concurrent saving logic has been provided for the end users for testing.

The changes to the design are presented in Section 6.1. Following it in Section 6.2 the result of the project is compared with the set goals from Chapter 3. Next, in Section 6.3 the advantages and disadvantages of the solution and tool are evaluated. Lastly, the future implementations are presented in Section 6.4.

6.1 Changes to the design.

Implementation of the messages standard tool followed mostly the design presented in Section 5.2. A change to how the items reference other items in the configuration was changed.

The messages reference the groups they belong to using the groups unique STTID. Language and the tags of a message however were referenced with the value of the item, such as English for language as seen in Figure 14. The reference logic was changed to be uniform and done using the STTID attribute. The Language attribute was changed to LanguageId and Tag element was changed to ReferenceId. This allows easier updating when the referenced value is edited. If a message has a reference to *Alarm* tag, and the user changes the name of that tag, all the references would have to be updated as well. This can be skipped with referencing the items with STTID as the attribute does not change.

The new save logic implementation can be divided into three high-level parts: refactoring the existing solution to support the new implementation, the implementation of the rebase logic and the implementation of the actual server communication. The implementation followed mostly the design presented in Sections 5.3 and 5.4, with some changes.

Checking whether the added item is safe to add could not be done during the conflict checking (before performing the XML writing). Reason was that it was not possible to determine whether the node where the new item would be added to existed in the standard before but was removed or was added in the same session. Solution for the issue would have been to create a lookbehind logic in the conflict checking to check whether the parent was added in the same session if it does not already exist in the XML configuration.

The lookbehind logic is unnecessary, because the check could also be performed during the XML writing. The changes are written in order of XPath query depth, meaning that parent nodes will always be written before the child nodes. Therefore, if the parent of an

added item does not exist during the writing step, the parent must truly be missing, and the item is in conflict.

In the design, it was deemed enough to pass the change log to the server to ensure no accidental data overwriting could occur. When a user releases a new standard version from the work version, the change log of the work is cleared. At first, this was not seen as a problem because the change log comparison either matches the logs as empty (standard has been released and no further changes have been made) or non-empty and normal comparison can be done. This left a situation incorrect, if a user makes a second standard release after the first one. Steps of the situation are: 1. a new standard branch has been created from the Work version and the log has been cleared. 2. One user starts making changes and starts the saving process. 3. The second user makes a change to Work version and immediately releases another standard branch, before the first user finishes the save in step 2. 4. The first user's save request is being handled on the server-side and an empty log is passed there.

The database contains an empty log as the second user had made a change and released a new standard. Therefore, the XML configuration in the database has changes that are not found in the request and the save should not be allowed. The server did not notice this because the change logs were both empty and thus considered equal. The save was finished, and changes made by the second user were overwritten in the database. The situation is very unlikely, but a solution must be created. The client will send the latest standard version the edited standard type has according to it. The server will then compare both the received log and the received standard version. If the logs differ or a newer major standard version has been released, the server data has changed, and the server responds to the client with failure, after which the client will start a new save attempt.

Another change is to limit the automatic save attempts to the maximum of three. If the limit is exceeded, the process is cancelled, and the user is informed about the situation. The limit is not mandatory for the save functionality because the user is able to cancel the process. The limit was implemented because unnecessarily long operations are not user friendly.

6.2 Realized goals

Some of the goals from Chapter 3 were met only partially. The development process was started from the basic functionalities and the saving logic which is the main question of this thesis. The goals and the results are presented in Table 2 below. The table has three columns: first one for the actual goal, second whether the goal was reached, partially done or not reached, and the chapter where the goal was defined. The comparison was done on March 2018 using the program version which was the same as the second test version which was provided for the users.

Table 2. *Thesis goals and their realization.*

Goal	Realized	Chapter
Messages standard in tool	Yes	3.1
System tree shows messages & groups	Yes	3.1
Message belongs to multiple groups	Partially	3.1
Context menu: Add/Edit/Remove	Yes	3.1
Message is editable	Partially	3.1
XML validation	Partially	3.1
Standard dependencies	Yes	3.1
Standard API support older application versions	Yes	3.2
Data loading refactoring	Yes	3.2
Concurrent saving	Yes	3.3
Conflicting items are discarded or save cancelled	Yes	3.3

When user logs into the standardization tool, they can choose the *Messages* standard type from the system tree explorer. The system tree contains the data structures, with group structure where the messages can be found. A single message can belong to multiple groups and the message is shown correctly in the system explorer. Currently, the user cannot manage the multiple groups, and thus the feature is only partially implemented. The system supports showing existing messages belonging to multiple groups, but the user cannot configure a message to belong under multiple groups.

