

Quality Assessment Procedure for IEC 61131-3-based Control Software for Machine and Plant Manufacturers

Juliane Barbara Fischer

Vollständiger Abdruck der von der TUM School of Engineering and Design
der Technischen Universität München zur Erlangung des akademischen Grades einer
Doktorin der Ingenieurwissenschaften (Dr.-Ing.)
genehmigten Dissertation.

Vorsitz: Prof. Dr.-Ing. Johannes Fottner

Prüfer*innen der Dissertation:

1. Prof. Dr.-Ing. Birgit Vogel-Heuser
2. Assoc. Prof. Dr. Elisabet Estévez Estévez
3. Prof. Dr.-Ing. Ina Schaefer

Die Dissertation wurde am 21.06.2022 bei der Technischen Universität München eingereicht
und durch die TUM School of Engineering and Design am 04.10.2022 angenommen.

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.de> abrufbar.

Quality Assessment Procedure for IEC 61131-3-based Control Software for Machine and Plant Manufacturers

Autorin:

Juliane Barbara Fischer

ISBN: 978-3-96548-155-8 (Print)

ISBN: 978-3-96548-156-5 (E-Book)

1. Auflage 2023

Cover: sierke MEDIA, Göttingen

© 2022 sierke VERLAG
sierke WWS GmbH
info@sierke-verlag.de
<http://www.sierke-verlag.de>



Das Buch einschließlich aller seiner Teile ist urheberrechtlich geschützt. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Verlags unzulässig und strafbar. Der Nutzer verpflichtet sich, die Urheberrechte anzuerkennen und einzuhalten.

La manera más efectiva de hacer algo es hacerlo.

(The most effective way to do it, is to do it.)

Amelia Earhart

Acknowledgments

First of all, I want to thank Prof. Birgit Vogel-Heuser for giving me the chance to pursue a Ph.D. under her supervision. She gave me the freedom to work on my research projects independently but with continuous encouragement, support and advice. I am deeply thankful that she kept constantly challenging me, which enabled me to grow beyond myself professionally and personally.

Moreover, I want to especially thank Prof. Elisabet Estévez Estévez and Prof. Ina Schaefer for their agreement to examine my work. Furthermore, I want to thank Prof. Johannes Fottner for chairing the examination committee.

I would also like to thank Prof. Marga Marcos for all the fruitful discussions we have had over the years ever since she gave me the opportunity to write my master's thesis at her institute in Bilbao.

Throughout my time at the institute, I had the pleasure to work with many motivated students who significantly contributed to my research through numerous discussions and prototypical implementations. I hope that all of you enjoyed and benefited from our collaborations as much as I did. Of all the students I want to especially thank Simon Parigger, Fabian Haben, Jan Wilch, Anja Berscheit, and Christoph Huber.

I am very grateful to all my colleagues and former colleagues for numerous discussions during coffee breaks (not always of scientific nature). Their motivation, support and advice helped me to keep going even in stressful times. I would like to say a special thank you to Emanuel Trunzer, who encouraged me and gave constructive, honest feedback at all times, to my office colleague Eva-Maria Neumann for always being supportive and tirelessly showing my doubting perfectionism that the glass is half full, to Suhyun Cha for our virtual co-working sessions, to Frieder Loch for cheering me up in stressful times and to Dorothea Pantförder for her constant helpful advice. Moreover, many thanks to Thomas Mikschl for his great support with various implementations in the projects I was involved in.

Furthermore, I would like to thank all industry partners I worked with for their continuous feedback. They enabled me to develop and evaluate the concept proposed in this thesis with real industrial control software projects while considering industrial challenges and boundary conditions.

In addition, I want to thank Heinz-Gerd, who had a guilty conscience when he didn't make me feel guilty on Mondays because I still had not set myself a deadline to complete this work.

Finally, I want to thank my family and friends (especially the Cesation), who supported me at all times, always believed in me and were understanding and patient when I was busy.

Table of Contents

1.	Introduction	1
1.1.	Motivation for Software Assessment with Static Code Analysis	1
1.2.	Structure of this Dissertation	3
1.3.	Delimitation to Other Research Work in the Scope of Various Research Projects	3
2.	Field of Investigation	5
2.1.	Industrial Automation	5
2.2.	Background on the Domain of Automated Production Systems	7
2.3.	Reuse Strategies for Control Software.....	10
2.4.	Static Code Analysis and Software Metrics	13
3.	Requirements for a Software Quality Assessment Procedure with Static Code Analysis.....	17
3.1.	Applicability in the Domain of Automated Production Systems.....	17
3.2.	Procedure Application by Software Developers including Adaptations to Company-specific Boundary Conditions.....	18
3.3.	Conducting Static Code Analysis for Quality Assessment.....	20
3.4.	Use of Analysis Results to Derive Recommendations for Action.....	22
3.5.	Limitations of the Concept's Scope.....	23
4.	State-of-the-Art.....	25
4.1.	Static Control Software Analysis and Software Metrics	26
4.1.1.	Static Code Analysis targeting Control Software	26
4.1.2.	Overview of Software Metrics Focusing on Control Software	33
4.1.3.	Commercial Tool-support for PLC Software Analysis	36
4.2.	Code Analysis Procedures for the Quality Assessment of Industrial Software.....	38
4.3.	Research Gap in Quality Assessment of Control Software	43
5.	Procedure for Quality Assessment of Legacy Control Software with Static Code Analysis.....	45
5.1.	Quality Assessment Procedure for Legacy Control Software	45
5.1.1.	Pre-considerations Regarding the Quality Assessment of Control Software	45
5.1.2.	Overview of the Quality Assessment Procedure for IEC 61131-3-based Control Software in an Industrial Context	49
5.2.	Detailed Introduction to the Quality Assessment Procedure	51
5.2.1.	Familiarizing with Company-specific Boundary Conditions (Step 1)	51
5.2.2.	Static Code Analysis of a PLC Software Excerpt or Single Project (Step 2)	59
5.2.3.	Comparison and Results' Documentation of Additional PLC Software Parts or Projects Regarding the Selected Goal (Step 3).....	71
5.2.4.	Identification of Improvement Potentials, Including the Derivation of Recommendations for Action (Step 4).....	80

6.	Implementation	91
6.1.	Prototypical Implementation of Context-sensitive, Configurable Static Code Analysis Concepts.....	91
6.2.	Different Visualizations of Static Code Analysis Results	93
7.	Qualitative Evaluation.....	97
7.1.	Qualitative Evaluation with Case Studies.....	98
7.1.1.	Industrial Case Study B: Variability Analysis in the Intralogistics Sector	102
7.1.2.	Industrial Case Study C: Analysis of Version History and Estimation of Reuse Potential in the Automotive Sector (Component Assembly)	113
7.1.3.	Additional Insights from Industrial Case Study D.....	124
7.1.4.	Insights from Case Study E with a Lab-sized Demonstrator	125
7.1.5.	Summary of Insights Gained Through Case Studies	127
7.2.	Evaluation with Industrial Experts in Industry Working Group.....	132
7.2.1.	Evaluation of the Industrial Applicability of the Quality Assessment Procedure with Online Questionnaire	132
7.2.2.	Group Discussions on Challenges of Application in an Industrial Context.....	136
7.3.	Expert Workshop with an Industrial Focus Group in the Food and Beverage Sector	138
7.3.1.	Evaluation of the Applicability of the Quality Assessment Procedure.....	139
7.3.2.	Challenges Regarding the Applicability in an Industrial Context	141
8.	Assessment of the Fulfillment of the Requirements.....	147
9.	Summary and Outlook	151
10.	Literature.....	155
11.	List of Figures.....	177
12.	List of Tables	181
13.	List of Abbreviations	185
Appendix A.	Interview Guidelines and Checklist	187
Appendix A.1	Interview Guiding Questions and Project Selection (in Procedure Step 1)....	187
Appendix A.2	Analysis Checklist (Used in Procedure Step 2).....	191
Appendix B.	Industry-WG: Questionnaire and Results.....	193
Appendix B.1	Industry-WG Questions in German.....	193
Appendix B.2	Answers to the Online Questionnaire (translated to English)	195
Appendix C.	Industrial Expert Workshop – Questions and Results.....	201
Appendix C.1	Single-choice Questions in German	201
Appendix C.2	Answers During the Workshop (translated to English).....	202

1. Introduction

A short motivation for the quality assessment of control software for machines and plants is provided in the following. Subsequently, the structure of this thesis is introduced, followed by a delimitation to other research work in the research projects closely related to this thesis.

1.1. Motivation for Software Assessment with Static Code Analysis

Automated Production Systems (aPS) form a special, highly complex class of mechatronic systems, namely automated machines and plants, whose development generally comprises mechanics, electrics/electronics and software parts, all closely interwoven [Vog⁺15b]. The amount of aPS functionality implemented by control software is steadily increasing [DZ21; TB10; Vog⁺15a] and, thus, software quality is gaining importance. Moreover, Industry 4.0 poses additional requirements on the control software, such as evolvability, maintainability and changeability. In addition, the global competition entails a shorter time-to-market, which necessitates reusing available and mature implementation parts, including control software on different platforms.

For the planned reuse of software, modular design approaches have been identified as key enablers [Mey97]. They are supported by the three different types of control software units standardized in IEC 61131-3 [IEC61131-3]. Despite standardized programming languages defined in IEC 61131-3, the reuse of control software across different platforms is challenging since vendors provide differing implementations of this standard [Bau⁺04; SS16; Sta⁺14]. Moreover, control software is highly influenced and dependent on the controlled automation hardware [Can⁺21] and customer requirements, leading to additional software variants [Vog⁺15b]. As a consequence, the unplanned reuse strategy *copy, paste and modify*, where existing parts of or entire software projects are copied and modified to fulfill the requirements of the currently developed aPS, is still the most common form of reuse despite its various drawbacks [Fis⁺14; Fis⁺18]. This leads to a high amount of historically grown legacy control software, which usually does not follow modular structures and rarely follows strict coding guidelines [KP14].

According to a recent report, static code analysis is a preferred means to ensure software quality in general-purpose software programming [Ove20]. Moreover, it is used for understanding and documenting existing software [NNB19] and to reduce overall development costs. However, standard tools from computer science cannot be used in the domain of aPS without adaptations [Jet⁺13b] and the programming languages defined by IEC 61131-3 are supported by a few suppliers only [Prä⁺17]. Thus, despite many advantages, static code analysis is not yet commonly used

in aPS control software development. Consequently, the various stakeholders with different tasks involved in the control software development, e.g., developers for standardized and for project- or customer-specific software parts and their group leaders or project managers [Bou⁺19], so far have little experience with quality assessment utilizing static code analysis.

The main objective of this doctoral thesis is to bridge this gap by developing a procedure for the systematic, goal-oriented and context-aware quality assessment of control software by applying static code analysis and software metrics. Thereby, the documentation of analysis results for different stakeholders and the subsequent derivation of recommendations for action are targeted to enable the use of the gained insights. Consequently, the following research question is targeted:

How can the various stakeholders involved in control software development be supported to independently integrate static analysis means and concepts from computer science into their development workflow to assess control software quality and identify potentials for its improvement, e.g., increasing planned reuse of existing software solutions?

The contents and contributions of this dissertation are based on previous publications by the author, namely [Fis⁺21a; Fis⁺21b; Fis⁺21c; Fis⁺22b; FVF15; Neu⁺20c; Vog⁺22a]. Key aspects of the respective publications are given in the following:

- [FVF15] Analysis of the industrial control software of an automated warehouse to enhance planned reuse.

- [Neu⁺20c] Software modularity is a key enabler of planned reuse in control software and influences many other factors.

- [Fis⁺21a] Procedure to assess the modularity and reusability of control software units based on software metrics, which quantify the dependencies between software units in accordance with company-specific programming guidelines.

- [Fis⁺21b] Approach for measuring the overall complexity of control software units in different categories to identify outliers as a starting point for refactoring.

- [Fis⁺21c] Company-specific boundary conditions, software characteristics and usability of the reuse approaches need to be considered when choosing a reuse strategy.

- [Vog⁺22a] Highlighting challenges and solution approaches for the reuse of extra-functional control software parts with a focus on error handling.

[Fis+22b] Proposal for assessing the conformance of control software with company-specific programming guidelines using configurable rules. Highlights the importance of using suitable visualization means to illustrate the considered control software units within their context.

1.2. Structure of this Dissertation

This thesis is structured as follows: Chapter 2 (p. 5) provides an overview of the field of investigation and basic definitions. In Chapter 3 (p. 17), the domain requirements are presented, which were derived from the current state of the art and industrial practices. Based on these requirements, related works in the field of production automation and adjacent domains are investigated and rated for their applicability in Chapter 4 (p. 25). In this literature review, a research gap for the considered domain was identified. In Chapter 5, the procedure developed in this thesis for the identified requirements and research gap is presented. Insights gained in pre-studies, which are regarded by the developed procedure, are summarized in Section 5.1 (p. 45). The procedure consists of four main steps, which are presented in detail in Section 5.2 (p. 51). Chapter 6 (p. 91) introduces a prototypical implementation of selected aspects of the procedure. In Chapter 7, the evaluation of the procedure is presented. It was performed using industrial case studies (Section 7.1, p.98), an expert workshop within the scope of an industry working group meeting (Section 7.2, p. 127) and an expert workshop with a company from the food and beverage application sector (Section 7.3, p. 138). An assessment of the developed procedure regarding the derived requirements is presented in Chapter 8 (p. 147). The thesis concludes with Chapter 9 (p. 151), in which a summary of the achieved results and an outlook on future research are given.

1.3. Delimitation to Other Research Work in the Scope of Various Research Projects

The content of this thesis partially results from three research projects, which the author has worked on during her time at the Institute of Automation and Information Systems (AIS) at the Technical University of Munich (TUM).

Within the research project *Increased flexibility for heterogeneously structured material flow systems enabled by intelligent software agents controlling self-configuring conveyors* (iSiKon – 251665026) [GEP22b], which was funded by the German Research Foundation (DFG), the aim was to achieve flexible, reconfigurable automated material flow systems with reusable high-quality control software. The project was a cooperation between the institutes AIS and Materials Handling, Material Flow, Logistics (fml) at TUM from 2015 to 2018. After defining a joint system

architecture, Christian Lieberoth-Leden from fml targeted the adaptation to material flow requirements, transport planning and route calculation and optimization from an intralogistics point of view [Lie22]. Since 2017, the author of this thesis has worked on conceptualizing and implementing software agents to control the individual material flow modules focusing on reuse, uniform software module interfaces and functionality encapsulation in the control software [Fis⁺20c].

In the DFG-funded project *Reverse Engineering Design of Software Product Lines for Automation Technology* (RED SPLAT – 335427442) [GEP22a], introducing a concept for variant management and planned reuse of historically grown legacy control software was targeted. The project was a cooperation between the AIS at TUM and the Institute of Software Engineering and Automotive Informatics (ISF) at TU Braunschweig from 2017 to 2020. Alexander Schlie and Kamil Rosiak from ISF developed and implemented a concept for defining and automatically calculating similarity metrics to identify control software variants and store them as family models [Ros⁺21a]. The author of this thesis manually analyzed several industry-sized control software variants and documented them in models to derive prerequisites and requirements for their tool-based identification. This included requirements for a domain-specific representation of the identified variability in a software product line from the perspective of the machine and plant manufacturers. Moreover, the author of this thesis conducted expert interviews to gain an insight into the current state of practice regarding variant management and reuse [Fis⁺18]. In addition, Safa Bougouffa contributed at AIS to the RED SPLAT project by preparing representative lab-sized control software examples, including examples of domain knowledge and metadata, and developing a concept for visualizing the identified variability, which was subsequently enhanced by the author of this thesis. In the scope of evaluating the quality assessment procedure presented in this thesis, results from RED SPLAT were used in *Case Study E* highlighting the identification and subsequent planned reuse of control software variants as an essential aspect of improving software quality.

Moreover, the static code analysis of control software to support its revision and quality improvement using refactoring was targeted in the research project *Advanced systems engineering for control software as a prerequisite for flexible, adaptive cyber-physical production systems* (advacode – DIK0112/04) conducted at AIS in cooperation with four German companies. The project was funded by *Bayerisches Staatsministerium für Wirtschaft, Landesentwicklung und Energie* and *Zentrum Digitalisierung Bayern (ZD.B)* from 2020 to 2022. During the project, the focus of the author of this thesis and her colleague Eva-Maria Neumann was on the consideration of company-specific boundary conditions during the static analysis. More precisely, Eva-Maria Neumann targeted the identification of code clones and a function-oriented reuse approach for control software. The author of this thesis worked on conformance checks of company-specific data exchange rules, taking into account the implemented functionality and complexity of the various software parts.

2. Field of Investigation

The presented procedure aims to support the quality assessment of control software with guiding questions, checklists and static code analysis. It is intended for industrial application in the domain of aPS. To better understand the domain requirements and boundary conditions, the theoretical background of the field of investigation, i.e., industrial automation, aPS and control software development, is presented below. Moreover, available reuse strategies for control software and a short introduction to static code analysis are provided.

2.1. Industrial Automation

Industrial automation aims to automate *technical processes*, i.e., processes that manipulate the state of material, energy or information (cf. [LG99], p. 1). Technical processes consist of different *procedures* (cf. [LG99], p. 43-47), which can be classified as *continuous*, *event-discrete* and *object-related*. Continuous procedures such as deformation procedures in hydraulic presses contain time-related, continuous process values and can be described with linear differential equations. Event-discrete procedures consist of sequentially occurring, distinguishable (discrete) process states such as different steps in a manufacturing process. They can be described with Petri nets or flow charts. In object-related procedures, individual objects, e.g., produced parts, are transformed, transported or stored, which can be modeled with simulations, state machines or Petri nets. Although different procedures are mixed in a technical process, in most cases, a dominant procedure type can be identified, which determines the type of the technical process, i.e., continuous, sequential (discrete) or batch process (cf. [LG99], p. 47).

Technical processes are executed in *technical systems* and can be automated with process automation systems (cf. Figure 1 and [LG99], p. 6). A process automation system consists of three closely coupled subsystems, i.e., a controlled technical system (bottom) executing a technical process, a computing and communication system (middle), and operating personnel (top). For interaction with the technical process, sensors and actuators are utilized: sensors, which measure physical quantities and convert them to electrical or optical signals, are used to observe the technical system, while actuators influence the physical quantities of the technical process to control the technical system (cf. [LG99], p. 29). The sensor and actuator signals are exchanged between the technical system and the supervisory computing and communication system, which observes and controls the technical process. Human operators usually interact with the computing and communication system via a *human-machine interface* (HMI) to supervise and influence the process or react in case of an error.

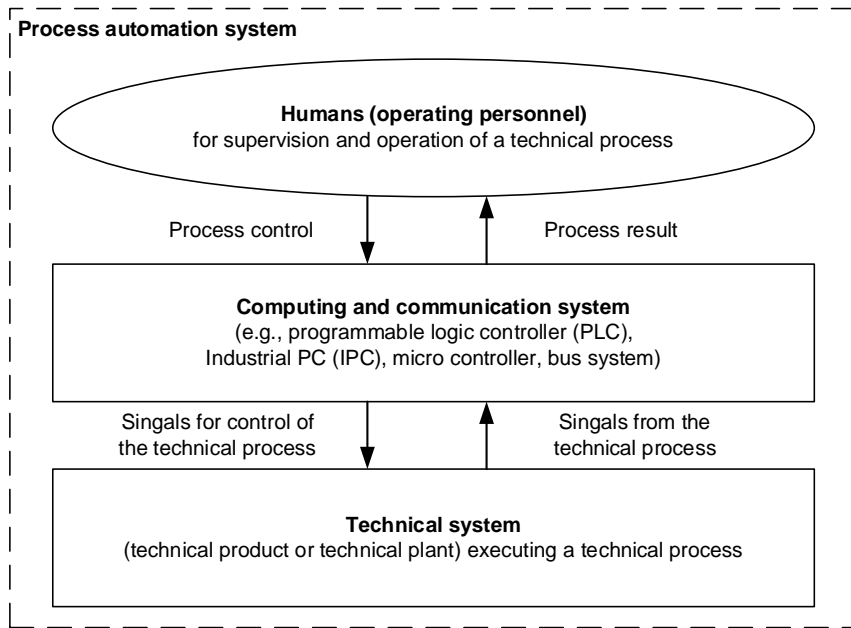


Figure 1: Schematic structure of process automation systems (translated from [LG99], p. 7).

aPS are process automation systems that control production processes. Moreover, aPS form a special class of mechatronic systems [BF03], i.e., automated machines and plants, whose development involves mechanics, electrics/electronics and software, all closely interwoven [Vog⁺15b]. As mechatronic systems, aPS have “the task of converting, transporting, and/or storing energy, material, and information flows with the help of sensors, information processing, and actuators” [VDI2206].

In aPS, typically, a programmable logic controller (PLC) or an industrial PC (IPC) is used as a computing and communication system executing the control software. PLCs consist of a central processing unit (CPU), power supply, memory, communication and circuit modules (input/output (I/O) ports) to receive data from sensors and transmit data to actuators (cf. [Han15], p. 8). The connection to plant peripherals, e.g., sensors and actuators, can be implemented via direct wiring to the I/O ports or remote I/O modules, which communicate with the PLC via fieldbuses [Vya13]. The control software is loaded on the PLC memory and executed by the CPU, accessing I/O ports. Typically, PLCs are characterized by their cyclic processing of the control software, with a cycle being divided into four steps [Vog⁺15b]. At the beginning of each cycle, the PLC reads the input values, i.e., sensor signals, from the technical process and stores them in a process image. Based on the stored values, the PLC program is executed. Next, the output values, i.e., the control signals for the actuators manipulating the technical process, are written. Finally, the PLC waits until the predefined cycle time has elapsed. By executing tasks within designated cycle times, PLCs adhere to real-time requirements, meaning that the set cycle times may never be exceeded to ensure process stability, safety and security.

2.2. Background on the Domain of Automated Production Systems

This Section introduces the characteristics of aPS with a focus on control software development, including platforms, programming languages, and boundary conditions to better understand the domain's constraints and requirements.

As aPS are mechatronic systems, their development process includes different disciplines, e.g., mechanics, electrics/electronics and software. The discipline's collaboration during the development still poses a major challenge [Fol⁺11]. Development processes for mechatronic products like aPS are often sequential and range from established ones such as the V-model [VDI2206] to more recent methods like Scrum [Scrum22]. In size, aPS range from machines to entire production plants, with the input and output signals to and from the peripherals differing from some hundred to up to 10.000 [Fis⁺18]. Moreover, aPS are often in use for decades, which leads to a high amount of legacy software that has been programmed over several years by multiple software developers and, consequently, rarely follows strict coding guidelines [KP14]. The legacy software is differentiated into sequentially developed versions, i.e., “an initial release or re-release of a computer software configuration item”, and in parallel existing variants, i.e., “a version of a program resulting from the application of software diversity” [IEEE610].