The items found in the system tree explorer can be right clicked, which will open a context menu with actions related to the clicked item: Add and remove for both messages and groups, edit for groups and show the underlying XML action are found for the items. Also, when the message items are left clicked in the system tree, the configuration view for the message is opened. Messages data is displayed in different fields and the user can edit the message if they have the correct user profiles. Currently, it is not possible to configure parameters to the messages, and therefore the message editing is only partially implemented.

Custom event manager for *Messages* standard is partially implemented. The system notices errors when the erroneous message or group is opened, but the full configuration check has not yet been implemented. Therefore, the configuration check found in saving does not yet automatically ensure the configuration is valid after rebase operation. When implementing the rebase operation, the validation had to be done manually.

The user can change the version dependency of *Messages* standard and the process will change the dependency standard data on the XML configuration. The dialog is like the

dependency dialog of the existing *Communication setup* standard, and the implementation could be reused as is with minor changes. Validation of the dependency change is not possible because of the missing configuration check.

Server communication for the new saving logic was implemented as a new API function version, and the old functionality was preserved as is. This was done to ensure that saving other standard types works with current and older application versions. The new saving logic can be taken into use in other standards in the future. Data loading on the client side was modified, and as it is generic functionality, the changes affect the other standards. The implementation was done first before the saving logic was implemented to ensure the existing standard types were not unintentionally affected by the refactoring.

First working version of the new concurrent saving logic was implemented and will be available to the users in the second test version of the application. The logic allows the users to save their changes even if the server data has been changed if the changes do not target the same items. The functionality also handles concurrency issues where one user has managed to make a save during the second user's saving operation, between the rebase and server communication. This way the data integrity on the server side is ensured and the changes do not accidentally override each other. The process was automated, and if the first attempt fails as the data has changed on the server side, the application starts the saving process again.

If the rebase operation notices a conflicting item or items, the user is informed about the situation and asked whether they wish to continue the operation (in which case the conflicting changes are discarded) or to cancel it and preserve the local data as is. The functionality fulfills the basic requirements, and the implementation was done so that possible conflict resolution wizard can be added to the logic in the future.

6.3 Advantages and disadvantages of the solution

The created solution allows the users to create messages standards. The standards reduce the amount of repetitive manual work. When the importing logic is implemented, the users can take the created standards into use. The time to create a system configuration by hand can be weeks or months. By reducing the amount of manual work and importing ready-made standards, the amount of work can be reduced to days or weeks.

Added benefit of automating the process with the standards and their importing is the reduced number of user mistakes. If user had to create thousands of items in the configuration, the risk of a user error increases. By importing ready-made standards that have already been validated to the system, the software does the repetitive work. This improves the quality of created system configurations and allows the users to allocate more time and resources to other tasks.

Without the standards, the users would have to manually create the basic data structure definitions such as unit systems. Then they would have to create the measurement items and what kind of unit systems they use. Following that, they would have to define communication addresses to the system and define the measurement items to individual addresses. Then the user would have to define the messages that the system can send and read. With the standardization tool, the user can simply import the different standard types one by one and skip the presented manual work.

The new saving logic improves the usability of the standardization tool. Before, more than one user could not edit the same standard without the other users losing the effort they had made. With the created solution, the users can edit the same standard at the same time without a considerable loss of work. An exception to this is if the users were to edit the same items. In normal workflow, this should not happen.

By making the conflict situation item-level, meaning that a conflict is created if two users edit the same item (for example a message), the user experience was improved. Although the item-level design can in theory increase the amount of conflicts, the gain in user experience outweighs the risk. The users do not have to be experts on XML, but rather experts in their own field. If a conflict resolution wizard is created, it can show the conflict in user friendly manner and further improve usability.

Added benefit of the new saving logic is that users' cooperation can increase. Instead of waiting for one user to finish their change, both users can make their changes at the same time. Similar logic can be found in online office tools offered by Google [7]. The users can create different documents and edit them at the same time with other users. The standardization tool differs from them in that the users cannot see what other users are currently editing. The main reason for this is that the solution is mostly client based and the server is thin [29].

Because the chosen solution for concurrent saving was the client-side option, the architecture of the system remains heavily on the client-side. This means that further improvements to the user coordination become more difficult. As the client is fat and handles all the business logic, any coordination between clients would have to be done in three-way communication from one client to the server to another client. This is different in fat server implementations. The clients could be very simple, and the server could contain the business logic. Showing information about other clients would be simpler, as the information would be stored to the server.

Updating the software is another issue of the fat client and that the new saving logic was done on the client-side. Improvements and bugfixes that are done after the standardization tool has been published will not be available for the users. Only when a new version of the tool is created, will those improvements be made available. If the solution would be more on the server-side, the server could be updated. The new functionality would be

available for existing users immediately if the logic does not require changes on the client-side.

The new save logic is very specific to the use case and the solution environment. A lot of decisions in the design were influenced by the existing functionality. The combination of unique identifier (STTID) and active tracking of changes is a novel idea not presented by the existing solutions presented in Section 4.2 where algorithm calculated a delta from the XML documents. In the new save logic, modification handler already tracks changes to the XML configuration. Using it as part of the save logic allowed skipping the calculation of deltas when the actual rebase and save are done. The solution is also an example of where an XML configuration merge is done on a higher abstraction level than in the existing solutions in Section 4.2.

6.4 Future implementation

During the design and implementation, several ideas were created about further developing the application. Most prominent ones are presented here.

The minimum requirement for the concurrent saving was to implement logic for optimistic situation where items never or rarely conflict and therefore the conflicting items can be discarded and redone after the save has been completed. This logic can be replaced by implementing a conflict resolution wizard, whereby the user could resolve the conflicting items. The wizard would have to allow the user to view all the conflicting items, the value stored in the server and the local values, and a reason to the conflict. Then the user could fix the conflict by providing the correct result. As with the saving, the end user is not necessarily aware of XML or its usage and therefore the wizard must abstract the items to be relevant to the user (messages, groups, etc.). Using the wizard, the usability of the saving is improved, as all kinds of items can be changed in a single save attempt. Also, the conflicting items become more visible to the user.

Currently, the new concurrent saving is limited to the work version of the standard. The reason is that the way the versions of the standards are numbered, where each change is its own standard revision. The version numbering can be changed by introducing a standard work concept, where each standard major branch would have its own work version. That version would not be released and would not be available for the users who import the standards into system configurations. The users could make multiple saves to the standard work, like the current work version. When the version is deemed to be acceptable, it would be published, and a new revision is created. This way unnecessary standard revisions can be reduced, as each small change would not require its own version.

The new modification handler is currently only used in the new *Messages* standard implementation. The existing *Parameters*, *Measurement details* and *Communication setup*

standards still use the old handler which has several known issues. Of the existing standards, *Measurement details* and *Communication setup* items can be modified to use the handler by adding the support for handling the older type XPaths where the unique STTID is not located on the item itself, but one of its children. In addition, the handler must be extended so that a change log entry can be created from the XPaths the handler has stored. *Parameters* standard type is more complex, because it has the greatest number of different items which have the STTID but that may also contain a varying number of items which have the STTID. Unlike the other standards, logic for importing the *Parameters* uses the item STTIDs to update references to the items. Therefore, changing the structure of XML and the location of the STTID attributes is not acceptable.

Like the modification handler, the new saving logic can be taken into use in other standard types with some modifications. The implementation requires using the new handler, and therefore the prerequisite must be fulfilled before this feature can be implemented. Considerable work effort is required when defining the singular items in the other standard types, and how they can become conflicted. This implementation would also break compatibility for modifying the work standard with older standardization tool versions.

Standard importing is planned, but it was not implemented as part of this thesis. The implementation will allow the end users to utilize the created messages standards by directly importing them from the server. Another way of using the *Messages* standards is implementing an export functionality into the standardization tool, which allows creating XML files containing the relevant data for using in a system. The user could then copy the file to relevant location for usage in configuration tool. Currently, the messages used by the system are stored in a separate file, so the export functionality is a smaller feature to implement as opposed to a full-fledged import.

Currently, *Messages* standard supports configuring messages in English only. The XML has been designed to support multiple languages, but the standardization tool does not have support for creating and managing multiple language versions. One of the goals of this thesis was to offer only English support with other features available to the users, thus allowing the users to give feedback for the created functionalities. Logic for adding other languages and configuring messages with them is needed for the full language support.