In recent years the degree of automation and, thus, the proportion of software in production automation has been growing [DZ21] and, consequently, control software implements an increasing proportion of aPS functionality [Thr10]. This rising importance of control software requires approaches to cope with the challenges and specifics of aPS to enable a quick time-to-market with well-tested, high-quality control software. Additionally, control software needs to fulfill specific requirements regarding real-time and reliability [Vog⁺15b]. Typically, aPS are designed-to-order systems with high complexity and variations resulting from customer-specific requirements and the degree of on-site changes, increasing from machine manufacturing over special-purpose machinery to plant manufacturing [Vog⁺15b].

Usually, aPS control software is executed on PLCs, which are commonly programmed following the IEC 61131-3 standard [IEC61131-3]. It defines three so-called *Program Organization Units* (POUs) to enable the encapsulation of PLC software into reusable units, namely *Programs* (PRGs), *Function Blocks* (FBs) and *Functions* (FCs). The main differences between these POUs are that, in contrast to FCs, PRGs and FBs possess internal memory and FBs need to be instantiated before their use. Tasks are used to define entry points, i.e., PRGs, into an aPS's control software, which are invoked depending on the task's specified cycle time or priority. The entry points then call other POUs, which can execute code and sub-calls of further POUs.

Each POU consists of a declaration part to define its variables and an implementation part that contains the control code programmed in one of the five IEC 61131-3 programming languages. The standard defines two textual programming languages, i.e., Structured Text (ST) and Instruction List (IL), and three graphical programming languages, i.e., Ladder Diagram (LD), Function Block Diagram (FBD), and Sequential Function Chart (SFC). These programming languages are usually mixed within a PLC project since they have different strengths [VFB03]. For example, ST is similar to the high-level programming languages Pascal or C [Han15] and used for mathematical equations, implementing controllers or technological functions. SFC is suitable for control sequences, while LD (derived from physical relays in circuit diagrams [LÅ07]) is proper for interlock conditions and the assembler-like IL for time-critical functionalities. Consequently, the same functionality can be implemented in different languages, leading to additional variability [Fis⁺18]. The implementation part of POUs programmed in IL, LD and FBD is divided into networks. For structuring the data exchanged between POUs, apart from elementary variable data types, user-defined data types (UDTs) can be defined, e.g., to store all data belonging to a manufactured product in one data type. Moreover, variables can be defined in global variable lists (GVL).

The recent version of the IEC 61131-3 standard introduces three new language elements (methods, inheritance and interface abstraction) as object-oriented (OO) extensions to enhance modularity and reuse of control software. Various authors highlighted the benefits of object-oriented programming (OOP) [BFS13; Vog⁺22a; Wer09] and tool support for the OO extension of IEC 61131-3 has been available for a decade [Wer09]. Moreover, first guidelines have been developed, such as [PLC21]. However, according to a recent survey, with only 10% of the industrial participants using OO IEC 61131-3 by default and 42% not using it at all for their control software development, it is not yet the state of practice in industry [VO18]. Apart from the IEC 61131-3 and PLCs, also more sophisticated control platforms and high-level programming languages are applied in the industry. Furthermore, new approaches aim at low-cost, open-source solutions for designing aPS (cf. [KS19] for an overview). However, these are not targeted in the scope of this thesis.

As the market penetration of different PLC platform suppliers differs, globally operating aPS manufacturers regularly need to deliver their systems equipped with different PLC types, depending on the market they are targeting [Fis⁺18]. Despite the standardized exchange format PLCopen XML [IEC61131-10], which allows exchanging configuration elements, data types and POUs between different IEC 61131-3 development environments, reuse of PLC software across different platforms is still challenging. This results from inconsistencies in the standard leading to different implementations of the standard, which reduce portability [SS16]. Additionally, some vendors like Siemens use slightly different constructs, e.g., Organization Blocks (OBs) similar in function to

PRGs, but state their compliance to IEC 61131-3 [Sie15]. In Siemens PLCs, variables can be organized in Data Blocks (DBs), which are either global (similar to a GVL) or belong to an FB.

Functionality-wise, control software for aPS consists of two main parts, i.e., functional and extra-functional implementation parts [Vog⁺22a]. Functional implementations include the control of actuators and the processing logic for implementing the aPS behavior in automatic mode. In contrast, extra-functional software implements communication tasks, diagnosis, fault handling and operating modes, which are highly interconnected with functional parts. For example, the extra-functional task error handling requires close connections to all functional parts controlling actuators to trigger an actuator's emergency stop if needed, which leads to fundamentally different code structures [Vog⁺22a]. Extra-functional code makes up around 50-75% of industrial control code [LT03], causing complexity due to strong dependencies with functional code parts [Neu⁺20c]. Thus, suitable approaches for reusing the differently structured extra-functional parts are required [Vog⁺22a]. An overview of typical functionalities in aPS software is provided in [Wil⁺22].

According to [Mey97; Vog⁺15b], defining a suitable *software architecture* is essential to ensure high software quality and enable extendibility and reusability, as the architecture design directly affects the software's quality attributes [Lyt⁺20]. The standard ISO 42010 defines architecture as the “fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution” [ISO42010]. Various definitions from computer science are in line with this definition, e.g., Reussner et al. [Reu⁺19] or Medvidovic and Taylor [MT00]. However, these definitions are not applicable to control software since they lack core characteristics of aPS, such as real-time requirements or the dependency on automation hardware [Neu⁺22]. Therefore, in the domain of aPS, the following definition was derived:

*Software architecture “is composed by the basic structure of the software determined by **design decisions** forming the skeleton of the software. These design decisions primarily determine modules' size, interfaces, and interaction, i.e., groups of POUs jointly performing a specific functionality. Automation software architecture in CPPS [cyber-physical production systems] is required to consider the **hardware topology**, including the number, distribution, and connection of PLCs. Thus, architecture is dependent on the interface to and the number and type of **hardware I/Os**, i.e., the sensors and actuators for controlling the mechatronic system's behavior.” [Vog⁺22b]*

In the scope of this thesis, in addition to the definition by [Vog⁺22b], the implementation of extra-functional tasks and information on the controlled production process, including its boundary conditions, are considered essential to comprehend control software and assess its quality.

Vyatkin [Vya11] introduces a software architecture for distributed automation systems based on the IEC 61499 standard [IEC61499], which defines event-driven FBs. Significant benefits of this approach are reduced time and effort to develop automation software, a high degree of code modularity and a high potential for reuse. These benefits are already proven through first industrial applications. However, the standard is not commonly used in the industry yet and “[...] has [still] a long way in order to be seriously considered by the industry” [Thr13]. Furthermore, the programming model proposed by IEC 61499 proved to be nondeterministic [CLÄ06] and is rated as “too complicated to gain wide adoption in industry” [Seh+21].

For the development of high-quality PLC software, some application sector-specific guidelines are available. For example, ISA 88 for batch processes [IEC61512; ISA88], PackML, including the OMAC state machine, in the food and beverage sector [PackML] or SAIL for intralogistics systems [VDI5100]. Moreover, some application sectors need to consider specific laws and standards, e.g., quality management of certified software in medical applications [ISO13485]. In addition, general recommendations and guidelines have been defined, e.g., by the PLCopen [PLC22], a vendor- and product-independent consortium that works on standardization and harmonization in industrial automation, e.g., concerning aspects such as motion control or safety. Moreover, a technical report describes guidelines for applying and implementing the IEC 61131-3 programming languages and their debugging [IEC61131-8]. Still, most companies define their own internal programming guidelines tailored to their needs and applied reuse strategies.

2.3. Reuse Strategies for Control Software

This section shortly introduces selected means from academia and industry to support the reuse of PLC software while coping with aPS-specific challenges and constraints, including approaches adopted from the computer science domain.

Generally, ISO/IEC 25010 defines reusability as the “degree to which an asset can be used in more than one system” [ISO25010]. Moreover, modularity is considered a prerequisite for planned software reuse [Mey97]. It is defined as the “degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components” [ISO25010]. According to the mechatronic development standard, a module consists of components (sensors and actuators) and fulfills a specific process or task [VDI2206-04]. With a focus on control software, “conceptually, modules represent a separation of concerns and improve maintainability by enforcing logical boundaries between components.” [ISO25023]. In the scope of this thesis, a module is considered “a software unit that can be reused without any modifications in different software systems” [Vya13]. Depending on the applied programming style, a

module can either be a single POU or a group of POUs. Modules should be managed according to their level of granularity [MJG11a]. More precisely, their reuse potential increases with higher granularity, but it also leads to a higher organizational effort in managing the modules [MJG11b].

The control software of aPS is usually modularized in a hardware-oriented way, i.e., following the physical machine layout [Neu⁺22]. A function-oriented modularization principle for intralogistics systems is proposed in [Hom⁺11]. Also, combining both principles is common: on the top level, the control software is modularized following a hardware-oriented principle and the resulting modules are designed function-oriented [Fis⁺21a]. Schröck et al. propose to divide the software development process into project-dependent and -independent tasks [SFJ15]. This separation enables the design of software modules independently from a specific machine or order, facilitating their reuse in different contexts without requiring extensive changes [Fis⁺21a]. Consequently, different stakeholders are involved in the development of control software, e.g., managers, application developers for customized machine- or order-specific tasks and module developers for the development of reusable, standardized modules [Bou⁺19]. Moreover, control software needs to be understood by operators for troubleshooting [LÅ07]. By analyzing the control software of different industrial companies, Vogel-Heuser et al. [Vog⁺15a] identified five architectural levels in PLC software, which can be mapped to the levels defined by ISA 88 [IEC61512; ISA88]. Generally, the reuse potential of application-specific POUs correlates with the architectural hierarchy levels they are located on: the higher the POU level, the lower usually its potential for reuse [Fis⁺18].

Generally, reuse strategies are differentiated into planned and unplanned forms [Mah14]. To stay globally competitive and enable a short time-to-market, especially means for planned reuse are essential [Fis⁺21c]. Apart from standardized modules organized in libraries, templates and design patterns, also the use of models has been promoted as a solution to cope with the rising complexity of software development [Vya13]. Various model-based approaches with different aims (e.g., generation of interlocking conditions or HMI software) and based on different inputs (e.g., design models from specific application sectors like the process industry) exist in the aPS domain. In a recent survey comparing 13 code generation approaches in the aPS domain, the merging of generated and manually developed code was identified as a point for future research [Koz⁺20]. Moreover, variant and version management was identified as a prerequisite for planned reuse (cf. [Fis⁺14; Rab⁺18; Vog⁺15b]). However, identifying suitable, reusable software parts is a challenging and time-consuming task, which requires detailed knowledge of existing variants [Fis⁺15]. Additionally, approaches from the computer science domain cannot be applied to control software from the aPS domain without changes due to its boundary conditions [Vog⁺15b]. A short overview of available, planned reuse strategies is presented in Table 1.

Table 1: Selected reuse approaches for control software in the aPS domain.

Reuse Strategy	Details on available approaches
Library modules/ library POU's	<ul style="list-style-type: none"> • Company-specific, standardized software modules (often developed in a dedicated department; distinguished in company-wide or machine-type-specific libraries) [Fis⁺18] • Vendor-specific library modules to ease controlling a vendor's hardware • Activities such as <i>Open Source Community for Automation Technology</i> [OSCAT]: development of cross-vendor, open-source libraries to support programming of recurring functionalities in PLC projects, which market-leading companies use • Parameterizable <i>universal modules</i> as a particular form to represent different variants (risk of a high amount of dead code) [Vog⁺17] • Functionalities of a well-defined, reusable POU for hardware control [GWF08] • Encryption of library modules to prevent modification by library users [Sta⁺14]
Templates	<ul style="list-style-type: none"> • Pre-developed formats for individual POU's or entire software projects [Fis⁺21a] • Useable in combination with code generation approaches [PKK12; PKS18]
Design Patterns	<p>Design patterns describe abstract, proven solutions to frequently recurring problems [Wu⁺20]</p> <ul style="list-style-type: none"> • <i>The Gang of Four</i> defined 23 design patterns in high-level OO programming [Gam⁺95] • Design patterns for translating UML models into control software [FSB11], for PackML-compliant PLC software, including operating modes [BBF15], and for mechatronic systems using OOP in packaging machines [BFS13] • Structural design patterns in industrial PLC software [Fuc⁺14], formalized in [Neu⁺20b] • Two Patterns for implementing the extra-functional task error handling [Vog⁺22a]
Model-driven approaches and code generation (often in combination with templates)	<p>Overview and classification of control code generation approaches with criteria derived from industrial best practices, including inputs and generated outputs provided by [Koz⁺20]</p> <ul style="list-style-type: none"> • PLC Coder Matlab Toolbox (generation of LD and ST control logic; supported platforms CODESYS, Rockwell, Siemens TIA, OMRON) [Math⁺22] • Code generation from application sector-specific models, e.g., <ul style="list-style-type: none"> ◦ generating interlocking conditions in the process industry by using documents from the development process [DFS06] ◦ based on the module-type package (MTP) in process plants, with operating modes (via PackML state machines) and HMI functionality [Lad⁺18] ◦ from the layout plan in the intralogistics domain [AFV22] ◦ with models used in the power generation industry [SC21] ◦ in process industry with piping and instrumentation diagrams (P&ID) [KKV18] • PLCopen-compliant software generation, e.g., <ul style="list-style-type: none"> ◦ from UML models (with mapping between profiles for control, electrical/electronic and software engineering views) [Pri⁺16] ◦ from SysML and feature models (generation of OO-IEC 61131-3) [PS13] ◦ with methodologies combining GEMMA (operation modes) and GRAFCET (behavior description in sequential processes) [Alv⁺12; BG21; Cas⁺21] ◦ with a formal model of GRAFCET for SFC [SF14] and ST code [Jul⁺17] ◦ a component-based IEC 61131-3 model using a markup language [EMO07] • Code generation, including HMI functionality [HVA16] • Template-based code generation from a plant model for TIA Portal [PKS18] • Modular tool suite for code generation (XML-based; TIA Portal) [Arm⁺18] • Model-driven development of PLC software for machine tools [ZP08] • Ontology-based automatic code generation [SZ12], for different vendors [An⁺21]
Variant and version management (adopted from computer science domain)	<p>Feature-oriented development paradigm [Cza98] and software product lines (SPLs) are popular approaches to enhance the planned reuse of variant-rich software in computer science [PBv05].</p> <ul style="list-style-type: none"> • Reverse engineering approaches to document variability and enable planned reuse of legacy software, e.g., [Fis⁺14; Hin⁺18; Ros⁺21a; Sch⁺19] • Supporting an enhanced application of <i>copy, paste and modify</i> with ECCO [Fis⁺15] • Interdisciplinary SPLs to represent the mechatronic characteristics of aPS, e.g., [Fad⁺22; Fan⁺15; FV17; SFJ15; VSF16] • Interdisciplinary version management [Bif⁺15] based on AutomationML, enlarged with cardinality-based variability modeling by [Wim⁺17]

Overall, to reach a suitable base for planned reuse, a combination of different reuse strategies is required [Mah14]. This assumption is confirmed by a previous study in the aPS domain, which identified that market-leading companies pursue mixed forms of reuse [VON18], e.g., by combining templates and library modules. Further, [Neu⁺20c] emphasizes the impact of factors such as the development team size on applicable reuse strategies in software development and their potential benefits. Moreover, to ensure the applicability of the selected reuse strategies, a suitable software architecture is required, including documentation such as guidelines or checklists, e.g., for integrating standardized library modules into the application-specific control software project. Although several approaches for planned reuse are available, the most common reuse strategy applied is still the ad-hoc and unplanned method *copy, paste and modify* [Fis⁺14; Fis⁺18; Vog⁺17], despite its many drawbacks such as being error-prone and time-consuming [Fis⁺14].

In summary, various reuse strategies are known (cf. Table 1), but not all recent approaches from academia are adopted in an industrial context or are yet adaptable to aPS boundary conditions. Moreover, there are neither publicly available lists of coding best practices for IEC 61131-3 software [Jet⁺13b] nor guidelines to support software developers in choosing a suitable reuse strategy for developing high-quality software in greenfield projects. Instead, many legacy control software projects developed using *copy, paste and modify* are available. Respectively, the procedure proposed in this thesis aims to assess existing brownfield legacy control software regarding a defined analysis goal. For deciding on measures to improve existing, analyzed software to overcome identified drawbacks, it is advantageous if the involved software developers are aware of basic, available reuse strategies, e.g., modularization, libraries, templates, code generation and variant management.

2.4. Static Code Analysis and Software Metrics

During static code analysis, software programs are examined without executing them [EN08], which enables detecting potential errors and problematic code constructs in early development phases [Prä⁺17]. The structure of a program and its elements are examined, including their dependencies, e.g., with visualizations such as call graphs [Ryd79], to approximate the program behavior [Prä⁺12]. More precisely, static code analysis checks the code for errors that the compiler cannot detect during the syntactical assessment of the source code, e.g., violations of naming conventions or possible performance problems [Prä⁺12], as well as the source code's conformance to company- or domain-specific programming guidelines [DZ21] and safety coding rules [Jet⁺13b]. As a result, static code analysis points out "potentially weak spots to the developer" [NNB19] that could affect the quality of the analyzed software. However, recent research has identified a gap

between the “academically perceived potential of static analysis and its use in practice”, which results from usability issues of analysis tools, especially targeting explainability [NNB19].

The application of static code analysis tools is a common approach in computer science for understanding and documenting existing software and for developing high-quality software by detecting potential bugs or weaknesses [NNB19]. The first static analysis tool was Lint [Joh77] for the programming language C to assess programs that compile without errors for undetected bugs [Lou06]. However, so far, the PLC programming languages defined by IEC 61131-3 are supported by a few suppliers only [Prä⁺17].

Software quality is specified, measured and evaluated using quality properties defined in standards, e.g., SQuaRE, which is a series of international standards that defines software quality models [ISO25010] and means for measuring them [ISO25023]. The IEC/ISO 25010 defines characteristics (e.g., *maintainability*), which are further subdivided into sub-characteristics (e.g., *modularity* and *reusability*), to categorize software quality [ISO25010]. These quality characteristics often pose conflicting requirements on the software, which results in trade-offs during the software architecture design [Lyt⁺20]. Software metrics are applied to provide objective, quantifiable results for comparing software parts according to quality characteristics [SZ20], which is achieved by counting code properties and aggregating them according to a fixed algorithm [Wey88]. The IEEE standard 1061 defines a software metric as

“a function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality” [IEEE1061].

While some sub-characteristics are directly measurable, others require the definition of quantifiable software quality properties [ISO25010]. For example, software complexity metrics “do not measure the complexity itself, but instead measure the degree to which those characteristics thought to lead to complexity exist within the code” [LC94]. Consequently, defining appropriate thresholds and understanding and interpreting the metric results can pose a challenging task [VFN20]. Generally, software metrics developed for general-purpose programming languages (GPL) in computer science cannot be directly adopted to IEC 61131-3 control software [Jet⁺13b].

Static code analysis and software metrics can support the generation or enhancement of documentation about the control software. At the same time, documentation from different disciplines enables the interpretation and understanding of the software parts in their mechatronic context. In the scope of this thesis, the following definition for documentation is used:

Documentation is any material related to an aPS that contains recorded information used or created during the aPS life cycle in any involved discipline. Examples are functional descriptions, mechanical layout plans, circuit diagrams, programming guidelines, source code or gained and documented code analysis results such as call graphs or identified design decisions. Thereby documentation is available in different formats, e.g., graphical like layout plans or textual such as programming guidelines, and can be provided in digital or analog form (on paper).

Understanding software is currently still done by reading the source code [GC15], since available documentation is often outdated [Kir⁺16]. Thus, generating up-to-date documentation during static code analysis can improve the comprehension of legacy software [Kir⁺16]. In addition, it is recommended to use static code analysis as part of code review processes [CW07]. The IEEE standard 1028 distinguishes five types of software reviews and audits, i.e., management reviews, technical reviews, inspections, walk-throughs, and audits [IEEE1028].

3. Requirements for a Software Quality Assessment Procedure with Static Code Analysis

A quality assessment procedure for legacy control software using static code analysis must fulfill different requirements. These requirements can be derived from the field of investigation, current challenges in the industry and selected industrial and academic case studies (cf. Table 2). In the following, the derived requirements are introduced in four categories, targeting the applicability in the aPS domain, adaptations to company-specific constraints, conducting the static code analysis and, finally, using the gained insights to assess and improve the software quality.

Table 2: Overview of conducted industrial and lab-sized demonstrator case studies.

	Industrial Case Study A	Industrial Case Study B	Industrial Case Study C	Industrial Case Study D	Lab-Sized Case Study E
Product (company type)	PM	PM	MM (SPM)	PM (with SPM)	Lab-sized demonstrator
Application sector	Woodworking	Intralogistics	Automotive engineering (supplier), pharma / medicine	Automotive engineering (supplier), pharma / medicine	Intralogistics / factory automation
PLC development environments	Siemens STEP 7; Rockwell	Siemens STEP 7	Siemens TIA; Beckhoff; Rockwell	Siemens TIA; Beckhoff	Beckhoff
Used programming languages	LD, IL	IL, FBD	S7 Graph, LD, SCL	LD, FBD; (S7 Graph, IL and SCL on request)	SFC, ST
Applied reuse strategies (at case study time)	Copy, paste and modify	Copy, paste and modify; some standardized POU's	Library modules, template project	Library modules, template project, code generation/configuration	OMAC state machine on ISA 88 hierarchy levels

Legend: MM: machine manufacturing; PM: plant manufacturing; SPM: special purpose machines; Siemens STEP 7: Siemens SIMATIC STEP 7; Siemens TIA: Siemens Totally Integrated Automation Portal.

The first category contains requirements that the concept for quality assessment must fulfill to be applicable in the domain of aPS.

3.1. Applicability in the Domain of Automated Production Systems

In the aPS domain, there are various PLC suppliers and, consequently, different platforms are currently used in the industry (cf. Table 2). PLCs from different suppliers pose different boundary conditions for developing control software, e.g., IEC 61131-3-based PLCs enable structuring the

control software with PRGs, FBs and FCs and store variables in GVLs. In contrast, Siemens PLC software is structured using OBs, FBs and FCs and variables stored in DBs. Additionally, IEC 61131-3 based PLCs support the standard's OO extension. PLC suppliers offer different library modules for reusable functionalities such as timers or the control of vendor-specific automation hardware like drives. Due to these differences and since PLC suppliers provide different implementations of IEC 61131-3 [Sta⁺14], control software development strategies and software architectures differ. Accordingly, the assessment procedure should be independent of these platforms and their boundary conditions, focusing on IEC 61131-3 based platforms and Siemens PLCs.

Requirement R_{PLC} – Platform Independence

The procedure should be applicable to control software from different platform suppliers.

Manufactured aPS vary in size and complexity, ranging from serial machines to entire production plants (cf. Table 2). Special purpose machines and plants are usually highly customized. Due to their physical size, commissioning of the whole aPS is often performed on-site at the customer's premises, resulting in software changes under time pressure [Vog⁺15b]. In contrast, serial machines are configured rather than custom-built, which leads to different software architectures and reuse strategies. Since the analysis concept should be universal, it needs to be applicable to the control software of different manufactured aPS, which are considered as products, ranging from serial machines to plants.

Requirement R_{Pro} – Manufactured aPS as Product

Independent of the manufactured product, i.e., machines and plants, the procedure should be applicable to the product's control software.

To ensure the industrial applicability of the quality assessment procedure by software developers, it needs to be applicable to company-specific boundary conditions targeted in the following.

3.2. Procedure Application by Software Developers including Adaptations to Company-specific Boundary Conditions

aPS are long-living systems and, consequently, their control software needs to evolve and adapt to changing requirements and new customer demands. Therefore, the control software assessment regarding aspects such as modularity, reuse and variability is a continuous process, which has to be repeated. A single, one-time analysis of the control software through static code analysis is insufficient to sustainably improve the control software quality and simultaneously shorten the

development time. Thus, after an introduction or training, the procedure needs to be applicable by the software developers within the company to avoid dependence on an external service provider.

Requirement R_{Use} – Application by Software Developers (Users)

After an introduction, an application or module software developer of the company must be able to apply the procedure independently (no dependence on external analysis experts).

Depending on the application sectors, aPS manufacturers must comply with different guidelines, laws and regulations [Nai⁺15]. These application sector-specific guidelines, e.g., PackML [PackML] in the food and beverage sector or SAIL [VDI5100] in the intralogistics sector, influence the control software architecture. Further, laws and legal guidelines, e.g., the Good Automated Manufacturing Practice (GAMP) [GMP; ISO13485] for aPS in the medical or pharmaceutical sector, impact the control software design and functionality distribution. Since the concept for quality assessment should be universally applicable, it must be independent of the application sector but support the consideration of its respective guidelines, standards and constraints.

Requirement R_{Sec} – Application Sector

The concept should be applicable regardless of the application sector while taking into account application sector-specific requirements, e.g., regulations, standards and constraints.

Static code analysis is not meant to be a rigid standard procedure following fixed steps. Instead, it should be adaptable to the circumstances and challenges in a company's software development. Depending on different factors, e.g., a company's current software development process, the involved personnel and targeted reuse approaches, different aspects should be focused on in the analysis to identify improvement potentials. Moreover, known challenges during the development or recurring issues during start-up usually provide hints regarding sub-optimal implementation parts or detrimental design decisions in the control software. Thus, the developed assessment procedure should support identifying a company's pain points with respect to the control software.

Requirement R_{PP} – Pain Points

The procedure should identify a company's pain points in the software development to derive the analysis goal and consider the pain points during the analysis.

An essential factor for the successful application of the assessment procedure in the industry is its integration into the development workflow of the respective companies [NNB19]. Depending on the organization of the software development process, the standards to be taken into account and the applied reuse strategies (cf. Table 2), different stakeholders are involved and the development workflows differ. Also, depending on the analysis goal, the quality assessment procedure needs to

be applied at different stages of software development. For example, assessing a software's conformance to programming guidelines should be performed in parallel to the development process, while assessing a software unit's reuse potential is only feasible after its development has been completed. Thus, the procedure needs to be integrateable into different company development workflows and, depending on the analysis goal, at different development steps in these workflows.

Requirement R_{Work} – Workflow Integration

The procedure should be able to be integrated into the workflow at different steps, depending on the selected analysis goal.

Depending on the development team size and organizational aspects of the software development workflow, design decisions regarding applied reuse strategies and the overall software architecture are taken during software development [Neu⁺20c]. Moreover, considering software in its coding context is essential since missing knowledge about the software's context may lead to wrong or misleading analysis results [NNB19]. Thus, the assessment of the control software requires taking deliberate design decisions and the coding context into account to gain valuable insights into the software quality during the static code analysis.

Requirement R_{DD} – Design Decisions

Deliberate design decisions and the coding context must be considered in the analysis. Therefore, the quality assessment procedure must be adaptable since they usually depend on the boundary conditions of the company, e.g., organizational aspects such as the development process or the experience of the software developers and special customer requests.

Apart from the applicability in the aPS domain and in an industrial context, also requirements targeting the static code analysis itself have been derived and are presented in the following.

3.3. Conducting Static Code Analysis for Quality Assessment

Applying static code analysis without a specific goal prevents choosing suitable measures such as metrics, call graphs and visualizations. Depending on the faced challenges, i.e., pain points, during the software development, different analysis goals and measures need to be considered. During control software development, the unplanned reuse strategy *copy, paste, and modify* is still predominant [Fis⁺14], resulting in many duplicated code parts. Identifying these duplicates, i.e., code clones, for their subsequent planned reuse is a potential analysis goal. In more mature companies applying template-based software development, the conformance to programming guidelines and correct use of templates is essential. Thus, tracking modifications of the template and checking their conformance to guidelines during the development process would be a suitable analysis goal.

Requirement R_{Goal} – Analysis Goal

To be universally applicable, the procedure needs to be suitable for different analysis goals, e.g., reducing code clones, increasing planned reuse, restructuring, variant management, tracing changes during the development process and conformance to company guidelines.

It needs to be ensured that the developed procedure can be applied to industrial control software to support its assessment and identify improvement potentials. Therefore, the procedure needs to cope with the size of industrial control software (up to 270 or more POUs [Vog⁺15a], some of which, implemented in ST, have up to 1500 Source Lines of Code [Fis⁺21b]).

Requirement R_{Scal} – Scalability

The developed procedure needs to be applicable to industry-sized control software.

Moreover, the effort to apply the procedure should not increase with repeated use to enable its application in an industrial context. Instead, it should decrease due to familiarization with the procedure and the available analysis means, allowing an efficient software quality assessment.

Requirement R_{Eff} – Application Effort

The procedure's application effort should decrease compared to the initial application when applied several times to control software projects of comparable size, programming style and aPS type.

The long life cycles of aPS combined with customer-specific requirements lead to a high amount of legacy software that has been programmed over years and rarely follows strict coding guidelines [KP14]. However, understanding the legacy software is a prerequisite for performing time-consuming, error-prone development tasks [Ste00]. Studies illustrate that, on average, software developers spend as much time understanding existing code as they do writing new code [LVD06] or even 70% of their time understanding code [MML15]. Moreover, “understanding the rationale behind existing code” is perceived as a big problem [LVD06] and can be frustrating and time-consuming [Ste00]. Currently, understanding software is still done by reading the source code. Since control software is highly contextual, considering software parts in the context of the entire software system is essential for understanding. However, the few available tools are too rigid and generic to consider the context [GC15]. Although static code analysis tools can ease and shorten gaining analysis results, Prähofer et al. [Prä⁺17] emphasize that the tools often only provide hints that need to be examined in detail by industry experts. In summary, a purely automatic static code analysis is not sufficient to consider control software in its context and assess its quality; instead, the rationale behind the code needs to be considered.

Requirement R_{Rat} – Consideration of Code Rationale

During the software analysis and the interpretation of results, the intention behind the control software, e.g., the implemented functionality (potentially indicated by metadata such as comments or naming conventions), needs to be considered for the software quality assessment.

Finally, the last category of requirements targets the use of the gained analysis results.

3.4. Use of Analysis Results to Derive Recommendations for Action

Following the procedure, the performed static code analysis steps should support assessing the analyzed control software regarding the chosen analysis goal and reveal strengths and weaknesses. However, solely identifying weaknesses is not sufficient to support companies in improving their control software since they need to estimate how costly the elimination of a weakness will be as a basis for choosing the quality measures to be taken or improvements to be made. Therefore, an estimation of the expected change effort, including different aspects such as changes in the software itself or required adaptations in the development workflow, is needed.

Requirement R_{Weak} – Weaknesses and Change Effort

The procedure should identify the strengths and weaknesses of the analyzed software concerning a chosen analysis goal, including a qualitative estimation of the expected change effort for eliminating the weaknesses.

Depending on the maturity of the software development process, the availability, amount and quality of software documentation vary significantly. Especially in historically grown legacy software, often, no documentation or only outdated documentation is available [Kir⁺16]. Instead, the knowledge about design decisions or functionality distribution is only known by the software developer himself/herself, who, in the worst case, has already retired or left the company. In companies with programming guidelines and templates, the conformance of the application projects to these standards is an essential point, especially in the case of different software developers, maybe even at different sites, programming one machine/plant. Thus, in both cases, documenting the analysis results, the software itself and identified design decisions is required and helpful for planning and conducting software improvement steps. Moreover, generating up-to-date documentation improves the comprehension of legacy software [Kir⁺16].

Requirement R_{Doc} – Documentation of Analysis Results

Documentation of the analysis results and gained insights during the application of the quality assessment procedure, such as identified design decisions, is essential for planning and implementing software improvements to increase software quality.

3.5. Limitations of the Concept's Scope

The procedure presented in this thesis covers the analysis of existing brownfield control software, i.e., legacy software, for its quality assessment, including the identification of beneficial design decisions and improvement potentials. However, specific aspects regarding the development and reuse of high-quality control software are out of the scope of this work. These are listed in the following as limitations of the concept's scope.

The quality assessment procedure proposed in this thesis is not a design guideline or overview of best practices for developing high-quality greenfield control software. Instead, it is limited to providing the means for assessing brownfield, legacy control software and identifying improvement potentials. However, the legacy software to be analyzed must not be monolithic; at least attempts of functionality encapsulation are required to successfully apply the proposed quality assessment procedure.

The developed procedure does not explicitly target the analysis of OO elements of the third edition of IEC 61131-3 since not all PLC suppliers support the development of OOP control software yet. Moreover, no OOP elements were contained in the analyzed industrial legacy software projects. The concept was evaluated with classical, procedural IEC 61131-3 control software. Embedded systems and general-purpose or high-programming languages, which can be used on specific controllers in combination with approaches from computer science, are not considered either.

Regarding the documentation of analysis results and gained insights, the procedure neither provides an exhaustive list of available documentation types nor specifies and links analysis goals to their ideal type of documentation. Thus, the proposed concept does not support the choice of appropriate means of documentation but only provides selected examples.

The presented procedure was developed for and evaluated with software from factory automation (discrete processes, including intralogistics); process automation and the control of continuous processes are not considered. Furthermore, assessment of the considered control software's functional correctness in the sense of software testing and verification is not targeted in the scope of this thesis.

4. State-of-the-Art

Utilizing the requirements derived in the previous chapter, current research works were reviewed and rated for their applicability to the domain and problem. In addition, adjacent domains such as computer science were investigated. The following Sections present approaches targeting static code analysis and software metrics (Section 4.1) and quality assessment procedures (Section 4.2). In each Section, the reviewed approaches are compiled into a table, rating each approach regarding the requirements derived in Chapter 3. The applied rating scheme is summarized in Table 3. The resulting research gap is formulated in Section 4.3.

Table 3: Details of rating scheme for the evaluation of existing, related approaches with + (completely satisfied), o (partially satisfied), – (not satisfied), ? (unknown) and n.a. (not applicable).

Requirement	Rating Scheme
Platform Independence (R_{PLC})	+: Approach is independent of the PLC programming platform o: Approach is applicable to control software of a specific PLC programming platform -: Approach is not applicable to PLC software/used with general-purpose languages
aPS as Product (R_{Pro})	+: Applicable to control software from different aPS types o: Not in focus, but applied in different companies with different aPS types -: Not in focus and only used in one company
User (R_{Use})	+: Approach is intended for independent use by software developer o: Intended for independent use by software developer, but not evaluated -: Approach requires an external expert
Application Sector (R_{Sec})	+: Analysis is adaptable to boundary conditions of different application sectors o: Applicable in different sectors, but no specific consideration of boundary conditions -: Tailored to a specific application sector/applied in only one application sector
Pain Points (R_{PP})	+: Identification and consideration of pain points during the analysis o: Pain points are identified but not considered in the analysis -: Identification not targeted
Workflow Integration (R_{Work})	+: Integration into development workflow (optionally at different points) possible o: Integration into development workflow not targeted, but likely possible -: Integration into development workflow not targeted and unlikely
Design Decision (R_{DD})	+: Deliberate design decisions are considered as constraints for analysis and assessment o: Deliberate design decisions are not specifically investigated; consideration likely possible -: Deliberate design decisions are not specifically investigated; consideration is unlikely
Analysis Goal (R_{Goal})	+: Applicable to different goals o: Applicable to a specific goal only or a very general goal -: Analysis goal is not specified
Scalability (R_{Scal})	+: Approach was applied to industry-sized control software o: Approach was applied to small examples, likely applicable to industry-sized control software -: Approach has not been applied to industry-sized control software
Application Effort (R_{Eff})	+: Application effort lowered with repeated application o: Application effort not investigated; lowering likely possible -: Application effort not targeted; lowering unlikely possible
Code Rationale (R_{Rat})	+: Combination of manual and automatic static analysis, including interpretation by developer considering code rationale o: Analysis results are interpreted by the developer, but code rationale is not focused -: Consideration of code rationale is not targeted
Weaknesses and Change Effort (R_{Weak})	+: Identified strengths and weaknesses support effort estimation o: Strengths and/or weaknesses are identified, effort estimation not targeted/unlikely -: Strengths and weaknesses not targeted
Documentation (R_{Doc})	+: Documentation incl. analysis results on various granularity levels to derive recommendations o: Documentation includes analysis results, likely helpful for deriving recommendations -: Documentation includes analysis results, support to derive recommendations unlikely

4.1. Static Control Software Analysis and Software Metrics

This Section presents approaches for the static code analysis of control software, including prototypical tools (Sub-section 4.1.1), control software metrics (Sub-section 4.1.2) and available commercial tools developed and provided by PLC platform suppliers (Sub-section 4.1.3).

4.1.1. Static Code Analysis targeting Control Software

Static code analysis supports the development of high-quality software by detecting potential errors and defects in early development phases without executing the code. Approaches to applying static code analysis targeting control software are discussed below and summarized in Table 4.

Prähofer et al. [Prä⁺12] present an approach and the tool *SCoRe* (Source Code Review) for a rule-based static code analysis of software programmed according to a proprietary dialect of IEC 61131-3 (R_{PLC}). The authors analyzed recurring issues of their industry partner (R_{PP}) and classified these into eight categories ranging from violations over naming conventions to dynamic statement dependencies [Prä⁺17]. *SCoRe* was integrated into the GPL static analysis platform SonarQube and used to evaluate industry-size PLC projects for injection moulding machines programmed in ST and SFC (R_{Pro} , R_{Sec} , R_{Scal}). The most recent version of *SCoRe* includes a set of 46 rules, which implement the company's guidelines (R_{DD}), for checking the source code regarding bad programming practices. According to Prähofer et al., the rules can be adapted to different application sectors and programming guidelines (R_{Pro} , R_{Sec} , R_{DD}) [Prä⁺17]. However, so far, it has been used in one company only. *SCoRe* offers call graph and pointer analysis [Ang⁺13] as well as control flow and data flow analysis [Prä⁺17]. The identified violations are displayed as a list in the SonarQube dashboard view (R_{Doc}). Furthermore, the program elements and dependencies can be visualized and queried using a web frontend [Ram⁺19]. For three years, the company has integrated *SCoRe* into its development process as part of the nightly built (R_{Work}) and the software developers use it daily (R_{Use}). According to the authors, the analysis provides hints, which are examined by the software developers (R_{Rat}). Apart from manually resolving or suppressing identified violations, *SCoRe* does not support identifying strengths in the analyzed control software or deriving recommendations for action (R_{Weak}). Moreover, albeit the authors identified recurring issues, no further support to identify pain points and define an analysis goal is provided (R_{Goal}).

Another static code analysis tool for control software was developed by the research group of Kowalewski in the scope of their model-checking and test case generation framework *Arcade.PLC* [BBK12]. *Arcade.PLC* supports analysis of PLC software written in ST, IL, SFC and FBD, including vendor-specific extensions for ABB's Compact Control Builder, CODESYS and Siemens SIMATIC S7 statement list (R_{PLC}). From source code text files or PLCopen XML files, the PLC

software is translated into an intermediate representation to enable a language-independent quality control (R_{Goal}) [BBK12]. More precisely, it checks the source code for general rules regarding runtime errors and code smells (R_{Goal}) based on an approximation of variable value ranges without considering application sector- or company-specific guidelines (R_{Pro} , R_{Sec} , R_{PP} , R_{DD}). By executing a set of predefined checks, aspects like array out-of-bounds access, unreachable branches in conditions, division by zero or multiple assignments to an output are identified [Sta⁺14]. Moreover, Simon and Kowalewski perform a structure-preserving analysis to generate warnings for erroneous SFC behavior based on variable values and structural checks [SK16]. Static code analysis with *Arcade.PLC* is intended for application by the software developer (R_{Use}) and was evaluated with industry-sized projects programmed in ST in ABB's Compact Control Builder (R_{Scal}) [Bia16; Sta⁺14]. Identified violations (R_{Weak}) are displayed to the user in a list, including the source code position (R_{Doc}). Stylistic warnings like redundant compares or dead code have to be inspected manually. Also, missing information resulting from encrypted library modules requires manual review to decide which warnings should be resolved (R_{Rat}) [Sta⁺14].

Obster et al. proposed an architecture to extend *Arcade.PLC* for live static code analysis integrated into the PLC development environment to provide instant feedback to software developers while editing the source code (R_{Work}) [OK17]. The approach was implemented for software written in ST in the CODESYS development environment (R_{PLC}) and evaluated with two small test programs (R_{Scal}) by seven industry experts from two companies (R_{Use}) [Obs21]. During editing, *Arcade.PLC* checks for code smells and errors detectable by calculating value set ranges of variables (R_{Goal}). The developer receives feedback in a text editor view and via annotations of suspicious code parts (R_{Doc}) and needs to decide which of the identified weaknesses require resolving (R_{Weak} , R_{Rat}) [OK17]. Neither application- and aPS type-specific boundary conditions nor pain points and design decisions are targeted by the approach (R_{Pro} , R_{Sec} , R_{PP} , R_{DD}).

The research group around Jetley at ABB targeted different goals for applying static code analysis to aPS control software. Using a platform-neutral implementation of IEC 61131-3 (R_{PLC}), Jetley et al. compared project versions of FBD and SFC control software to support version management (R_{Goal}) and provided a color-coded visualization of identified changes (R_{Doc}) [Jet⁺13a]. Their prototype was applied by software developers (R_{Use}) during the software development (R_{Work}) of industry-sized control software projects (R_{Scal}). For the identification of circular dependencies (R_{Goal}), i.e., code loops, in ABB control software (R_{PLC}), Nair and Jetley developed an approach and prototype to visualize the code loops on different abstraction levels (R_{Doc}) [NJ16]. The tool provides resolution suggestions, which are checked by the software developer (R_{Rat} , R_{Use}), and was successfully applied to industry-sized legacy control software (R_{Scal}). Finally, Nair et al. use a

combination of data flow analysis and customizable rule checks to detect runtime errors and violations of coding guidelines or best practices (R_{Goal}) in IEC 61131-3 ST and FBD control software (R_{PLC}) [Nai⁺15]. A proof-of-concept prototype generates the Abstract Syntax Tree (AST), the Control Flow Graph (CFG) and calculates variable value ranges to check for aspects such as type constraints, division by zero, unreachable code, uninitialized and unused variables, datatype mismatches and non-terminating loops. The tool supports the definition of customized rules to check naming conventions and identify incorrectly set attribute values and nesting limitations (R_{DD}). The prototype was included in an analysis framework for industrial automation software, which supports three customized rules for IEC 61131-3 control software based on errors mentioned by domain experts (R_{PP}) [Man⁺18]. The approach was applied to an industry-sized control software project (R_{Scal}). Identified violations, which are classified as warnings and errors (R_{Weak}) and displayed in a list (R_{Doc}), were validated by software developers utilizing their domain knowledge (R_{Use} , R_{Rat}) [Nai⁺15]. However, application- and aPS type-specific boundary conditions are not targeted by the approach (R_{Pro} , R_{Sec}).

With the aim to detect semantic and structural duplicates in IEC 61131-3 ST control software (R_{PLC} , R_{Goal}), Jnanamurthy et al. propose a combination of structural, semantic and data interval-based analysis [Jna⁺20]. The approach addresses the pain points and analysis goals of code optimization, bug detection and analysis of reused code (R_{PP} , R_{Goal}). To detect semantic clones, i.e., duplicated code parts, they use an I/O variable impact and dependency analysis. Limiting their analysis to hardware inputs and outputs reduces the amount of data to be analyzed. The authors evaluated their approach with real-world industrial library modules of machine control software (R_{Scal} , R_{Pro}) [Jna⁺20]. Results are documented in a table format within the paper, but no additional information regarding the documentation is provided (R_{Doc}). Based on the results, duplicated code parts are identified as a basis for refactoring (R_{Weak}). Neither manual code analysis nor the consideration of design decisions or the workflow integration are targeted (R_{Rat} , R_{DD} , R_{Work}).