7. CONCLUSIONS

A message standardization tool was designed and implemented in this thesis. The main research topic was: concurrency issues when saving standards with the tool.

First the background and the environment of the application were introduced. The standardization tool is an extension to a configuration tool that is used to create industrial system configurations. Template XML configurations are created to reduce the amount of repetitive work when creating a new system configuration. Common data structures are defined in the template XML configurations. The templates are then loaded from a server to become part of the system configurations. This reduces the amount of repetitive work.

Next, the goals of the thesis were presented. A new standard type support (messages) was to be added to the standardization tool. The users can create messages standards in the standardization tool. The standards would contain messages and groups where the messages can be found. The user can add, edit and remove the messages and the groups.

The standardization tool has concurrency issues when saving changes to the template XML configurations. If more than one user attempts to edit a template, only one can save their changes. The other user must reload the data and make the changes again. Before this has not been considerable issue because the number of users has been low. The number is expected to increase with the addition of new standard type support. Therefore, a new saving logic is to be designed and implemented. Logic is to allow multiple users to save their changes at the same time.

Next, the theory of the concurrency issue and existing solutions were presented. Git version control system allows the users to work on the same file and combine their changes. Git has two relevant functionalities: merge and rebase. Git supports branching different versions of a file. One branch can be rebased on top of another, combining the changes in both. This logic can be used as an example to solve the concurrent save issue. The new save logic has a rebase functionality, where the local changes are rebased on top of a fresh XML configuration.

The tool has two copies of the XML configuration, the original unchanged one and the modified one. The server has a third version. The original XML configuration in the tool is the common point in the change history of the XML configuration. The other XML configuration in the client is a new branch of history. The XML configuration on the server is the main branch.

Other existing solutions to merge XML documents were introduced as well. Using version trees and UIDs to detect changes and calculating context finger prints for nodes were ways to implement an XML merge. The existing solutions could not be used directly as

the end users are not necessarily familiar with XML and the elements representing a single message instance. Therefore, in case a conflict situation is encountered, more relevant information should be shown to the user.

Two solutions were proposed for the new save logic: 1. client-side rebase and 2. server-side rebase. The client-side rebase was chosen for further design and implementation because the costs of implementation were lower and overall logic was simpler.

Using Git rebase and existing XML merge algorithms as an example of rebase and merge improved the speed of the design and eventually implementation. Implementation costs were reduced by choosing the client-side solution for the rebase as greater amount of existing code and logic could be used. This allowed improving the functionality and designing a more extensible solution. In addition, the implementation logic became simpler, as the logic could be divided into two separate parts: rebase and server communication. In the client-side solution, the rebase was done completely in the standardization tool with only existing data loading from the server being used. This allowed the implementation to be split into smaller items which could be implemented and validated separately.

Most of the goals for the thesis were achieved. Only three goals were partially met: 1. a message belongs to multiple groups, 2. a message can be edited and 3. standard can be validated. The standardization tool supports showing a message in multiple groups. The users cannot configure the message to belong to multiple groups, so the goal is only partially reached. Messages can be created and removed. The messages can mostly be edited as well. Only the parameters of a message cannot be configured. Thus, the goal is only partially met. Currently, a message can be validated when it is opened. But a whole configuration validation is not yet implemented. This means that the new save logic does not validate the configuration.

The new save logic allows more than one user to edit a standard. This allows greater cooperation between the users, as they can edit the same standard at the same time instead of having to wait one user to finish their changes. It will also reduce the amount of lost work, as the tool can automatically rebase their changes. The conflict situations were evaluated to be uncommon. If a conflict is found, the relevant change must be discarded to save the changes. This situation can be further improved by implementing a conflict resolution wizard. The saving logic can be extended with further logic as well as extended to the existing standard types in the future.

Currently, the new extension of the standardization tool is only partly useful to the end users, as importing the data directly to the system configuration is not available yet. This means that there is no easy way for the end users to take the standards into use when creating system configurations. But, as the goals of the thesis were mostly met, the created solution is successful.

REFERENCES

[1] S. Chacon, B. Straub, Pro Git, 2nd ed. Apress, 2014, pp. 12-13, 70-98, ISBN 1484200772.