Preliminary work in the research group of Vogel-Heuser comprises a prototypical tool for the analysis of Siemens SIMATIC Manager control software (R_{PLC}). The prototype includes dependency graphs and software metrics and supports identifying structural, recurring patterns [Fuc⁺14]. To support manual interpretation (R_{Rat}), the presence and absence of the structural patterns were rated in [Neu⁺20c]. Furthermore, the search for structural clusters in the call graph enabled, even without considering the implemented functionality, the identification of copied software parts across several project variants as a basis for refactoring and increasing the planned reuse of legacy control software (R_{Goal}) [Fah⁺19]. Feldmann et al. present an analysis framework to evaluate control software quality utilizing Semantic Web technologies [Fel⁺16a]. More precisely, a dependency model of the control software is created, which can be queried to check user-defined programming

guidelines (R_{DD}), application sector-specific guidelines (R_{Sec}), calculate software metrics or search for known disadvantageous code structures (R_{PP}). The approach was integrated into Schneider Electric's development environment (R_{PLC}) in the scope of a cloud-based analysis concept to support the continuous integration of the analysis means in the development workflow (R_{Work} , R_{Use}) [Bou⁺17]. Apart from call and data exchange graphs to identify beneficial and disadvantageous structures (R_{Weak}) [Fuc⁺14], the cloud-based code analysis offers a dashboard results view for managers (R_{Doc}). The analysis tools have been applied to industry-sized control software projects (R_{Scal}) from different application sectors (R_{Sec}) and aPS types (R_{Pro}). Moreover, the manual static code analysis of several industry-sized control software projects with different focuses (R_{Scal} , R_{Goal}) provided insights on architectural hierarchy levels and the implementation of extra-functional tasks such as error handling and operation modes [Vog⁺15a; Vog⁺16]. The individual approaches of Vogel-Heuser's group fulfill the derived requirements to different extents but do not provide a systematic procedure for a goal-oriented quality assessment considering company-specific boundary conditions and identified pain points. The concept developed and proposed in the scope of this doctoral thesis is based on the insights gained by Vogel-Heuser's research group.

Thaller et al. apply static code analysis in an industry-sized PLC software project (R_{Scal}) written in the textual IEC 61131-3 language ST and C/C++ (R_{PLC}) to identify POU's or POU parts, which have been duplicated for reuse, e.g., by copying and pasting (R_{Goal}) [Tha⁺17]. They target the identification of two different types of duplicated software, i.e., exact copies and parameterized copies [RBS13]. For the analysis, the authors extend the Simian tool (R_{Use}). Identified clones are classified and rated regarding their relevance by a group of experts (R_{Rat}). The considered machine control software (R_{Pro}) from the metal processing sector (R_{Sec}) was developed as a 150%-model, i.e., containing control software for different variants of the machine type, which results in a high amount of duplicated code (R_{PP}). The duplicates are identified as a basis for refactoring, but deriving recommendations for action is not targeted (R_{Weak}). Moreover, the authors conclude that existing tool support for detecting duplicates in PLC software is insufficient.

Reengineering is often applied to enhance the quality of historically grown software. It consists of steps such as code analysis, assessment of pain points and migration from legacy software into a new software structure [NDG05]. During this process, apart from software, additional available documentation of different formats is used for software understanding (R_{Rat}). For the rapid development of customized analysis tools, Nierstrasz et al. developed the language-independent platform *Moose* as a meta-tooling environment (focused on OO and GPL) (R_{PLC}) to support program understanding and reengineering [NDG05]. *Moose* combines different analysis techniques, including metrics evaluations, a repository for source code versions, visualization and querying and browsing of results (R_{Doc} , R_{Weak} , R_{Use}). Depending on the targeted analysis goal during the software

assessment (R_{Goal}), respective means can be rapidly combined in a prototype. Moreover, scalability is considered an essential requirement due to the size of legacy software (R_{Scal}) [Nie12].

Canedo et al. propose *ArduCode*, a machine-learning-based approach for supporting PLC software developers' quality-related decision making [Can⁺21]. Their goal is to apply machine learning to assist in PLC code classification according to the implemented functionality, identify similar code parts and support hardware selection and configuration (R_{Goal}). The approach has been prototypically implemented as a plug-in into Siemens TIA Portal (R_{PLC}), including a user interface (R_{Use}), with the aim to only minimally disrupt existing development workflows (R_{Work}). The OSCAT library has been used for evaluation purposes, which is mainly implemented in SCL (R_{Scal}). During programming in the TIA Portal, results are directly displayed (R_{Doc}), including recommendations for action for resolving identified weaknesses (R_{Weak}). The user can choose the recommendations to be carried out (R_{Rat}), ignore recommendations (R_{DD}) or provide feedback to the assistance system, which enhances the tool's knowledge base and, thus, continuously lowers the application effort (R_{Eff}). The identification of pain points is not targeted (R_{PP}).

Some companies and organizations, such as CERN, use their own development frameworks to support PLC software development for different platforms [Fer⁺15]. The *Unified Industrial Control System* (UNICOS) used at CERN generates control software from a list of field devices linked to the PLC as a base for application software development. *UNICOS* supports software development for Siemens and Schneider Electric PLCs (R_{PLC}). Due to known pain points resulting from evolution, e.g., nested if-statements, a high number of input conditions, dead code and long expressions (R_{PP}), Ferreira et al. target the identification of these pain points and violations of company-specific naming conventions (R_{DD}) for improving the overall software quality (R_{Goal}) [TBF17]. They implemented 19 basic analysis rules in the model checker *PLCverif* [Lop⁺21] used at CERN (R_{Work} , R_{Use}) and applied them to individual FBs (R_{Scal}). The identified violations are reported as a list to the user (R_{Doc}) for subsequent correction (R_{Weak}). Neither application sector-specific rules nor consideration of the implemented functionality are targeted (R_{Sec} , R_{Rat}).

In the application sector of electrical engineering and steelmaking (R_{Sec}), Klammer and Pichler [KP14] analyzed four industry-sized legacy control software programs (R_{Scal}) written in Fortran, PL/SQL and C++ (R_{PLC}). Their analysis focused on identifying domain knowledge in the analyzed code as part of reengineering and re-documentation of industrial legacy software (R_{Goal}). With the aim of developing a multi-language analysis tool, they propose an analysis workflow for linking code instructions via manual annotations to domain knowledge concepts (R_{Rat}). These annotations, i.e., physical units and domain concepts, are usable to search for specific computations in the source code. Moreover, overlays support program comprehension by considering design decisions

and domain knowledge (R_{Sec}). From a code repository, source code fragments are annotated with meta-tags, which are considered during the dependency and data flow analysis and displayed to the user (R_{Doc} , R_{Use}). Based on the tool, Kirchmayr et al. generate documentation of legacy control code on different abstraction levels by considering units and mathematical formulae (R_{Doc}) [Kir⁺16]. Due to the goal of identifying domain knowledge, neither pain points nor design decisions or the identification of strengths and weaknesses are targeted (R_{PP} , R_{DD} , R_{Weak}).

Various approaches are available, which target the one-time static code analysis of a single control software project or a group of projects. These are usually tailored to a specific company or application sector and a single analysis goal. Moreover, multiple use of the applied analysis process is not targeted. An example of this is the study by Ljungkrantz and Åkesson [LÅ07]. They analyzed Mitsubishi PLC projects from two companies in the automotive sector (R_{Sec}) regarding the number of instantiations of library FBs. Furthermore, they sorted the used FB instances into nine categories to analyze the functionality distribution in the projects and, thus, gain insights into industrial practices (R_{Goal}). For their analysis, Ljungkrantz and Åkesson programmed a tool to identify and count FB instances and considered company-specific programming guidelines and templates (R_{DD}) when manually assigning the categories (R_{Rat}).

Jung et al. target the conformance checking of control software for safety-critical applications in nuclear power plants (R_{Pro} , R_{Sec}) programmed in FBD with the safety guideline Nu-REG/CR-6463 (R_{Goal}) [JYL17]. Based on the PLCopen XML format (R_{PLC}), their tool *FBD Checker* identifies violations of 56 structural analysis rules, e.g., implicit type conversion, missing inputs to FBD blocks, missing initial values of variables and the graphical program layout for readability (R_{Rat}). Identified violations are listed, including the violated rule, and graphically highlighted in the program (R_{Doc}) to enable their correction (R_{Weak}). The approach was applied to the FBD program of a Korean nuclear power plant (R_{Scal}) during the design phase (R_{Work}). Apart from the structural rules, no other design decisions are targeted (R_{DD} , R_{Rat}).

Recent approaches target static code analysis of control software programmed following the IEC 61499 (R_{PLC}). For example, the research group of Zoitl supports the identification of code smells [Son⁺21a] and their refactoring with a catalog of refactoring operations [Obe⁺21]. A combination of both is targeted in future work to support the user (R_{Use}) in improving the software quality (R_{Goal} , R_{Weak} , R_{Doc}). Furthermore, their research focuses on design patterns: three architectural design patterns for IEC 61499 are implemented for a lab-sized application and compared based on complexity metrics to determine their scalability for industry-sized applications [WSZ20]. A skill-based adaptation of the distributed hierarchical control pattern is applied to a more complex lab-sized factory automation application (R_{Sec} , R_{Scal}) in [Son⁺21b]. To assess the

proposed design, the module granularity (amount of FBs and their functionality), reuse (number of instantiations) and understandability and adaptability are analyzed with software metrics and compared to two other designs (R_{DD} , R_{Rat}). However, due to the different programming paradigms, their IEC 61499-based concepts are not directly applicable to IEC 61131-3 control software.

The approaches from the field of static control code analysis are summarized in Table 4.

Table 4: Evaluation of related approaches in the field of static code analysis.

Approach	Platform Independence (R_{PLC})	aPS as Product (R_{Pro})	User (R_{Use})	Application Sector (R_{Sec})	Pain Points (R_{PP})	Workflow Integration (R_{Work})	Design Decision (R_{DD})	Analysis Goal (R_{Goal})	Scalability (R_{Scal})	Application Effort (R_{Eff})	Code Rationale (R_{Rat})	Weaknesses, Change Effort (R_{Weak})	Documentation (R_{Doc})
Group of Pr�hofer [Ang ⁺ 13; Pr� ⁺ 12; Pr� ⁺ 17; Ram ⁺ 19]	o	o	+	+	+	+	o	o	+	n.a.	o	o	o
Kowalewski et al. [BBK12; Bia16; SK16; Sta ⁺ 14]	+	n.a.	o	n.a.	n.a.	o	n.a.	o	+	n.a.	+	o	o
Obster, Kowalewski et al. [Obs21; OK17]	o	n.a.	+	n.a.	n.a.	+	n.a.	o	o	n.a.	+	o	o
Jetley, Nair et al. [Jet ⁺ 13a; Man ⁺ 18; Nai ⁺ 15; NJ16]	o	n.a.	+	n.a.	o	o	o	o	+	n.a.	+	o	o
Jnanamurthy et al. [Jna ⁺ 20]	o	o	?	n.a.	+	n.a.	n.a.	o	+	n.a.	-	o	o
Group of Vogel-Heuser [Bou ⁺ 17; Fah ⁺ 19; Fel ⁺ 16b; Fuc ⁺ 14; Neu ⁺ 20c]	+	o	+	+	o	+	+	o	+	n.a.	o	o	+
Thaller et al. [Tha ⁺ 17]	o	o	-	n.a.	n.a.	n.a.	-	o	+	n.a.	+	o	o
Nierstrasz [NDG05; Nie12]	-	n.a.	+	n.a.	-	+	o	+	+	n.a.	+	o	+
Canedo et al. [Can ⁺ 21]	o	n.a.	o	n.a.	-	+	o	+	o	+	o	o	+
Research group at CERN [Fer ⁺ 15; Lop ⁺ 21; TBF17]	+	n.a.	o	o	+	+	o	o	o	n.a.	-	o	o
Klammer and Pichler [Kir ⁺ 16; KP14]	-	o	+	+	-	-	n.a.	o	+	n.a.	+	-	+
Ljungkrantz, �kesson [L�07]	o	o	-	o	-	o	+	o	+	n.a.	+	-	o
Jung, Yoo and Lee [JYL17]	o	o	o	o	-	o	-	o	+	n.a.	o	o	+
Group of Zoitl [Obe ⁺ 21; Son ⁺ 21a; Son ⁺ 21b; WSZ20]	o	n.a.	o	o	-	n.a.	o	o	-	n.a.	+	o	o

Overall, existing approaches for static code analysis often target a specific PLC platform (R_{PLC}) and some of them have not been applied to industry-sized code at all (R_{Scal}) or only to control software of one company and, thus, in one application sector (R_{Sec}). Furthermore, visualization of results, identifying beneficial design decisions and deriving recommendations for actions are usually not the focus (R_{Weak} , R_{Doc}). In addition, most of the approaches do not support deriving pain points (R_{PP}) as a basis for a goal-oriented (R_{Goal}), systematic static code analysis. Adaptations of the analysis to company-specific boundary conditions and design decisions are not in focus (R_{DD}). Finally, static code analysis tools neither support the results interpretation nor the analysis planning, making their successful application challenging for inexperienced software developers.

4.1.2. Overview of Software Metrics Focusing on Control Software

As introduced in Section 2.4, software metrics enable quantifying specific quality characteristics by counting code properties of individual elements and their dependencies. The use of software metrics is a well-established means in computer science. Popular measures for software quality in OO software are, according to [Jab⁺15], the metric suite of Chidamber and Kemerer [CK94] or QMOOD [BD02]. In contrast, software metrics are less common in the aPS domain [Prä⁺12].

Nevertheless, some approaches are available that apply common software metrics from computer science in the aPS domain. These metrics include *Lines of Code (LOC)*, which measures a program's length by counting the code lines, and *Source Lines of Code (SLOC)*, which measures the lengths by counting non-commentary, non-empty lines only [Ros97]. Also, McCabe's *Cyclomatic Complexity*, which is based on the software's control flow graph [McC76], and Halstead's metrics counting operands and operators for measuring the length, vocabulary and difficulty of a software program [Hal77] are applied in the aPS domain. Moreover, the *Fan-in Fan-out* metric based on the information flow to and from a given software part [HK81] has been transferred to control software. Overall, metrics from computer science are not applicable to graphical programming languages and do not consider the significantly different boundary conditions, e.g., close connection to hardware, of PLC software [Jet⁺13b]. Thus, they are not transferrable to IEC 61131-3-based control software without adaptations.

Accordingly, Younis and Frey [YF07] transfer popular metrics from computer science [Hal77; McC76; Ros97] to a didactic and an industrial example to measure the diagnosability of PLC software written in IL. Capitán and Vogel-Heuser successfully adapt the metrics from [McC76] and [Hal77] to all five PLC programming languages using a lab-sized demonstrator and illustrate that the metrics are, in principle, suitable for determining the complexity of aPS software [CV17]. Similar, Kumar et al. introduce source code metrics for control software written in LD for measuring program length, difficulty, or cognitive complexity and, thus, quantify the understandability

and testing complexity [KJS16]. However, the approach is not evaluated with industrial control software. In contrast, the ten ST metrics by Kumar et al. for predicting change proneness at POU level were evaluated with two industrial projects [KS17]. For assessing the usability and maintainability of SFC programs, Engell et al. [EDL07] develop new metrics based on McCabe's *Cyclomatic Complexity* [McC76], including a results visualization, and evaluate them with an academic example.

Muslija and Enoiu [ME20] applied the computer science metrics [Hal77; HK81; McC76] to control software written in FBD and adapted the size metric SLOC to measure the complexity in safety-critical PLC software. Their approach was evaluated with an industrial case study. Similarly, the approach by Wilch et al. [Wil⁺19] for identifying the most complex networks in industrial FBD control software is based on Halstead's metrics [Hal77] and considers the information flow [HK81]. It has been implemented as part of the Schneider Electric *Machine Expert* code analysis plugin for evaluation purposes [SE22a]. The implementation includes the documentation and visualization of the calculated metric results to identify the most complex networks for their subsequent refactoring and has been utilized by the software developers of a German machine manufacturer. Similarly, Prähofer et al. implemented software metrics in their prototypical analysis tool *SCoRe* to measure control software complexity. The tool indicates if defined upper limits, e.g., for the number of lines or branch depths, are exceeded [Prä⁺12]. The implemented metrics are LOC for each POU, McCabe's *Cyclomatic Complexity* and the number of comment lines [Prä⁺17].

Moreover, some researchers propose software metrics tailored explicitly to the aPS domain. For example, Lee and Hsu transform PLC software programmed in LD into Petri nets to compare both regarding their design complexity and understandability using a small industrial example [LH01]. Gharieb proposed four metrics targeting simplicity, reconfigurability, reliability and flexibility of LD and applied them to a lab-sized application [Gha06]. Also targeting LD software, Lucas and Tilbury developed metrics to measure the size, modularity and interconnectedness of the PLC software programs and evaluated them with a test-bed [LT05]. Nair presents a methodology framework in four phases for defining IEC 61131-3 software metrics systematically [Nai12]. Using the framework, a set of product metrics for measuring the reusability and reliability of POU is defined, which was applied in a real project. With a focus on non-functional requirements, Ladiges et al. assess machine, process, routing and operation flexibility as a pre-step for defining metrics to measure their fulfillment [Lad⁺13]. Rosiak et al. propose a metric-based approach for the re-engineering of variant-rich IEC 61131-3-based legacy software into SPLs for their subsequent planned reuse [Ros⁺21a]. The used similarity metrics are tailored to the programming style and the programming languages. So far, they have not been applied to industry-size PLC software.

The group of Vogel-Heuser developed metrics to estimate the criticality of changes performed during the evolution of PLC control software [Vog⁺18]. Based on the changes, the maturity of modified POU's is estimated. The approach was successfully implemented and industrially evaluated for Siemens PLCs [Neu⁺20b] and IEC 61131-3-based control software [VNF22]. Moreover, it was integrated into the software development workflow of a company and applied by the software developers [VNF22]. In addition, Neumann et al. developed software metrics to identify OO code constructs within IEC 61131-3-based control software, which are likely to lead to an increased runtime [Neu⁺20a]. The runtime metrics have been evaluated with the control software of a packaging machine company.

In the scope of IEC 61499 control software, Sonnleithner et al. propose a set of software measures, which were applied to a small implementation example [SZ20]. Another example is the approach by Zhabelova and Vyatkin, who adapted, among others, McCabe's *Cyclomatic Complexity* and Halstead's metrics to IEC 61499. Additionally, they suggest metrics for quantifying the complexity of Execution Control Charts (ECC) [ZV15]. However, IEC 61499-based concepts are not directly applicable to IEC 61131-3 control software.

Overall, some software metrics for PLC control software are available, with most of them being platform-specific in their concept or implementation (R_{PLC}). Moreover, many of the metrics recently developed in the aPS domain have not yet been applied in an industrial context (R_{Scal}) or, so far, only in one company (R_{Pro} , R_{Sec}). Although software metrics offer consistent, reproducible results, they are not a definite quality judgment but rather indicate possible weaknesses (R_{Weak}) [Wil⁺19]. Moreover, they usually target a specific quality attribute and are, thus, not applicable to different analysis goals (R_{Goal}) and lack adaptability to company-specific design decisions (R_{DD}). So far, the choice of a suitable software metric for analyzing the control software concerning a specific pain point or analysis goal is not supported. Since the use of software metrics for control software is relatively new, PLC software developers lack experience in interpreting the calculated metric results, which is not trivial, even if approaches such as [VNF22] target the interpretation of results in the context of the analyzed software and its evolution. Consequently, deriving recommendations for action from the gained and documented analysis results is often challenging (R_{Weak}) and requires adequate documentation and visualization of the metric results (R_{Doc}). In conclusion, software metrics are a suitable means for assessing control software in regard to specific quality attributes, which is beneficial but not enough for the overall quality assessment [Plö⁺10]. Thus, a concept for the goal-oriented, context-aware quality assessment of control software should include software metrics but not be limited to them.

4.1.3. Commercial Tool-support for PLC Software Analysis

In high-level programming, various tools are available that support static code analysis. Some of these are directly integrated into the programming environment, such as the extension *ReSharper* for Visual Studio [Jet22], and provide direct feedback about quality defects during programming. Moreover, GPL tools such as *SonarQube* [Son22] support refactoring operations, including explanations about the identified defect and the strategy to resolve it. Some tools, such as CQSE Teamscale support, among other textual languages, ST code analysis [CQS22]; however, control software analysis is not the tool's primary focus.

In the aPS domain, PLC platform suppliers offer first applications for static code analysis, including conformance checks to guidelines and software metrics, in their development environments (cf. Table 5). For example, Beckhoff TwinCAT 3 *Static Analysis* enables checking if the source code follows specified coding guidelines utilizing rules and naming conventions and calculating metrics. The analysis can be triggered manually or performed automatically and identified violations to activated rules are reported as warnings and errors [Bec22]. Available rules and metrics are contained in [Bec21] and include prefixes for specific POU or variable types and also support the definition of rules with regular expressions (R_{DD}). Similarly, CODESYS *Static Analysis* [COD22] offers a set of rules (cf. [COD19] for a list), which can be (de-)activated if needed, to check the PLCopen guidelines [PLC16], prefixes of variables specified in naming conventions and also supports the calculation of software metrics. Moreover, it offers the detection of code duplicates in software parts programmed in ST in the *Code Clone Detection*. First means for refactoring are available, e.g., extracting a marked code snippet into a new FC or method.

Formerly known as *Itris PLC Checker*, Schneider Electric offers a set of four tools in the scope of *EcoStruxure Control Engineering*, e.g., a reverse engineering tool for redocumentation of the code structure and design recovery [SE22b]. The documentation tool offers a dependency tree, a data flow and a control flow view, supporting the analysis of different PLC platform applications such as CODESYS, Beckhoff and Siemens (R_{PLC}) [SE21]. Additionally, Schneider Electric's *Machine Code Analysis* [SE22a] is directly integrated into the PLC development environment. It supports software metrics and convention checks, e.g., conformity to PLCopen rules [PLC16], to measure the software quality of IEC 61131-3 source code. Moreover, options to visualize, query and document dependencies within the control software are available. For creating queries, a SPARQL Editor and predefined SPARQL queries are provided (R_{DD}). The analysis results are reported, including key performance indicators (KPIs), in a dashboard view (R_{Doc}).

Logicals offers generic style guide checks in *logi.CAD Static Analysis* with pre-defined IEC 61131-3 design rules for ST, FBD and LD [log22a]. Available rules, e.g., references to local variables from methods and functions, are listed and described in the user guide [log22b] and reported as a list classified as information, warning and error. In the scope of *TIA Portal Test Suite Advanced*, Siemens offers to define programming style guides and check their compliance for selected POU's or entire applications. In the rule set editor, five categories of rules can be defined, which mainly target naming conventions, i.e., casing, prefix/suffix, name length, and additionally the completeness of object properties such as the existence of author, title, version and comments in the metadata [Sie20b]. Identified violations are displayed in the TIA Portal and, additionally, reported in a log file in two selectable criticality levels. Additionally, Siemens offers the *Project Check* for TIA Portal software [Sie20a], which checks the software against the Siemens programming style guide [Sie18], including naming conventions and general rules, e.g., for objects referencing, not requiring code analysis. It also supports the definition of rules by the user and exporting a report. Finally, the *Safety Code Analysis Tool* from ABB checks the conformance of a safety PLC project according to safety-specific coding guidelines [ABB21]. These include aspects such as using variables instead of literals, avoiding implicit type conversions and adding a comment to each variable declaration. The results can be generated and exported.

Overall, all available tools enable the analysis of industry-sized control software projects (R_{Scal}) regardless of the aPS type (R_{Pro}) and can be integrated into the development workflow (R_{Work}). Moreover, they are intended for use by the software developer (R_{Use}). However, except for *EcoStructure Control Engineering*, the available tools are tailored to the vendor's respective PLC development environment (R_{PLC}). Although many suppliers support the export in PLCOpen XML, the standard is not sufficient since, for example, vendor-specific libraries for actuators, e.g., for motion control, are not transferable between different PLC platforms. Moreover, the implementation of the IEC 61131-3 standard differs between platforms [DZ21; Jet⁺13b].

In addition, the tools are limited in their scope of adaptability. Although general rules or rules for naming conventions are available for conformance checks, the tools are not yet capable of tailoring the analysis to the company-specific or application sector-specific programming guidelines and boundary conditions (R_{Sec} , R_{DD}). Moreover, although first approaches to support refactoring are available, most tools do not yet support deriving recommendations for actions for using the insights gained from the static code analysis (R_{Weak} , R_{Doc}). Finally, no support for the identification of pain points as a basis for deriving an analysis goal is provided (R_{PP} , R_{Goal}). In summary, first means are available, but their systematic goal-oriented use is not yet sufficiently supported.

The available static code analysis tools for PLC software are summarized in Table 5.

Table 5: Tool-based static code analysis of PLC software evaluated with respect to the requirements.

Approach	Platform Independence (R_{PLC})	aPS as Product (R_{Pro})	User (R_{Use})	Application Sector (R_{Sec})	Pain Points (R_{PP})	Workflow Integration (R_{Work})	Design Decision (R_{DD})	Analysis Goal (R_{Goal})	Scalability (R_{Scal})	Application Effort (R_{Eff})	Code Rationale (R_{Rat})	Weaknesses, Change Effort (R_{Weak})	Documentation (R_{Doc})
CODESYS [COD22]	o	o	+	o	-	+	-	-	+	n.a.	o	o	o
Beckhoff [Bec21]	o	o	+	o	-	+	o	-	+	n.a.	o	o	o
<i>EcoStruxure Control Engineering</i> [SE22b]	+	o	+	o	-	+	-	-	+	n.a.	o	o	o
Schneider Electric [SE22a]	o	o	+	o	-	+	o	-	+	n.a.	o	o	+
logi.cals [log22a]	o	o	+	o	-	+	-	-	+	n.a.	o	o	o
Siemens [Sie20a; Sie20b]	o	o	+	o	-	+	-	-	+	n.a.	o	o	o
ABB [ABB21]	o	o	+	o	-	+	o	o	+	n.a.	o	o	o

4.2. Code Analysis Procedures for the Quality Assessment of Industrial Software

Apart from individual means supporting the static code analysis of control software or the calculation of software metrics, procedures for software quality assessment are available and will be presented below. A summary is provided in Table 6.

Code inspections or review processes have been in use for decades to reduce errors in software development, e.g., the manual code inspection approach described by Fagan [Fag76]. The inspection is performed to identify errors in the software design or the code (R_{Goal}). Fagan describes four stakeholders involved and required in the inspection process, i.e., a moderator, a designer, a coder (the software developer) and a tester, and estimates an effort of 90 to 100 people hours for the inspection (R_{Use}). The inspection process consists of five steps, which include an introduction to the design decisions in the software (R_{DD}) and manual comprehension and assessment of the software (R_{Rat}). Moreover, the inspection, which is supported by a list of common errors to look for, should be included in the workflow with the same priority as day-to-day tasks (R_{Work}). Every error

detected during the inspection is classified and documented, including its severity (R_{Doc}). In a re-work step, the errors are corrected using the report (R_{Weak}). Fagan rates the training of software developers to identify errors during the inspection as essential.

Since manual code reviews are time-consuming, it is recommended to apply static code analysis in the review process [CW07]. Chess and West introduce a four-phase code review cycle to identify security violations in GPL software (R_{PLC} , R_{Goal}). It consists of setting an analysis goal, running an analysis tool, manually reviewing the tool's results and correcting identified errors. For identifying the analysis goal (targeted security violations), known software risks are assessed and previously discovered errors are consulted (R_{PP}). However, the authors state the need for high-level guidance to prioritize the developers' potential code review targets (R_{Use}). It needs to be ensured that reviewers comprehend the code to be reviewed, including its high-level design (R_{DD}). Before running the tool-based analysis, customized rules should be defined, including company-specific guidelines, to detect errors that are specific to the analyzed program (R_{Sec}). In addition, the tool should be configured in terms of prioritization of reported errors and different means of documentation are recommended, depending on the targeted goal (R_{Doc}). Subsequently, the software developer, who has received training (R_{Use}), or a code reviewer, who collaborates with the developer, manually assesses the identified errors by the tool (R_{Rat}), which potentially provides hints to disadvantageous design decisions (R_{Weak}). The code review can be integrated at different stages of the development workflow (R_{Work}) and, if being applied continuously to real-world software projects (R_{Scal}), the effort for the code reviews of an individual project or similar programs decreases (R_{Eff}).

For measuring the internal software quality, Plösch et al. developed the *Evaluation Method for Internal Software Quality (EMISQ)* to guide the software assessment process and make software quality measurable (R_{Goal}) [Plö⁺08]. *EMISQ* is an expert-based procedure (R_{Use}), which combines manual code reviews and tool-based analysis (R_{DD} , R_{Rat}). It is focused on quality attributes linked to software metrics, which are aggregated to an overall quality model. This quality model is project-specific and needs to be defined before performing the quality assessment. The authors estimate that *EMISQ* "is too heavy-weight to constantly monitor the quality of source code in ongoing projects" [Plö⁺08]. Based on *EMISQ*, Plösch et al. derived the *Code Quality Monitoring Method (CQMM)* for continuous quality monitoring during the development, either as a benchmark-based or as a trend-based assessment [Plö⁺10].

Moreover, based on *EMISQ*, Samarthyam et al. propose the goal-driven and context-aware *Method for Intensive Design Assessments (MIDAS)* for the quality assessment of industrial software [Sam⁺13]. It was evaluated with three Siemens internal software projects from different applica-

tion sectors implemented in C# (R_{PLC} , R_{Sec}) and targeting different analysis goals (R_{Goal}). For enabling the systematic application of analysis tools, a three view-model is proposed to map quality attributes to design violations under consideration of project-specific constraints (R_{Sec}). The model serves as a basis for deriving recommendations for action on how to address the identified violations (R_{Weak}). However, it requires specifying and tailoring a quality model before starting the assessment, which is usually not available in companies programming control software. As a preparation step, a brainstorming session is held with software developers to identify technical issues faced by the project team (R_{PP}) to define an analysis goal. The analysis itself is performed by external experts (R_{Use}), who select suitable code analysis tools and assess the identified violations and errors in a manual code review (R_{Rat}). If required, they collaborate with the developers to consider project-, framework- or domain-specific constraints (R_{Sec} , R_{DD}). Apart from the tool's documentation, the experts create a report, which includes their manual review findings and derived refactoring suggestions for different stakeholders, e.g., a management summary and details for the software developers (R_{Doc}). If projects seem too large for feasible design analysis, the authors suggest focusing on relevant sub-systems or modules which are especially critical or suitable for analysis (R_{Scal}). Since templates from *EMISQ* are used at various stages, it is assumed that the application effort decreases with multiple applications despite the manual tasks performed by the external experts (R_{Eff}). Based on the insights gained from the assessment, the expert may suggest tools and techniques for regular use in the development process (R_{Work}). Overall, the background and experience of involved experts are essential for the procedure's successful application (R_{Use}).

Another method for assessing and improving software quality based on *EMISQ* is the *Structured Code Quality Assessment Method (SCQAM)* proposed by Gupta et al. [Gup⁺14]. *SCQAM* has been applied to eleven industry-sized projects from different application sectors programmed with GPL (R_{PLC} , R_{Scal}), but sector-specific boundary conditions are not targeted (R_{Sec}). The authors propose a combination of code analysis tools and manual assessment by external experts (R_{Rat} , R_{Use}). For enabling manual analysis despite large project sizes, components with known pain points, e.g., being instable or error-prone, are selected prior to starting the analysis (R_{PP}) with the aim of assessing conformance to internal programming guidelines (R_{DD}) and identifying issues with a negative impact on code quality. For this purpose, analysis criteria for the targeted goal are defined in the preparation step (R_{Goal}). Selected analysis tools are configured before the analysis. Thus, if similar software projects are targeted in future assessments, the application effort is expected to be lowered (R_{Eff}). Experts gather the tool findings in a tabular template, classified into eleven categories, to report the top 20 to 40 weaknesses and support actions to avoid them (R_{Doc} , R_{Weak}). In future work, it is intended to train one software developer per group as an expert for applying the assessment procedure independently (R_{Work} , R_{Use}).

Dorninger and Ziebermayr address the application of static code analysis for the quality assurance of control software as part of a continuous integration process (R_{Goal}) [DZ21]. The authors refer to a PLC code analysis platform applicable to software from different vendors (R_{PLC}), e.g., KEBA, CODESYS 2 and TwinCAT 2. The analysis platform contains language-specific parsers and generates an AST on which CFG, data flow and call graphs are built. The gained analysis results about identified weaknesses (R_{Weak}) are provided textually or as HTML, can be integrated into SonarQube and are usable to generate documentation of the code (R_{Doc}). With a rule framework, conformance with PLCopen guidelines can be checked and company- or application sector-specific rules can be defined (R_{Sec}). Additionally, checking naming conventions, calculating complexity metrics or identifying code smells, e.g., large proportions of commented out code, is possible (R_{DD} , R_{Goal}). However, the identification of pain points is not targeted (R_{PP}). While the authors suggest integrating the analysis tool into the development environment (R_{Use} , R_{Work}), no further details about the framework are provided. Moreover, the authors do not mention if their tool was applied to industry-sized control software (R_{Scal}).

In the application sector of packaging machinery (R_{Sec}), Neumann et al. developed an approach to systematically evaluate strengths and weaknesses in company-specific control software architectures, including underlying architectural design decisions (R_{Goal}) [Neu⁺22]. Based on industrial case studies, the authors defined different architectural views on control software, which are described with morphological boxes. For the evaluation, a template for documenting and analyzing architectural design decisions, including drivers, consequences and connections to the actual control code, was developed (R_{Doc}). The documentation forms the basis for identifying optimization potentials in the software architecture and for assessing influences between planned adaptations and existing design decisions. A template based on former research was derived (R_{Weak}) to formulate recommendations for actions [VFN20]. Neumann et al. conducted guided interviews with three machine manufacturing companies using PLCs from the same platform supplier (R_{Pro} , R_{PLC}) and, subsequently, manually performed the architecture analysis (R_{Scal} , R_{Rat}). Different design decisions from the companies were documented and their strengths and weaknesses under consideration of company-specific boundary conditions derived (R_{DD}). Future integration of the approach into the development workflow is targeted (R_{Work}), which would enable the application of the procedure by the software developers. Currently, external experts are required (R_{Use}). Familiarization with the templates and using documentation from previous analyses are expected to reduce the effort in case of multiple applications to the same or similar software projects in a company (R_{Eff}).

With the aim to develop an objective, quantitative classification to guide control software developers in selecting a suitable design methodology for their software (R_{Use} , R_{Goal} , R_{Work}), Mejia et al. assess control software developed according to the GEMMA-GRAFCET (GG) methodology

[MGB22]. This methodology is a recent approach for standardizing the vocabulary and management of operation modes in control software and, in combination with a design pattern, enables code generation [BG21]. Mejia et al. developed and applied a benchmark workflow to assess control software obtained with the GG-methodology compared to a statechart-based methodology for ST code generation using metrics (R_{Goal}) [MGB22]. Both software projects for controlling a lab-sized test bench (R_{Scal}) were programmed in CODESYS (R_{PLC}). Four software metrics targeting the code structure and two metrics related to the real-time behavior of the software were calculated using CODESYS *Static Analysis*, CODESYS *Test Manager* and manual calculation. The computed metric values for both methodologies are visualized in radar and bar charts (R_{Doc}). Moreover, they are manually assessed under consideration of specific design decisions in the compared methodologies (R_{Rat} , R_{DD}) to derive the benefits and drawbacks of the methodologies (R_{Weak}).

Table 6: Related quality assessment approaches evaluated with respect to the derived requirements.

Approach	Platform Independence (R_{PLC})	aPS as Product (R_{Pro})	User (R_{Use})	Application Sector (R_{Sec})	Pain Points (R_{PP})	Workflow Integration (R_{Work})	Design Decision (R_{DD})	Analysis Goal (R_{Goal})	Scalability (R_{Scal})	Application Effort (R_{Eff})	Code Rationale (R_{Rat})	Weaknesses, Change Effort (R_{Weak})	Documentation (R_{Doc})
Fagan [Fag76]	-	n.a.	+	n.a.	n.a.	+	+	o	+	o	+	o	+
Chess, West [CW07]	-	n.a.	+	o	+	+	+	o	+	+	+	o	+
Plösch et al. [Plö+08]; Samarthyam et al. [Sam+13]	-	n.a.	-	+	o	o	+	+	o	o	+	o	+
Plösch et a. [Plö+10]	-	n.a.	o	o	-	+	o	+	+	+	-	o	+
Gupta et al. [Gup+14]	-	n.a.	-	o	o	o	o	+	+	o	+	o	o
Dorninger and Ziebermayr [DZ21]	+	o	o	+	-	o	o	o	?	n.a.	?	o	o
Neumann et al. [Neu+22]	o	o	-	-	n.a.	o	+	o	+	o	+	+	+
Mejia et al. [MGB22]	o	-	o	n.a.	-	o	+	+	-	n.a.	+	+	o

Overall, available quality assessment procedures contain many valuable elements and concepts, but none of them fulfills all the requirements derived in Section 3. Moreover, approaches from the computer science domain targeting GPL are not applicable in the domain of aPS without changes due to the varying boundary conditions [Jet+13b; Vog+15b].

4.3. Research Gap in Quality Assessment of Control Software

The overview of the related work in the previous Sections shows that none of the analyzed approaches meets all requirements derived for quality assessment of control software and its subsequent improvement. This is primarily due to the lack of possibilities for adapting the first available static code analysis approaches and software metrics in the aPS domain to company-specific or application sector-specific boundary conditions. Furthermore, static code analysis and software metrics are not sufficient without a procedure or method guiding their systematic application [Plö+08] since control software developers lack experience with their application and interpretation of results. In addition, available analysis and quality assessment procedures often require an external expert or the definition and tailoring of a quality model prior to the assessment and target the identification of weaknesses only. However, in the aPS domain, only a few general programming guidelines and barely any best coding practices are available so far. Thus, procedures based on a quality model are not applicable without adaptation. Furthermore, not only weaknesses but also beneficial design decisions supporting the targeted analysis goal should be documented and considered as a basis for deriving best practices and reusing these beneficial design decisions in future software projects. Overall, the significantly different boundary conditions in the aPS domain prevent all approaches from computer science from being directly applicable. An overview of the analyzed approaches in the state of the art is depicted in Figure 2.

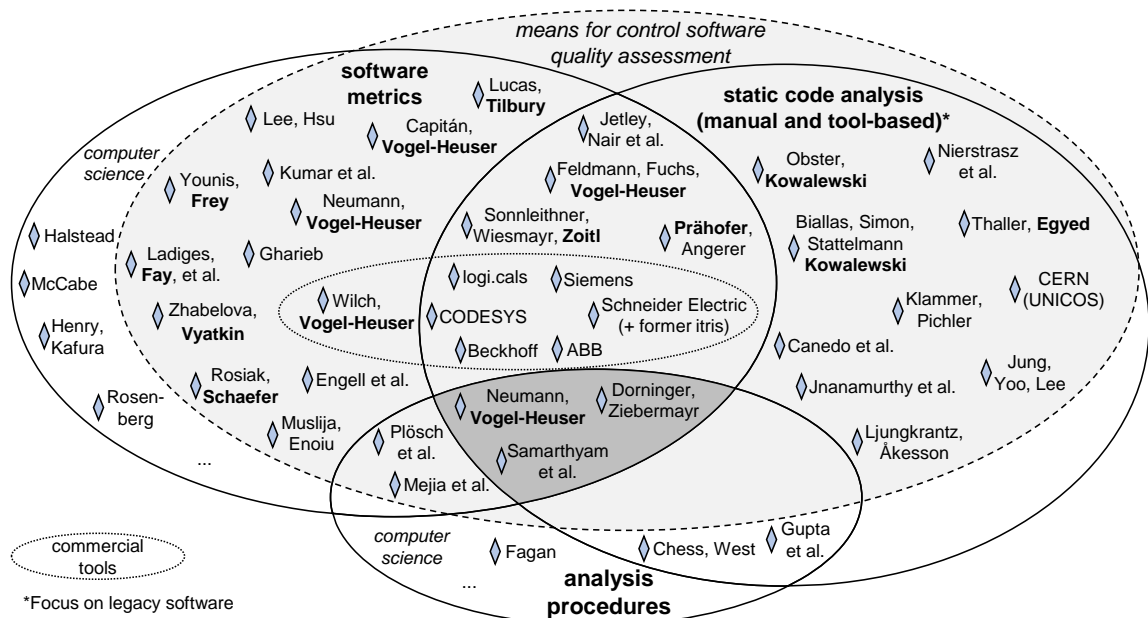


Figure 2: Overview of the research areas regarding means and approaches for PLC software quality assessment, including selected approaches from computer science. The identified research gap is highlighted in dark grey.

Consequently, the research gap that is addressed within this thesis is identified as:

Research gap

So far, to the best of the author's knowledge, there is no systematic quality assessment procedure for IEC 61131-3-based legacy control software using guiding questions, checklists, static code analysis and software metrics at different points of the software development workflow, independent of the development environment. Moreover, assessment approaches with suitable documentation of gained insights and results to enable industrial experts to identify improvement potentials, derive suitable means for addressing and overcoming the identified weaknesses, and capture and reuse beneficial design decisions are not yet available.

The quality assessment procedure developed and presented in this thesis aims at filling the identified research gap.

5. Procedure for Quality Assessment of Legacy Control Software with Static Code Analysis

This Chapter describes the proposed procedure for quality assessment of PLC legacy software with guiding questions, checklists and static code analysis. First, in Section 5.1, insights gained during various pre-studies used to develop the quality assessment procedure are summarized. Furthermore, a short overview of the proposed procedure is provided. Subsequently, details of the procedure's four concept steps are presented in Section 5.2, utilizing an industrial case study as an application example.

The proposed procedure for software quality assessment extends the metric-based analysis procedure for reuse and modularity assessment of selected control software modules by the author published in [Fis⁺21a].

5.1. Quality Assessment Procedure for Legacy Control Software

The proposed quality assessment procedure considers the insights from pre-studies, including questionnaires and expert interviews. Lessons learned from these, which should be considered during control software assessment, are illustrated below. Afterward, an overview of the proposed quality assessment procedure is presented.

5.1.1. Pre-considerations Regarding the Quality Assessment of Control Software

In general, software quality assessment with static code analysis and metrics is not a rigid method with universal rules suitable for every type of software in every context. Instead, apart from the control code itself, also company-specific boundary conditions need to be considered to assess control software, as they have a strong influence on the software and its development process [Neu⁺20c]. Furthermore, depending on the characteristics of the system under control and the educational background knowledge of the software developers, different reuse strategies are more or less beneficial [Fis⁺21c]. Thus, only if a context-sensitive assessment of the control software is performed, valuable recommendations for enhancing the software quality can be derived from the results. Insights gained during pre-studies are summarized in the following, focusing on control software complexity, dependencies within the control software, consideration of company-specific guidelines and constraints and the suitability of different reuse strategies.

The control software architecture, e.g., the software's components and their interconnections, plays an essential role in software quality as it enables, for example, planned reuse. Thus, the

software architecture should be considered during the quality assessment of control software. Regarding software components, i.e., POUs, their complexity is a relevant quality attribute: complexity potentially reduces a POU’s comprehensibility, reusability and maintainability, which leads to an increased effort during its development, testing and maintenance [KS13; Ram+85; YF07]. Various software metrics exist to quantify software complexity from different viewpoints (cf. [Fis+21b] for an overview). However, there is no universally agreed-upon value as an upper limit for the complexity of PLC control software, which indicates that the software is too complex. Similarly, ISO/IEC 25023 does not specify value ranges of software quality measures to rated levels/grades as this correlation depends on particular boundary conditions [ISO25023]. Moreover, practitioners highlight the challenge of interpreting the meaning of specific complexity values.

To overcome this challenge, a recent approach by the author proposes to *assess the software complexity of a POU in the context of other POUs* implemented according to the same programming style and implementing the same functionality [Fis+21b]. Thereby, metrics targeting different complexity classes, as described in [LC94], are combined to an overall complexity value, including the complexity composition (cf. Figure 3). Subsequently, a selected set of POUs can be analyzed to identify outliers, i.e., POUs with a single or an overall complexity value that is strikingly higher than the rest of the set.

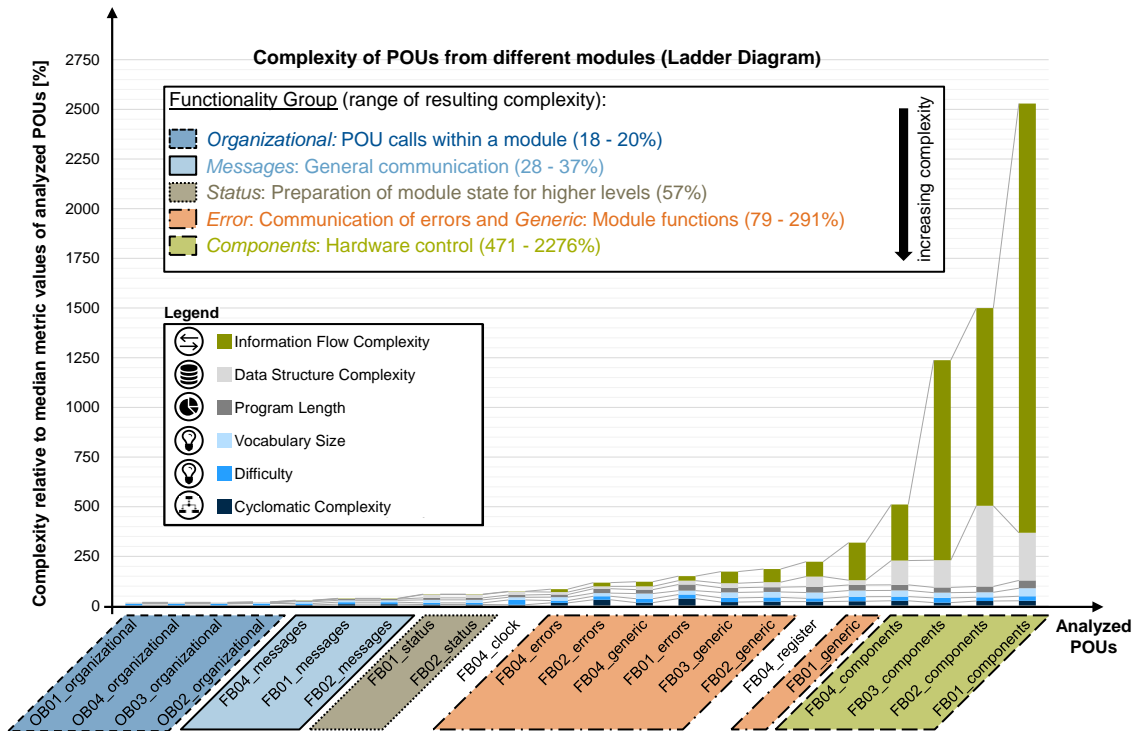


Figure 3: Overall complexity of POUs from an industrial PLC project, annotated according to their functionality (implemented in LD, PLC project taken from Case Study C (cf. Table 2, p. 17); graphically adapted from [Fis+21b]).