[2] K. Dick, XML: a manager's guide, Addison-Wesley, 2003, pp. 22-26, ISBN 0201770067.

[3] Digia Oyj, Language Bindings - Qt Wiki. Available (accessed 12.01.2018): https://wiki.qt.io/Language_Bindings.

[4] A. Ezust, P. Ezust, Introduction to Design Patterns in C++ with Qt 4, Prentice Hall, 2007, pp. 203-208, 238, 361, 392, ISBN 0131879057.

[5] M.C. Feathers, Working effectively with legacy code, Prentice Hall Professional Technical Reference, 2005, pp. xv-xix, 3-12, 249-264, ISBN 978-0-13-117705-5.

[6] E. Gamma, E. Gamma, Design patterns: elements of reusable object-oriented software, Addison-Wesley, Reading (MA), 1994, pp. 12-16, 127-132, ISBN 0-201-63361-2.

[7] Google, Google Docs. Available (accessed 19.05.2018): https://www.google.com/docs/about/.

[8] Microsoft, What is a DLL?. Available (accessed 12.01.2018): https://support.microsoft.com/en-us/help/815065/what-is-a-dll.

[9] J. Nielsen, Usability engineering, Academic Press, San Francisco (CA), 1993, pp. 123-132, ISBN 0-12-518405-0.

[10] Oracle, Java Software. Available (accessed 12.01.2018): https://www.oracle.com/java/index.html.

[11] Red Hat Inc., JBoss EAP. Available (accessed 12.01.2018): https://developers.redhat.com/products/eap/overview/.

[12] G. Reese, Database Programming with JDBC and Java, 2nd ed. O'Reilly & Associates, Inc., 2000, pp. 131-132, ISBN 1-56592-616-1.

[13] S. Rönnau, U.M. Borghoff, XCC: change control of XML documents: An Efficient and Reliable Framework for XML Diff, Patch, and Merge, Computer Science - Research and Development, Vol. 27, Iss. 2, 2010, pp. 95-111.

[14] S. Rönnau, U.M. Borghoff, Versioning XML-based office documents: An efficient, format-independent, merge-capable approach, Multimedia Tools and Applications, Vol. 43, Iss. 3, 2009, pp. 253-274.

[15] Software Freedom Conservancy, About Git, Free and Open Source. Available (accessed 19.01.2018): https://git-scm.com/about/free-and-open-source.

[16] C. Thao, E.V. Munson, Using versioned trees, change detection and node identity for three-way XML merging, Computer Science - Research and Development, 11/2014.

[17] The Apache Software Foundation, Apache Subversion. Available (accessed 12.01.2018): http://subversion.apache.org/.

[18] The PostgreSQL Global Development Group, PostgreSQL. Available (accessed 12.01.2018): https://www.postgresql.org/.

[19] The Qt Company, Signals & Slots | Qt Core 5.10. Available (accessed 12.01.2018): http://doc.qt.io/qt-5/signalsandslots.html.

[20] The Qt Company, QMap Class | Qt Core 5.10. Available (accessed 12.04.2018): http://doc.qt.io/qt-5/qmap.html.

[21] The Qt Company, Qt | Cross-platform software development for embedded & desktop. Available (accessed 20.03.2018): https://www.qt.io.

[22] The Qt Company, Qt Visual Studio Add-in 1.2. Available (accessed 20.03.2018): http://doc.qt.io/archives/vs-addin/index.html.

[23] The Qt Company, User Interfaces | Qt 5.10. Available (accessed 13.04.2018): http://doc.qt.io/qt-5/topics-ui.html.

[24] W3C, SOAP Specifications. Available (accessed 12.01.2018): https://www.w3.org/TR/soap/.

[25] W3C, About W3C. Available (accessed 12.01.2018): https://www.w3.org/Consortium/.

[26] W3C, Extensible Markup Language (XML). Available (accessed 12.01.2018): https://www.w3.org/XML/.

[27] W3C, Schema. Available (accessed 12.01.2018): https://www.w3.org/standards/xml/schema.

[28] W3C, XML Path Language (XPath) Version 1.0. Available (accessed 12.01.2018): https://www.w3.org/TR/1999/REC-xpath-19991116/.

[29] S.C. Yadav, Introduction to Client Server Computing, New Age International Pvt. Ltd., Publishers, Daryaganj, 2009, pp. 4-5, ISBN 9788122428612.