A detailed, manual evaluation of the proposed complexity metric was conducted with 50 POUs, programmed in Siemens development environments, from two global market-leading German special purpose machine and plant manufacturers (one further considered in *Case Study C*). It showed that POUs implementing the same functionality have similar values regarding their *overall complexity*. For example, when ordering all analyzed POUs implemented in LD of a project from *Case Study C* by increasing complexity values, they are at the same time grouped by their functionalities. POUs with the lowest complexity values implement organizational functionalities in the evaluated samples and the highest complexity values are reached by POUs implementing hardware component control (cf. Figure 3). Moreover, the performed analysis showed that POUs implementing the same functionality are similar in the value distribution of the individual complexity metrics, including their dominant complexity metric class (cf. [Fis⁺21b] for details). These findings illustrate that a universal complexity threshold is not feasible. Instead, POUs implementing the same functionality and programmed with the same guidelines should be compared to identify outliers concerning their complexity for their quality assessment. However, complexity is only one factor that affects software quality and needs to be combined with other factors to evaluate the overall quality of a software project.

Secondly, *dependencies between POUs* are targeted, focusing on indirect dependencies as a potentially disruptive factor for a planned reuse of control software. Static code analysis at various companies conducted by the research group of Vogel-Heuser (including the author) has shown that there is no ideal way to implement information exchange and, thus, dependencies between POUs. This is because dependencies are influenced by multiple factors: the implemented functionality, the amount of information to be shared, the time requirements regarding the data distribution and the number of POUs needing the shared information, to state a few. The analysis of a highly mature software architecture, which supports planned reuse and a clear separation of standardizable and application-specific software parts, shows that indirect data exchange is not necessarily hampering the reuse potential of the POUs included in a software project. The prerequisite is that the entire data exchange is implemented in conformance with mature programming guidelines, which contain detailed rules regarding the exchange of data [Fis⁺21a]. Consequently, when analyzing the dependencies between POUs in a software project to assess its quality, analysis rules tailored to the programming standard of the control software should be used, as demonstrated in [Fis⁺22b]. Apart from individual recommendations, e.g., flag variables in the bit memory should be avoided according to Siemens' programming guidelines (cf. [Sie18], p. 99), barely any universally accepted beneficial or disadvantageous ways of implementing data exchange exist. Consequently, a context-sensitive analysis, taking company-specific programming guidelines into account, is required to assess the considered control software's quality.

Moreover, a recent interview study conducted by the author showed that many companies use *programming guidelines and company- or department-wide naming conventions* [Fis⁺18]. Some even use unique equipment identifiers, which enable linking the automation hardware to the respective control software. Furthermore, in some companies, the symbolic name of a POU indicates its version number or a changelog is included within the POU's comment section to document the POU's version history. These examples show that programming guidelines contain valuable information or indicate available metadata supporting the quality assessment of control software. Thus, company-specific guidelines should be considered but require manual analysis and interpretation (cf. R_{Rat}). In some cases, it is possible to manually derive rules for automatically checking the code regarding conformance to guidelines [Fis⁺18]. Overall, their company-specific character highlights the necessity to analyze and assess control software in its context, including unique boundary conditions and guidelines.

Another essential aspect of control software assessment is that no perfect reuse strategy is applicable in any situation and software. Instead, choosing a suitable reuse strategy depends on different boundary conditions influencing software reuse, which go beyond the software itself. A recent study comparing the applicability of an OO-IEC and a feature-oriented development approach for the planned reuse of variant-rich control software identified *multiple factors influencing the suitability of reuse approaches* [Fis⁺21c]. The two presented concepts are compared concerning software-related factors such as modularity and software characteristics, e.g., the expected amount and scope of variation points. It is identified that the software developers' background, meaning their experience and knowledge, is essential for choosing a suitable reuse strategy. Also, the pre-work required to establish a specific reuse strategy, such as preparing templates or libraries, and the expected benefit, e.g., their reusability, plays an important role [Fis⁺21c]. Due to the high number of influencing factors, the study concludes that there is no perfect strategy for a planned reuse of control software that fits every application scenario in all companies. Consequently, quality assessment of PLC control software must be conducted in a context-aware analysis process, including boundary conditions.

A comparison of *reuse strategies for the extra-functional task alarm handling* raises further points to consider when comparing reuse strategies [Vog⁺22a]. These points include assessing general aspects such as the concept's evolvability, adaptability and reusability in a different context. Furthermore, the support of legacy systems and the customer- /application-specific adaptation of standardized parts, i.e., error handling, are compared. Apart from these aspects targeting the software architecture, points involving the software developers are also considered, e.g., ease of use targeting the risk for incomplete or incorrect use of standardized software parts. Additionally, a benchmark of market-leading companies in the aPS domain with mature software development

processes showed that these companies often do not pursue clear strategies for planned reuse. Instead, mixed forms combining templates and library modules are used [VON18].

Prior research results [VO18; Vog⁺17] highlight the *influence of company-specific constraints* on the software development process. Moreover, influences on the software architecture are diverse, as emphasized in [Neu⁺20c]. Apart from software aspects, e.g., functionality distribution in control code, also organizational aspects are crucial. For example, the impact of the development team size on appropriate reuse strategies and their potential benefits are highlighted: universal software modules, which often tend to get complex over time, can be handled by a relatively small software development team, which is agile enough to allow direct communication of all developers with low organizational effort. On the contrary, change management becomes increasingly complex and demanding for larger departments, which can even be spread across different locations or countries. In conclusion, the same reuse strategy (universal software modules) can have varying effects depending on company-specific constraints [Neu⁺20c]. There are no universally agreed-upon standards and the suitability of a reuse strategy depends on software characteristics, organizational aspects, used automation hardware and the application sector.

In summary, conducting a software analysis with strict, rigid rules for quality assessment is neither feasible nor expected to deliver beneficial results. It is not sufficient to analyze the software without its context, i.e., boundary conditions such as the number of software developers, programming guidelines, related automation hardware or variants. Consequently, although the use of static analysis tools is helpful for the analysis to save time, interpretation in the company's context is essential and requires knowledge not available or graspable by tools. Thus, manual effort is still required (R_{Rat}). The assessment procedure presented in the next Section takes these insights into account.

5.1.2. Overview of the Quality Assessment Procedure for IEC 61131-3-based Control Software in an Industrial Context

The proposed procedure targets the goal-oriented quality assessment of existing, often historically grown, legacy control software. In order to be generally applicable to control software in the aPS domain, it is designed in a platform-independent manner, i.e., it is not tailored to the specifics of a selected implementation platform but provides general assessment steps based on the IEC 61131-3 standard (R_{PLC}). The procedure serves as a framework and structured guide for the quality assessment of control software, which is adaptable to the characteristics of the considered product (R_{Pro}), e.g., serial machines, special purpose machines or plants, and also takes application sector-specific boundary conditions into account (R_{Sec}). The procedure's four steps are depicted in Figure 4 and their aims are shortly introduced in the following.

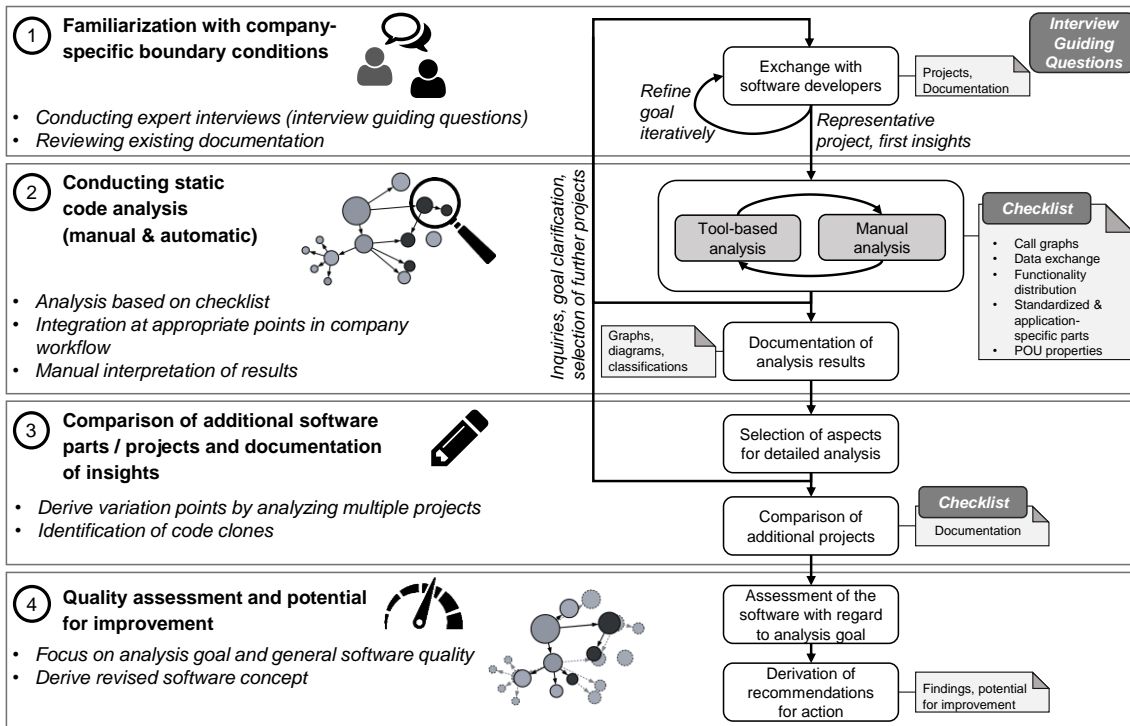


Figure 4: Software assessment procedure for classical IEC 61131-3 control software.

The first step aims to familiarize with the targeted software and the system it controls, including boundary conditions to be considered during the analysis. Thus, guided expert interviews with selected control software developers are conducted and available documentation regarding the control software is gathered. Further, the analysis goal is defined and a representative software project for conducting the first analysis is selected jointly with the software developers. Subsequently, in Step 2, following a checklist, static code analysis of the chosen project is conducted to understand the general structure of the control software. The analysis is performed as a combination of manual and tool-based analysis to cope with the large size of industrial control software projects while considering the semantics behind the tool-based analysis results. The insights gained during this analysis are documented, e.g., in graphs and diagrams, for a subsequent discussion with the software developers in which questions are clarified. If necessary, the chosen analysis goal is refined. Next, additional software projects are selected for analysis in Step 3. In this step, the focus lies on documenting the analysis results tailored to the chosen analysis goal. Using the insights gained from Step 2, the focus is narrowed down to the analysis goal when analyzing further projects. Therefore, unlike in Step 2, not necessarily the entire software projects are analyzed, but specific parts are targeted. Finally, in Step 4, the analysis results are used to assess the control software, identify beneficial and disadvantageous parts and derive recommendations for action to enhance the software's quality.

5.2. Detailed Introduction to the Quality Assessment Procedure

The details of the four procedure steps for quality assessment are introduced in the Sub-sections below, utilizing *Industrial Case Study A* (cf. Table 2 for general information) as application example for better understandability. The case study has been published in [FVF15] and [Vog⁺15a].

5.2.1. Familiarizing with Company-specific Boundary Conditions (Step 1)

In preparation for the context-aware quality assessment with static code analysis, familiarization with the considered aPS, its functionality, available documentation and known pain points is required. Further, it is essential to set an analysis goal to ensure that the analysis results are relevant and beneficial for the company, as highlighted in [Sam⁺13]. Means of static code analysis are available in a great variety and address different quality attributes (cf. Section 4.1, p. 26). Applying these means in an unstructured manner and without a pre-defined analysis goal impedes the usefulness of the analysis results. At the same time, analyzing too many quality attributes at once is not feasible either, since the final assessment requires domain knowledge, meaning input from the software developers familiar with the controlled process and its boundary conditions, and time-consuming, manual interpretation. An overview of the activities performed in this analysis step is depicted in Figure 5, which will be introduced utilizing *Case Study A* in the following.

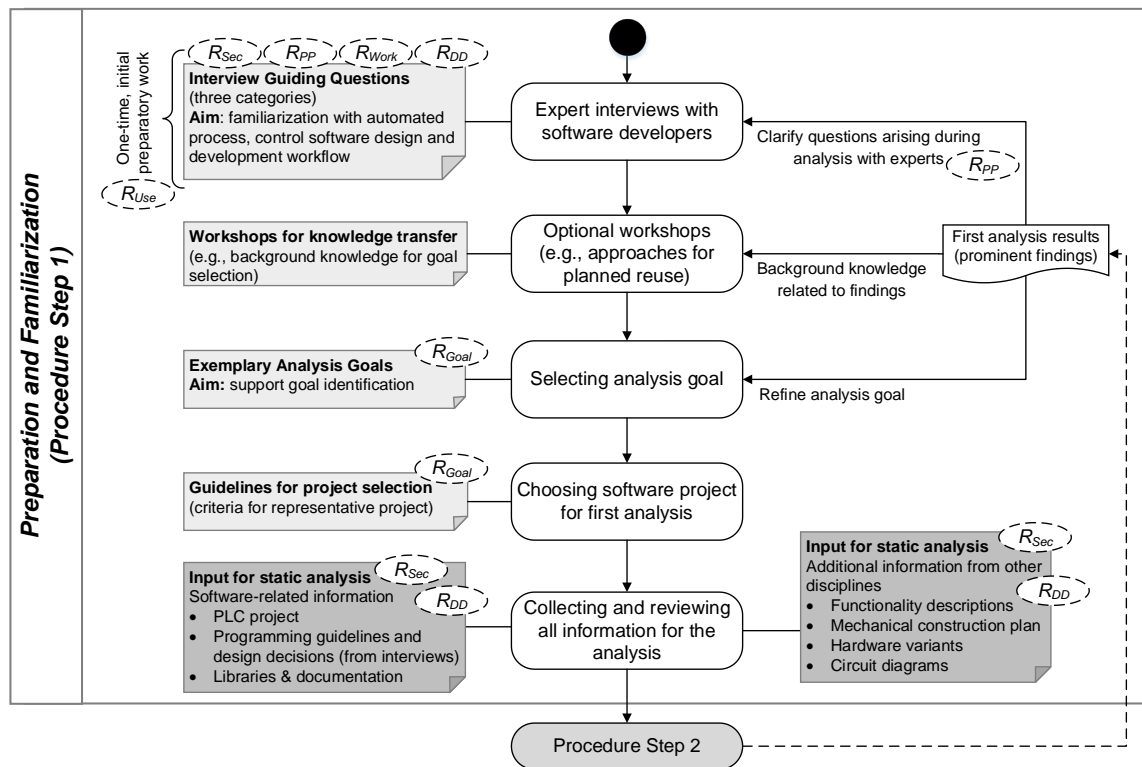


Figure 5: Details of Step 1 (Preparation and Familiarization) of the quality assessment procedure.

Case Study A was conducted in an internationally market-leading plant manufacturing company in the woodworking application sector. The company manufactures complete production plants, including discrete intralogistics processes like the analyzed plant part *warehouse*. The following details describe the software and its development process at the time the case study was conducted.

Expert interviews with software developers

As depicted in Figure 5, initially, an expert interview, ideally with software developers for the considered machine or plant, is conducted. The aim of the interviews is the familiarization with the considered aPS, the software development process, including involved stakeholders and their tasks, and the control software itself. Critical points regarding the control, e.g., parts with hard real-time restrictions or dependencies to systems outside the PLC, should be identified since their implementation usually contains undocumented expert knowledge and might be rated as disadvantageous if the reason for the implementation is not known.

Case Study A was conducted on-site at the company over several weeks, which enabled the exchange with the software developer of the warehouse on short notice. During the first interview, the software developer introduced the warehouse’s *general functionality*, including its *process sequence* and position within the plant. The warehouse, illustrated in Figure 6, is located between the cooling and stacking station (C&S) and the sanding line (SL). After a pressing process, produced particle boards are stacked at the C&S station’s stacking places, where they are collected by the storage car and transported to their storage place to cool down and harden. After hardening, the storage car transports the stacks to the SL for subsequent processing. The arrows in Figure 6 indicate the transport direction of the stacks. [FVF15]

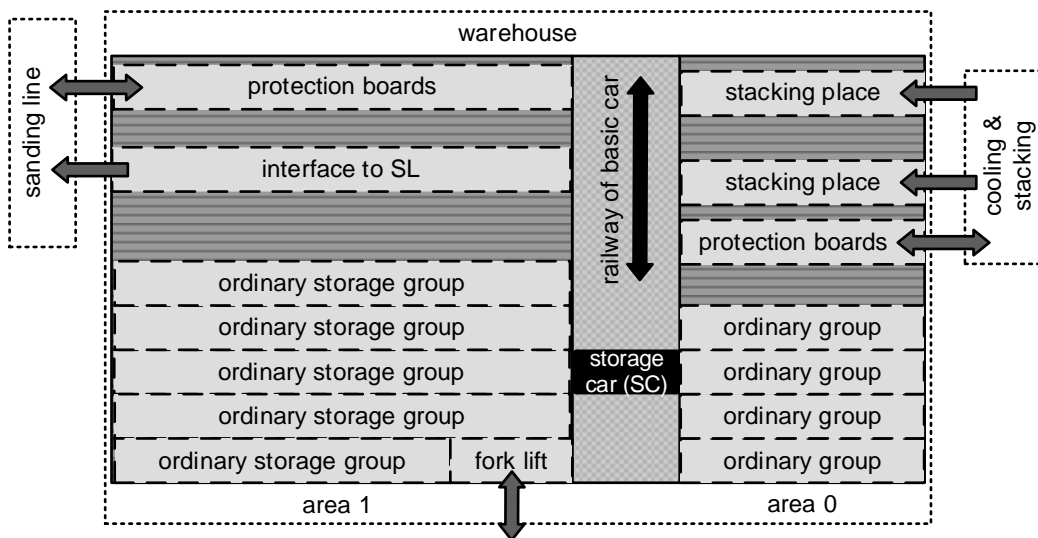


Figure 6: Exemplary layout of a warehouse (graphically adapted from [FVF15]).

Generally, a warehouse in *Case Study A* consists of a variable number of storage areas, which are located left and right of the storage car's railway. Figure 6 illustrates an exemplary warehouse with two areas. The areas are further divided into storage groups. The storage car consists of a basic car (BC) moving along the railway and a variable number of satellite cars located on the BC. The satellite cars move along the groups to lift or set down a stack. The number of storage cars is another variation point and limits the number of areas. Optionally, reusable protection boards can be utilized at the bottom of the stacks to prevent damage to the manufactured boards. These are stacked in two particular, optional storage groups – one at the C&S and one at the SL. [FVF15]

During the interview, the software developer mentioned *multiple variation points*. These are essential since their influence on the control software needs to be reflected in the applied reuse strategy. Moreover, the warehouse layout depends on customer requirements. Consequently, its composition varies considerably depending on individual orders, and the warehouse control software needs to be modified accordingly. This is confirmed by the identified special *boundary conditions* of the warehouse control software: it highly depends on the warehouse dimensions since these are used to position the storage car. Further, the communication between the warehouse PLC and other PLCs, e.g., the control of the storage car, is essential and not adaptable. Concluding, high variability is expected, which causes a high amount of software variants to be managed.

The control software is written by a single software developer utilizing the *unplanned reuse* strategy of *copy, paste and modify*. When starting the software development for a new warehouse, the software developer chooses the software of a completed project as a basis. Since no variant or version management is established, selecting a suitable project is based solely on the practical knowledge of the software developer. While a few standardized functions are used in the software development process, only library elements delivered from the PLC supplier are utilized and no company-specific module library exists. [FVF15; Vog⁺15a]

The company uses rather rough *programming guidelines*, which contain general software requirement specifications like interlocking conditions or the general structure of certain POU types. However, a company-wide numbering system, i.e., unique equipment identifiers in the style of [IEC81346], is used throughout the development in all involved disciplines and results in *naming conventions* for POUs. The control software is mainly programmed in LD, which is often a requirement by the customers. Additionally, IL is used for some elaborate calculations. [Vog⁺15a]

The *overall structure* of the control software and *design decisions* were shortly discussed with the software developer. Generally, the individual control software variants are documented in various forms, e.g., functional design descriptions, unique equipment identifiers and naming conventions

for POU_s, comments in POU headers and network titles. However, the influence of the identified variation points (from a functional and a hardware point of view) on the individual software parts is not documented in detail. Consequently, detailed information on the design requires static code analysis of an exemplary program in Step 2. In the discussion, five main implemented functionalities were identified to which the POU_s are assigned. These are general aspects (alarms, operation modes), data exchange between plant parts, storage car control, connection to the warehouse management system (WMS), and transferring data to the HMI. The mainly global variables used to fulfill those functionalities are organized in structures within DBs. Apart from the *communication to the HMI*, the communication between the storage car and the WMS is essential and should not be changed. The warehouse can be operated in three different *operation modes*, i.e., an automatic mode, a manual mode and a manual-unlocked mode, e.g., for the start-up. [FVF15; Vog⁺15a]

The software developer responsible for the warehouse is aware of the *drawbacks and risks* of introducing errors when applying the current reuse strategy *copy, paste and modify*. Furthermore, the lack of detailed documentation on the influence of different variation points on the control software is considered a *challenge in the development process* since, at the time of the analysis, only one software developer was familiar with the control software, which represented a significant risk for the company. Accordingly, it was decided to train a second developer and document the variation points to enhance planned reuse.

Overall, as can be seen from the insights gained above, interviewing the software developer enables a rough overview of the controlled aPS and the technical process as well as involved stakeholders and information such as naming conventions that help understand the software. Without this knowledge, the software analysis for quality assessment cannot be performed in a meaningful way with reasonable effort. On the one hand, familiarizing and understanding the unknown control software would take significantly longer, as understanding the rationale behind existing code is time-consuming and difficult [LVD06; MML15; Ste00], especially without background information on the controlled system. On the other hand, without talking to the domain experts, i.e., the software developers, application sector- or company-specific boundary conditions are not known and, thus, cannot be considered (R_{Sec}). However, they have a high impact on design decisions in the control software (R_{DD}). A set of interview guiding questions was developed to simplify the familiarization with the software to be analyzed (cf. Appendix A.1 for a detailed question list).

The questions were derived from former case studies, which illustrate the importance of architectural levels [Vog⁺15a], current challenges of reuse, variant and version management [Fis⁺18] and challenges and industrial approaches for implementing extra-functional software parts, e.g., error

handling [Vog⁺16], especially in variant-rich systems [Vog⁺22a]. Further, they highlight the relevance of considering a company's programming guidelines and development workflow during the software quality assessment [Fis⁺21a; Fis⁺22b]. Also, information from other disciplines, e.g., the mechanical layout plans and their variants, which influence the control software [FVF15], organizational aspects [Neu⁺20c] and the software developers' background knowledge [Fis⁺21c] were identified as influencing factors for control software. Finally, multiple questionnaire studies of the research group of Vogel-Heuser were reviewed. They are aimed to quantify and assess different parts of the control software development process, applied reuse strategies and the commissioning process for a benchmark between different companies [VO18; Vog⁺17; Vog⁺21c; VON18]. The gathered questions are organized into three blocks, i.e., questions regarding the automated system, questions targeting organizational aspects and questions regarding the control software.

The questions support identifying critical points in the application and making conscious design decisions, which are often not documented, explicit for the analysis (R_{DD}). A block of questions focuses on the control software itself, e.g., naming conventions. Especially numbering systems used across different disciplines support the analysis, as they connect software implementation parts to the controlled automation hardware. Additional questions target recurring challenges faced during the development or commissioning to identify pain points as potential points for improvements and, thus, derive the analysis goal from them (R_{PP}). The questions are phrased as general as possible to support the definition of a helpful analysis goal depending on the identified pain points and challenges (R_{Goal}). Finally, several questions target the software development workflow, which is essential to determine an appropriate point for conducting the static analysis in Step 2 (R_{Work}).

In conclusion, the main aim of the interview guiding questions is to enable software developers, after training, to perform the static code analysis themselves in a goal-oriented, systematic process (R_{Use}). They support identifying potentially helpful information for the analysis, including documents from different disciplines. Moreover, the questions target making implicit design decisions about the software architecture explicit for taking them into account during the analysis (R_{DD}). The familiarization step can be omitted or significantly reduced when using the procedure a second time to assess the same or similar control software from the same company (R_{Eff}).

Optional workshops for knowledge transfer

After the expert interviews, optional workshops regarding selected topics can be conducted throughout the procedure, for example, to support selecting a suitable project to start the analysis in Step 2. Another possible focus targets the identified, recurring challenges in the company's control software development to provide background knowledge about potential solutions for these.

During the first-time application of the assessment procedure in a company, there are multiple reasons for conducting a workshop with the company's software developers at different procedure steps. A group of targets is to clarify questions that arise during the quality assessment procedure about the control software and hinder performing the next step. For example, after the initial analysis of the first project in Step 2, a workshop can support *clarifying questions* that arose during the documentation of the analysis results. Second, focusing on *knowledge transfer* to provide the developers with theoretical background knowledge and, thus, enable them to perform the software quality assessment independently in the second procedure application (R_{Use}). An example of background knowledge potentially interesting for the software developer in *Case Study A* is available means for a planned reuse of variant-rich control software, including requirements and limitations.

Selection of the analysis goal

Based on the insights gained in the expert interview, the goal of the quality assessment was defined. Jointly with the software developer of the warehouse, the unplanned reuse with *copy, paste and modify* and the lack of documentation on detailed design decisions and influences of variation points were rated as the main challenges. More precisely, multiple variation points in the mechanical components of the warehouse were expected to have a strong influence on the control software. Consequently, to avoid the disadvantages of applying *copy, paste and modify*, the analysis goal "*enhance planned reuse of variant-rich control software*" was selected (cf. Table 7, Goal 2).

With the goal being defined, some coarse requirements regarding the next procedure steps can be set: During the analysis and familiarization with the control software in Step 2, aspects like its structure, functionality distribution and data exchange need to be considered and documented. In Step 3, the variation points of the warehouse should be determined jointly with the software developer, if required, including other experts from the company. Furthermore, control software variants of different warehouses need to be analyzed to identify the dependencies between the control software and the mechanical components of the warehouse. Finally, in Step 4, suitable means for planned reuse must be chosen based on the insights gained from the static code analysis.

Generally, to support the definition of an analysis goal, exemplary goals are provided for assistance (cf. Table 7). The suitability of an analysis goal always depends on the insights gained during the conducted interviews. Moreover, the same goal can be used when analyzing software of different maturity levels. For example, *Goal 1* proposes a general assessment of software modularity, which is a prerequisite for software reuse. This goal can be used in historically grown control software to analyze if any modularization strategy is already applied and to identify simple, low-effort changes to increase the control software's modularity. The goal is also suitable for assessing mature control software developed using planned reuse strategies. Points for improvement can be

identified even in mature software, as demonstrated in a questionnaire study with market-leading German aPS manufacturers [Vog⁺17]. Some goals require a minimum level of maturity of the development process, e.g., in the case of Goal 6, programming guidelines must be available, which vary significantly in their level of detail from company to company. However, many generic goals, e.g., code clone identification to enhance reuse (*Goal 3*) or documentation of the current software architecture and its primary design decisions, are suitable for legacy software without pre-defined software architectures. Regardless of the software maturity, the definition of an analysis goal is essential to prevent the code analysis from becoming an unorganized process. Finally, the analysis goals are not entirely independent; e.g., *Goal 3* and *Goal 6* can also be relevant when assessing *Goal 2*. Thus, targeting a combination of prioritized goals is also possible.

Table 7: Exemplary analysis goals for static code analysis of legacy control software (not independent from each other).

Number	Analysis Goal	Details regarding the aim
Goal 1	Assessment of software modularity	Modularity is a prerequisite for planned reuse and high-quality software. Distribution of software projects into reusable parts and their interfaces are targeted.
Goal 2	Increasing planned reuse in variant-rich software	Documentation of variable and unchangeable software parts to separate their implementation and potentially derive library modules of common parts or choose suitable variant management and reuse strategies.
Goal 3	Analysis of code duplicates	Identification of code duplicated on different granularity levels and within a single project and/or across multiple projects.
Goal 4	Analysis of software evolution during the development	Conduction of conformance checks to detect violations of programming guidelines in the version history of a software project. This goal aims to identify disadvantageous parts in application-independent control software intended for reuse in different projects, e.g., project or module templates, which are combined with application-specific parts.
Goal 5	Identifying the reuse potential in application-specific parts	Analysis of details of application-specific parts to identify commonalities in structure or source code as a basis for their encapsulation and planned reuse (to reduce the application-specific parts to a minimum).
Goal 6	Assessment of programming guidelines	Targeting, for example, library modules and application-specific software parts (functionality distribution between them, their interactions/communication/integration of standard and application-specific/customer-specific parts).
Goal 7	Preparation for refactoring/green-field development	Analysis regarding disadvantageous design decisions known from the software's lifecycle, e.g., challenges during commissioning, reuse or adaptations to customer requirements, maintainability, before refactoring a project. The aim is to integrate the analysis results into the conceptual design phase of the new software architecture (avoid repeating/keeping disadvantageous design decisions).

This list of exemplary analysis goals was derived from challenges identified in previous case studies and questionnaire studies, e.g., [MJG11a; Vog⁺15b], which highlight requirements for a planned reuse of high-quality control software. It supports the software developer in choosing a

reasonable goal by considering the identified pain points and challenges. Depending on the goal, the integration into the company workflow, timing and regularity of the analysis are influenced. These factors might impact selecting a suitable project for the first analysis.

Choice of the software project/project parts to be analyzed in Step 2

After the analysis goal has been defined, a suitable project should be selected for the first application of static code analysis. Since the selected analysis goal in *Case Study A* targets the planned reuse of variant-rich control software, the projects chosen for analysis should include variation points. Further, they should be complete, i.e., belonging to warehouses that are already operating or shortly before commissioning. Otherwise, the comparison of incomplete control software variants will incorrectly detect variation points at software parts that are not yet fully programmed. For familiarization with the warehouse and its control software, the software developer suggested a recently commissioned warehouse, which is well-known to him. In addition, in the expert's view, it is representative of the general warehouse functionalities and is relatively simple. For example, the selected warehouse contains only one storage car.

The procedure supports this step with an additional part of the guiding questions containing hints for selecting a representative project (cf. Appendix A.1, p. 187). For example, the first project should be representative of the aPS under consideration and the pain points addressed in the analysis. Both positive and negative examples help familiarize with the control software and comprehend its structure. Moreover, the functionality of the aPS being controlled by the selected software projects and, if available, associated programming guidelines should be known. Choosing a well-documented (or at least well-known) project for the first analysis, including its functional description, a mechanical layout plan or unique customer requirements, lowers the effort of the initial analysis as these documents can be consulted rather than extracting the knowledge from the software itself. Especially with legacy software, the source code is frequently the only documentation available, which makes it challenging to comprehend the implemented functionality [Kir⁺16]. Another aspect to consider is that the applied reuse strategies and associated templates or module libraries and the development process should be known. If questions arise during the analysis, it is beneficial to know which developers were involved in the software programming to include them in the clarification process.

Collection and review of additional information

For the initial analysis in *Case Study A*, the software developer provided two documents from other disciplines: the mechanical layout plan of the selected warehouse variant and the corresponding motor-valve-limit switch list (MVL-list). The MVL-list contains all automation hardware components used in the warehouse, including their unique equipment identifiers. Thus, it helps connect

the mechanical layout plan and the naming conventions in the control software to the respective hardware elements. During the expert interview, a significant influence of the warehouse layout on the control software was identified, e.g., the warehouse dimensions are used within the control software. Thus, both documents are expected to ease the analysis of the first project in Step 2.

Apart from documents from other disciplines, information about the control software is beneficial if available. For example, if the control software to be analyzed follows a pre-defined software architecture, it is helpful to gather documentation on the architecture concept. On the one hand, it eases the analysis of the first project and understanding the control software in analysis Step 2. On the other hand, it is a prerequisite for analysis goals such as checking the conformance to existing guidelines. However, it is not necessary to define the architectural levels in advance to enable the quality assessment of control software. Often the software architecture is not known or documented before conducting the first analysis, which is directly targeted in Step 2.

In summary, the aim of Step 1 is the preparation of the static code analysis by familiarizing with the controlled system and its control software and gathering as much information, which is potentially helpful for the analysis, as possible. To support the software developer in conducting this procedure step, respective interview guiding questions, exemplary analysis goals and hints for selecting a project for initial analysis are provided. It is important to note that not always all questions need to be answered and not necessarily all listed information is available. Furthermore, information usable for the analysis and discussions with experts is not limited to the mentioned aspects; they only provide a starting point for the systematic preparation of the static code analysis.

5.2.2. Static Code Analysis of a PLC Software Excerpt or Single Project (Step 2)

With the analysis goal set and the first project selected, the static analysis is prepared and performed in Step 2. The activities during this step are depicted in Figure 7. A checklist with aspects to be considered during the static analysis is provided as a starting point to support the selection of suitable analysis means. As suggested in [Sam⁺13], the analysis is conducted as a combination of manual and tool-based analysis since both methods complement each other well: due to the large size of industrial control software projects, a pure manual analysis is not feasible. However, manual analysis is essential to comprehend the intention of the control software parts and to take the semantics behind the tool-based analysis results into account.

The analysis results are visualized and documented in different formats and insights gained are summarized. If required, discussions with software developers about initial analysis results are conducted to comprehend software parts, which require their domain knowledge. These discussions are scheduled on-demand to clarify open questions about the documented analysis results.

As depicted in Figure 7, the analysis is iteratively refined to understand the overall software structure, its design decisions and details regarding the defined analysis goal utilizing the selected, representative project. To support this iterative process, the aspects contained in the checklist are sorted from a coarse-grained to a fine-grained level, i.e., from analyzing the overall software structure to analyzing selected properties of individual POUs.

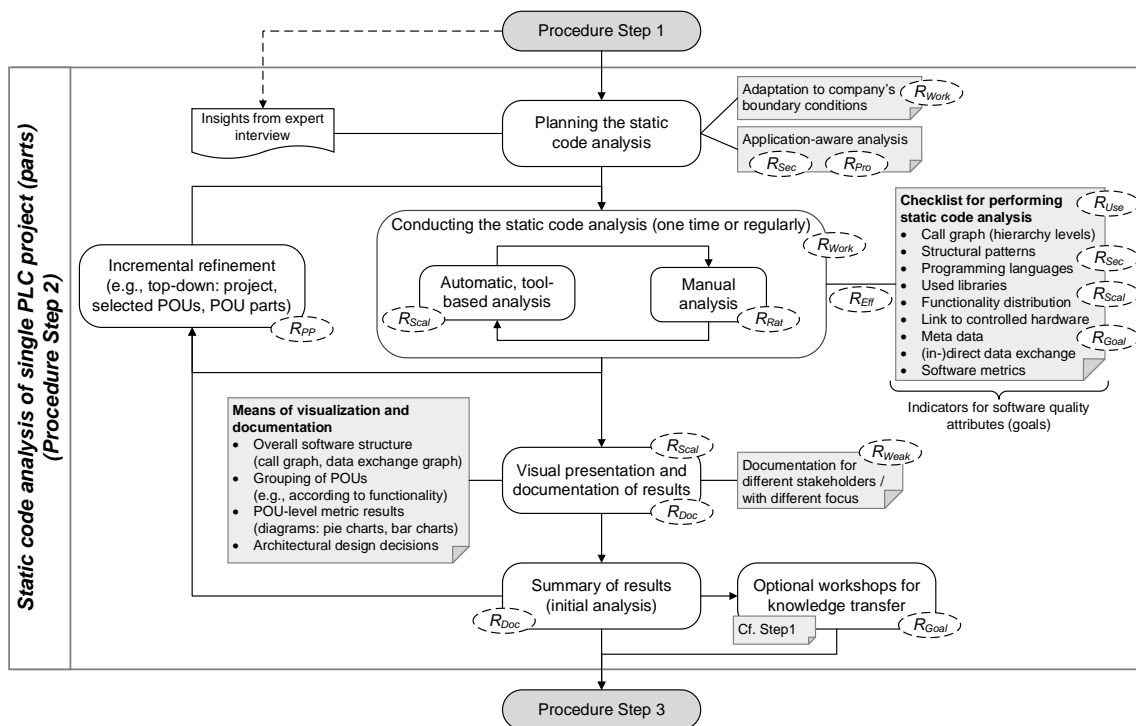


Figure 7: Details of Step 2 (Static code analysis of a single PLC project) of the quality assessment procedure.

In the following, the briefly introduced activities of Step 2 are illustrated utilizing *Case Study A*.

Planning the static code analysis based on insights from Step 1

From the interviews in Step 1, insights regarding the control software of *Case Study A* were gained. The considered warehouse control software is historically grown and software design decisions and influences of variation points are barely documented. However, knowledge about the general software structure, functionality distribution and design decisions is a prerequisite for planned reuse. Further, understanding the control software architecture and its relation to the automation hardware is essential for identifying variation points in different software projects (cf. Step 3). Thus, the subordinate analysis goal *documenting the current software architecture* is defined to address the identified challenge *lack of documentation on design decisions*. The results are relevant for determining software parts suitable for planned reuse, i.e., to address the defined analysis goal.

Due to the close link of PLC software to the controlled hardware, familiarization with the mechanical layout plan, the MVL-list and the unique equipment identifiers is required before the actual analysis. Generally, knowledge about the controlled aPS is a prerequisite for comprehending the rationale behind the design decisions in the control software (R_{Rat}). Moreover, a functionality-based definition of reusable software parts is often proposed [SFJ15]. Thus, in *Case Study A*, linking the software implementation parts to their functionality and the respective hardware modules is necessary. Otherwise, the functionality distribution and the separation of application-specific and standardizable software parts, which are essential to address the defined analysis goal, cannot be reconstructed. Further, a detailed analysis of the communication to external sources is required in *Case Study A* since it represents unchangeable boundary conditions. To address the defined goal, the analysis must be performed for differing, complete software project variants. A continuous quality assessment is not required to enhance planned reuse. Thus, the analysis can be performed as a post-processing step of finished software projects, which are known by the software developer currently programming the warehouse.

Generally, when planning the first static analysis, insights gained from Step 1 are summarized. These include identified challenges, boundary conditions and the selected analysis goal. Moreover, the insights enable an application-aware static code analysis by adapting to the application sector- or company-specific constraints (R_{Sec} , R_{Pro}). As illustrated in *Case Study A*, the goal of the initial analysis might differ from the defined overall analysis goal. This difference results from the aim of the first analysis, which is, apart from the defined analysis goal, to gain an overview and understanding of the basic software structure. Depending on the selected analysis goal and the company's development workflow, additional information needs to be considered and prepared for the analysis, e.g., the MVL-list in *Case Study A*. As highlighted above, for understanding the general software structure, it is essential to understand the controlled hardware and the aPS functionality. Therefore, apart from analyzing the control software itself, also influences from other disciplines must be considered for a holistic quality assessment of aPS control software.

Regarding its integration into industrial development workflows, the quality assessment procedure can be adapted and takes company-specific boundary conditions into account (R_{Sec} , R_{Pro}). The procedure is not linked to a specific point in the development process (R_{Work}). Consequently, the procedure enables, for example, a one-time quality assessment of the developed control software at particular development steps, e.g., at the end of the development process like in *Case Study A*. Moreover, continuous quality monitoring during the development is also supported, e.g., to analyze software evolution and monitor the modifications performed to standardized software parts during the development process (cf. *Goal 4* in Table 7) as described in [Fis⁺21a].

For integrating the quality assessment into the workflow, it needs to be examined at which points in the workflow the intended analysis is valuable and efficient. This examination includes available resources to realize adaptations if the analysis results indicate software structures negatively affecting the software quality, especially if continuous quality monitoring is targeted as described in [Fis⁺21a]. To support this step, information regarding the software development process, which was gained during the expert interviews in Step 1, can be documented utilizing the Business Process Model and Notation (BPMN) (depicted in Figure 8) or its extensions tailored to the aPS domain presented in [Vog⁺21b]. With BPMN, the organization of the software development department into sub-departments (e.g., separation of a standardized module and customer-specific application development in Figure 8), the use of information from other disciplines (e.g., the MVL-list from the mechanics' department) and the general workflow, including reuse of software parts from libraries or templates (cf. module database in Figure 8), can be illustrated.

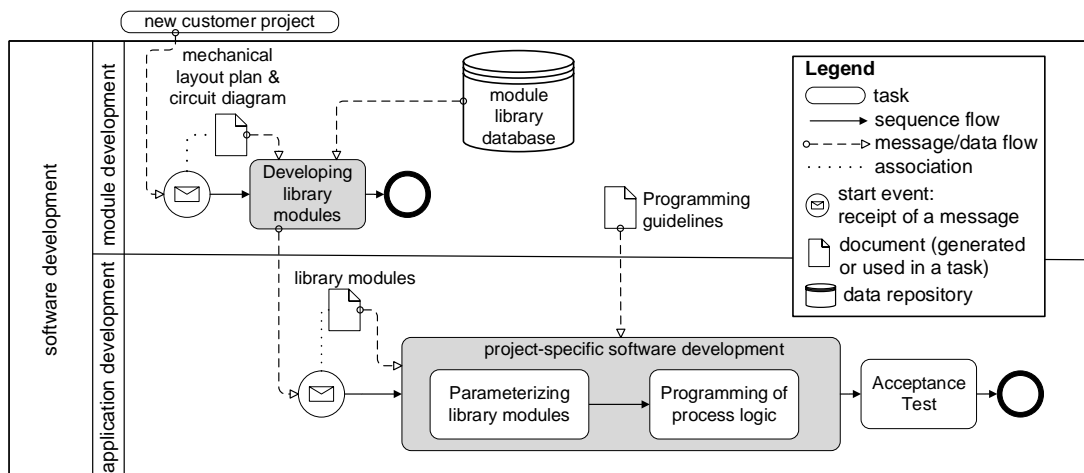


Figure 8: Exemplary software development workflow of an aPS manufacturer modeled using BPMN.

The resulting model includes involved stakeholders, their tasks and dependencies between departments or stakeholders. Such a model supports identifying an appropriate point in the development workflow to perform the goal-oriented, static code analysis and use its results.

Iterative, static code analysis of selected project

During the first analysis in *Case Study A*, mainly manual static code analysis was conducted to understand the rationale and design decisions in the control software. For this purpose, available means of the Siemens PLC development environment were used, e.g., reference data like the call structure or read and write access to selected variables [Sie06]. Wherever possible, this was supported by an available prototypical tool for automatic code analysis described in [Fuc⁺14].

The initial analysis is iteratively performed in accordance with the identified sub-goal *documentation of software architecture*. Selected aspects targeted during the analysis are listed in Table 8,

p. 65 (cf. detailed checklist in Appendix A.2, p. 191) and referred to in the following. First, a rough overview of the software project, the included POU and DBs, is targeted (*Aspect 1*). Due to the unique equipment identifiers, some POU can directly be linked to the hardware modules they control. Next, the call graph is generated with the prototypical analysis tool (*Aspect 2*) and illustrated in Figure 9. It enables the identification of direct call dependencies between the POU and the four hierarchy levels, representing the control software's coarse structure. Furthermore, frequently called POU, such as *warehouse route*, can be identified, indicating potential library modules (*Aspect 3*). In contrast, dead code, i.e., uncalled POU, can be identified as highlighted in Figure 9. The call graph forms the basis for analyzing the functionality distribution in the control software, for example, by highlighting all POU required for the storage car control (cf. Figure 9) or POU involved in error management (*Aspect 6* and *7*). Annotations of the call graph are part of the documentation of the analysis results and form the basis for more fine-grained analysis.

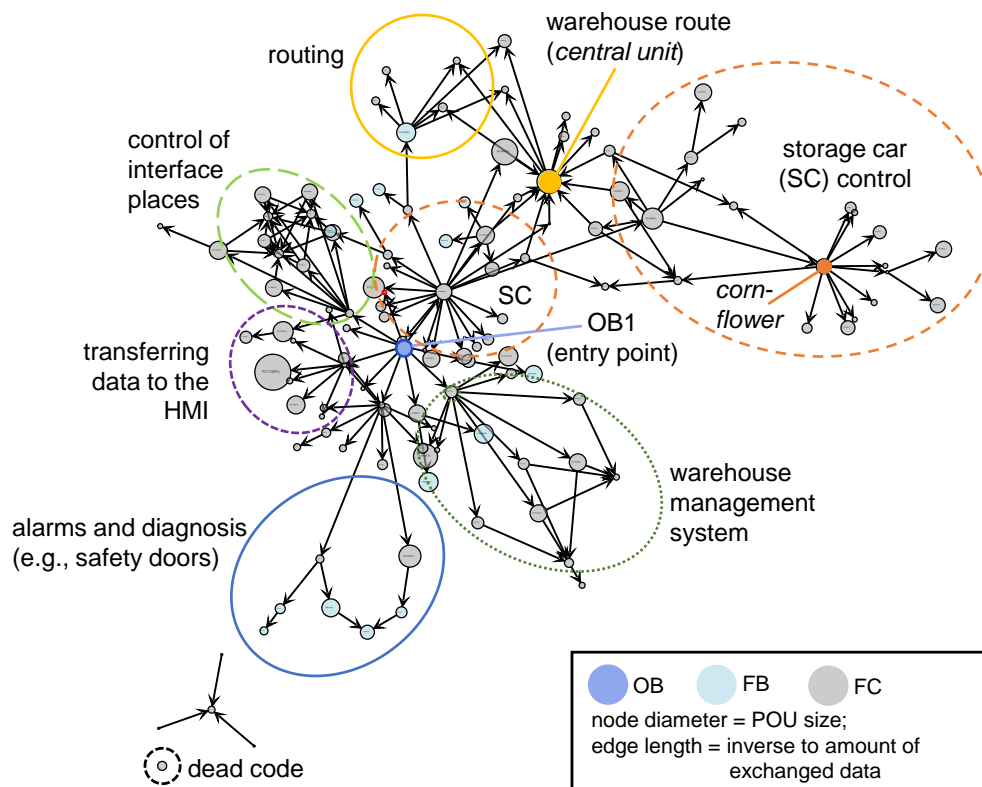


Figure 9: Annotated call graph of the analyzed software project of Case Study A, with entry point OB1.

By analyzing the call graph in combination with reviewing the control software in the development environment, the *call sequence* in a program cycle is analyzed. During this part of the analysis, interviews with the software developer are conducted. These also include the communication of the warehouse PLC with external sources, e.g., the WMS and the storage car (*Aspect 10*). Regarding the selected analysis goal, the exchange with the software developer is essential to understand the *functionality distribution* within the overall software structure, i.e., assigning functionalities to

the POUs within the software to analyze the effects of the warehouse variants on the control software in Step 3. The analysis confirmed the five main functionalities identified during the expert interview in Step 1 and they were annotated manually in the call graph.

Finally, the information flow, especially the indirect data exchange via DBs, is targeted (*Aspect 8*). As known from the expert interview, global DBs are used to organize and exchange data. The data exchange graph generated with the STEP 7 prototype [Fuc⁺14] confirms this: 100 DBs are contained in the project, with most POUs accessing and exchanging data via these. The variables are grouped according to their “functionality” in the program and the DBs have meaningful names. While this eases the comprehensibility of the software, it also allows the reuse of DBs during the software development via *copy, paste and modify*. For example, in the case of a warehouse with two storage cars, the DB containing the variables related to the storage car is copied and reused for the second storage car [FVF15]. Overall, an overview of the software architecture is achieved with the iterative software analysis, including multiple discussions with the software developer.

The performed analysis in *Case Study A* confirms the necessity to combine manual and automatic, tool-based analysis, as highlighted in [Sam⁺13]. For example, for the identification of the functionality distribution, considering information from different disciplines such as the mechanical layout plan or the MVL-list is necessary. This requires handling and understanding differently structured documents, which are usually highly company-specific. It is currently impossible to automatically take this information into account during automatic code analysis. Consequently, manual code audits and interpretation are necessary, for example, when amending the call graph with semantic insights from the discussions with the software developer (R_{Rat}). Nevertheless, using tools facilitates the analysis by reducing manual effort, e.g., the automatically generated call graph used in *Case Study A*, including the size metrics and the amount of exchanged data. Thus, a tool-based code analysis should be performed whenever possible to gain insights about the control software regarding the selected analysis goal as efficiently as possible. However, manual analysis is still required to prepare the tool-based analysis, interpret the automatically generated results and assess the control software in the context of the company- and application sector-specific boundary conditions. Current tools cannot perform this step, especially whenever expert knowledge is required to understand, for example, the intention behind the code structure or POU dependencies.

Different analysis means for static code analysis targeting various aspects are available. Potential aspects to be considered are summarized in Table 8 and the corresponding checklist in Appendix A.2 on p. 191. The aspects in the table are linked to the expected insights gained to enable software developers to perform systematic, goal-oriented code analysis on their own (R_{Use}). The checklist

is intended as a starting point, which guides the software analysis from the overall software structure to implementation details. It is proposed to start with coarse-grained aspects for gaining an overview of the software structure, cf. *Aspects 1 and 2*. Conspicuous, structural patterns identified during this coarse analysis (cf. *Aspect 3, 4 and 5*) provide an entry point for continuing the analysis on a more detailed level, e.g., by considering sub-parts of the control software, individual POUs and their dependencies or even implementation details within selected POUs (e.g., *Aspects 6 to 10*). For example, after gaining an overview of the contained elements and their dependencies using call graphs, searching for recurring call patterns as defined in [Fuc⁺14] supports the identification of structural clusters in the control software, which should be targeted closer by conducting a manual code analysis. These structural patterns are functionality indicators [Neu⁺20c] and, thus, give insights into architectural design decisions, e.g., if extra-functional tasks are implemented centrally or distributed in each module.

Table 8: Aspects to be targeted during the static code analysis (no claim to completeness).

Number	Analysis Aspect	Expected insights regarding architectural design decisions
Aspect 1	Number and type of elements	<ul style="list-style-type: none"> • Overview of the control software • Basis to assess functionality distribution, planned reuse (e.g., instantiations of FBs) and data exchange between elements or hierarchy levels
Aspect 2	Call graph and architectural hierarchy levels	<ul style="list-style-type: none"> • Dependencies between POUs via calls to understand the software structure • Combinable with views including the functionality distribution or POUs grouped according to the folders they are organized in. • Amount of hierarchy levels as an indicator regarding the encapsulation of functionality: <ul style="list-style-type: none"> - a flat hierarchy might indicate that process logic and hardware control are implemented on the same level, potentially mixed within one POU - a distinctive hierarchy might indicate that encapsulation is performed on a very fine-grained level, requiring many POUs on different levels to fulfill a certain functionality
Aspect 3	Structural patterns	<ul style="list-style-type: none"> • Identification of frequently called POUs (potential library POUs) or POUs that exchange data solely indirectly (cuckoo pattern described in [Fuc⁺14]) • Interpretation of implemented functionality provides insights into architectural design decisions (e.g., functionalities implemented by frequently used POUs, cf. [Neu⁺20c] for recurring structural patterns and their influence on software architecture)
Aspect 4	Included libraries	<ul style="list-style-type: none"> • Gaining a rough overview of available (company-specific) libraries efficiently, including the levels they are located on • Estimating the amount of planned reuse via used POUs from libraries
Aspect 5	Organization of software in the development environment	<p>If the development environment supports organizing POUs, e.g., in folders, this structure should be analyzed as it could provide hints regarding</p> <ul style="list-style-type: none"> • Functionality distribution across different POUs • Modularization strategy, e.g., multiple POUs organized in a folder for control of a particular hardware module and planned reuse

Number	Analysis Aspect	Expected insights regarding architectural design decisions
Aspect 6	Standardized and application-specific parts	<ul style="list-style-type: none"> • Assessment of separation of concerns (considering data exchange and hierarchy levels) • Assessment of reusability of control software parts • Standardized, application-independent POUs are potential library modules • Project-specific parts can potentially be generated from the information of other disciplines or merged into parameterizable POUs
Aspect 7	Extra-functional software parts/functionality distribution	<ul style="list-style-type: none"> • More fine-grained than Aspect 6 • Separation of functional and extra-functional software parts into standardized and application-specific parts • Insights regarding architectural design decisions and general software structure, which influence the reusability of implemented solutions (cf. [Vog⁺22a]); • Consideration of the interface between different functionalities (loose or close coupling affects modularity)
Aspect 8	Indirect data exchange graph (including information flows)	<ul style="list-style-type: none"> • Indirect dependencies are likely to hamper the software's reusability, especially as the dependencies are not directly visible • Data exchange and information flow provide insights into a software's degree of modularity and, consequently, reusability of sub-parts of the implementation in a different context
Aspect 9	Properties of individual POUs/groups of POUs	<ul style="list-style-type: none"> • Calculation of software metrics targeting specific quality attributes on the POU level to compare a group of POUs regarding an attribute, e.g., <ul style="list-style-type: none"> - Identification of the most complex POUs - Assessment of similarity of several POUs (estimate their suitability for being merged into a library POU) - Amount of dependencies to other parts of the control software • Selection of software metrics depends on the analysis goal
Aspect 10	Communication with external systems	<ul style="list-style-type: none"> • Insights on the information required to take control decisions in the PLC software's control logic • Interfaces for human intervention via HMI • Depending on the application sector, e.g., in intralogistics, routing is performed by a material flow controller outside the PLC • Communication of POUs executed on different PLCs, e.g., drive synchronization across two PLCs for product transportation

Guided by this checklist, the analysis in Step 2 aims to understand the software architecture, including the underlying design decisions, and identify software examples for the challenges and pain points identified in Step 1. Regardless of the software size and the selected goal, the procedure provides a set of relevant aspects for general software comprehension, which have been successfully applied in former research. Due to the size of control software (up to 450 POUs in a project, with up to 1500 LOC in a single POU [Fis⁺21b], and 1125 call edges in the constructed call graph), it is beneficial to use an automatic, tool-based analysis wherever possible. Especially for aspects such as calculating software metrics (cf. *Aspect 9*), many platform providers already offer functionalities in their IDEs, e.g., *CODESYS Static Analysis* [COD22] or the *Machine Code Analysis* from Schneider Electric [SE22a]. These tools are already applied in industrial practice and can be integrated flexibly into the procedure if their results provide insights for the chosen analysis goal.

Moreover, by sequentially focusing on selected aspects, the procedure supports coping with industry-sized software projects (R_{Scal}). Thus, insights gained from analyzing a coarse-grained aspect from the checklist refine the details to be targeted in the goal-oriented analysis of the following aspect. Thereby, application sector-specific guidelines or regulations identified in the expert interviews in Step 1 must be included as they support refining the subsequent analysis steps (R_{Sec}).

In the following, two central properties of PLC software, which have been identified as highly relevant for the software's quality assessment, are described in greater detail. First, **data exchange in PLC projects** is essential for their quality assessment. It can be classified according to and depending on the technically possible types, which the chosen PLC platform allows [Fis⁺22b]. For example, the Siemens development environment TIA Portal supports the use of flags and DBs, while the IEC 61131-3 provides GVLs to exchange data globally. Especially in the analysis of *Aspect 2* and *8*, these technically possible types of data exchange need to be known. However, quality assessment additionally requires a classification into intended data exchange and violations of a company's programming guidelines, hindering the reusability of software parts [Fis⁺21a]. For this purpose, the documentation and insights gained in Step 1 are considered. If required, further expert interviews need to be conducted to rate the criticality of violations regarding data exchange. Metrics concerning the dependencies and data exchange of software parts can be applied and linked to criticality levels as suggested in [Fis⁺21a; Fis⁺22b]. In summary, assessing dependencies between POU's regarding reuse highly depends on existing programming guidelines and requires manual analysis, at least manual pre-processing (R_{Rat}).

Another essential factor is the **consideration of semantics**: for example, to assess the call graph and the implemented data exchange, it is necessary to interpret which POU's implement which type of functionality and, therefore, exchange which type of information [Wil⁺22]. Without the semantic meaning of the exchanged data, it is impossible to assess whether the dependencies are required or could be avoided. This, in turn, is essential to rate the software's overall structure and modularity (cf. *Goal 1* in Table 7). The same holds true for quality attributes such as encapsulation or merging POU's into library modules, which are targeted, for example, in *Goals 2, 3* and *5*. If a similar structure is the only factor considered for merging POU's, two semantically different functionalities could be combined within one library module, which negatively impacts the understandability of the control software. First approaches to determine a POU's primary functionality based on implementation characteristics and naming conventions are available [Wil⁺22]. Although the approach shows promising results, it still has some shortcomings. For example, its applicability is limited to mature control software with a highly modular, function-oriented structural design. Thus, emerging approaches have high potential and support manual code analysis in specific cases, but they cannot yet replace it entirely.

The developed checklist serves various purposes for conducting the static code analysis. In the following, two exemplary scenarios are shortly introduced. If the **analysis goal is not clearly determinable** in Step 1 (which is often the case), the checklist can be followed to identify coarse aspects, e.g., the number of hierarchy levels, implemented reuse strategies and (in-)direct data exchange for gaining an overview of the design decisions. The results are discussed with the industrial experts to identify pain points and select a suitable analysis goal according to the expected benefits (R_{PP}). Furthermore, if required, discussing initial results enables refining the selected analysis goal. Overall, the main aim of the checklist is to provide a structured procedure for the analysis and, thereby, support different analysis goals (R_{Goal}).

If the targeted control software is **historically grown legacy software without available documentation**, performing a systematic, goal-oriented static code analysis following the checklist nevertheless supports identifying improvement potential. Of course, based on the analysis results, it is necessary to outweigh the effort of refactoring the legacy software compared to implementing it new. However, even if it is decided to completely renew the control software, it is possible to take insights from the performed analysis into account, which can avoid making the same disadvantageous design decisions as in the old architecture. In the case of undocumented legacy control software, **gaining an overview of the software structure** is a primary analysis goal; thus, comparing direct and indirect data exchange graphs (*Aspect 2* and *8*) and considering structural patterns (*Aspect 3*) is beneficial. It provides information about existing interfaces and dependencies, which is essential for assessing the software's modularity and reusability. Another insightful aspect is the consideration of the functionality implemented in frequently called POUs. For the analysis of dependencies and functionality distribution, the call graph (*Aspect 2* in Table 8) is an appropriate entry point, which supports the identification of particularities that should be analyzed in detail, e.g., with a POU-interface analysis.

Visualization and documentation of results

While performing the first static code analysis in *Case Study A*, the insights are continuously documented as the analysis is iteratively refined according to the checklist. As depicted in Figure 9, the automatically generated call graph, including manual annotations, is used for visualization and documentation purposes. From the call graph, the architectural hierarchy levels and the reuse of POUs on different levels are derived and summarized using the illustration from [Vog⁺15a] (cf. Figure 10, left). The entry point to the warehouse software is a *facility module*, which is influenced by a superordinate *plant module level* control. The visualization shows if and on which hierarchy levels frequently called POUs exist. These are potential candidates for the development of library POUs, which are directly linked to the selected analysis goal. Furthermore, it highlights the call dependencies between POUs. In the warehouse software, call dependencies are identified between

POUs within one hierarchy level or adjacent hierarchy levels. Additionally, the functionality distribution and related information flows are documented with respect to the identified hierarchy levels. Figure 10 depicts the information flow regarding alarm detection across the different levels on the right. Thus, it documents a design decision regarding the implementation of the extra-functional task *error handling*. Regarding planned reuse, analyzing and documenting the information flows of extra-functional software parts supports the identification of required POU interfaces.

Overall, there is little documentation about the software structure and design decisions of the warehouse available. Thus, the gained insights during the analysis are summed up on a coarse-grained level, as depicted in Figure 10. This documentation format enables depicting the general software structure and design decisions on an abstract level and independent from the analyzed control software project. Apart from alarm handling, other extra-functional tasks are considered and can be visualized as suggested by [Neu⁺22]. Furthermore, tables are used to list all POUs in the warehouse with the corresponding main functionality they implement. Moreover, dependencies between the mechanical layout plan and the control software are documented in textual form.

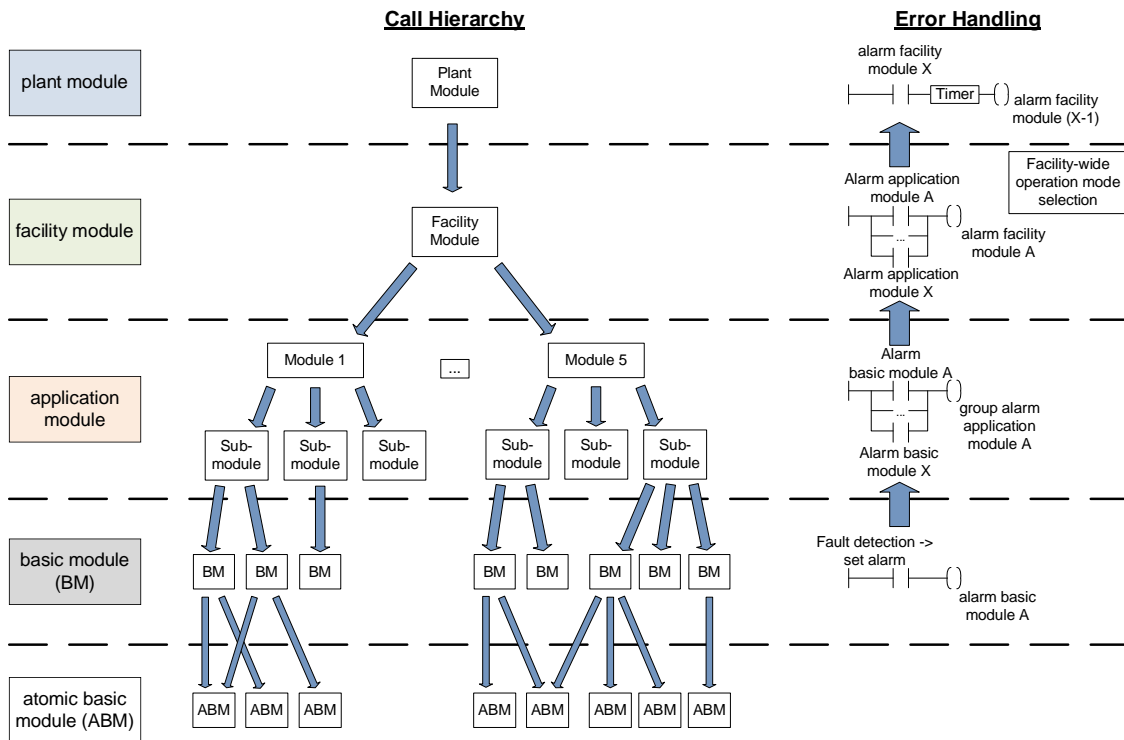


Figure 10: Call hierarchy (left, adapted from [Vog⁺17]) and communication of errors (right, adapted from [Vog⁺15a]) in regard to the identified architectural hierarchy levels in Case Study A.

As highlighted in *Case Study A*, documentation of different aspects is generated or manually derived. This documentation depends on the chosen goal and considered aspects but should contain an overview of the general software architecture, the identified design decisions and pain points.

Examples of documentation are, e.g., call graphs optionally annotated with functionalities, a design structure matrix to depict dependencies between POUs/groups of POUs or different types of diagrams reflecting calculated metric values. Depending on the focus of the analysis, e.g., software structure or implementation details of individual POUs, suitable means for the results' documentation need to be selected. Since there are no universally agreed-on thresholds for different metric values and software architectures can be quite different between different companies and application sectors, a general assessment of the control software by comparing it to an ideal implementation is not possible. Instead, the various gained analysis results should be visually prepared and presented in the context of the analyzed software set to detect outliers (R_{Weak} , R_{Doc}). Further, gained analysis results can be filtered to cope with the size of industrial control software projects (R_{Scal}). Examples of this are focusing on sub-parts of the call graph or depicting the “*worst/top ten*” POUs concerning a calculated metric as proposed in [Fis⁺21b] to identify the ten most complex POUs, which are expected to result in the highest benefit after refactoring. Finally, the documentation should be gathered on different degrees of abstraction, depending on the involved stakeholders and their tasks [Fis⁺20a].

Summary of insights gained – general and regarding the analysis goal

At the end of analysis Step 2, the insights gained are summarized from the documentation. In *Case Study A*, the software structure and design decisions are derived. In the control software, the *facility module* warehouse is called by a top module on the *plant module* level and the control software is divided into four architectural levels. On the top level, the five main functionalities identified during the expert interview are confirmed and each is represented by a POU on the *application module* level. Each *application module* POU calls several POUs necessary to execute the specific tasks. The *application module* “car control”, for example, calls other FCs on the *application module* level belonging to the storage car, which subsequently call *basic* and *atomic basic modules*. All POUs can be assigned to exactly one *application module*. The variables of the *application modules* are organized in DBs, which can be mapped to the respective hardware via the unique equipment identifiers. Although the control software is developed using *copy, paste and modify*, reusable *application modules* and associated variables structured in DBs could be identified. Moreover, many of the POUs at the *basic* and *atomic basic module* levels are already reused without adjustments, although no library modules exist. [Vog⁺15a]

Regarding the functionality distribution, also extra-functional aspects are targeted. For example, error handling in the warehouse software is divided according to the five *application modules*. Within the *application module* “general functions”, a few general alarms, i.e., safety door alarms, are generated and also alarms of other application modules are collected. Depending on the severity of a detected fault, the whole *facility module* warehouse or only parts of it are switched to

emergency mode. If the warehouse is switched to emergency mode, the alarm is additionally collected on the *plant module* level. After a delay time, the preceding *facility modules*, which transport the stacks to the warehouse, need to be stopped as well due to a limited buffer for accumulated goods. Moreover, communication between the warehouse and external systems is analyzed and the required information and the communication strategy are documented. [Vog⁺15a]

As illustrated with *Case Study A*, at the end of this procedure step, familiarization with the selected representative project is complete and the software structure and architectural design decisions are documented. In addition, positive or negative examples for challenges stated in Step 1 are identified to ensure that the challenges have been correctly understood. For gaining an overview of the selected project, the checklist provides a structure for analyzing different general aspects of control software that are potentially helpful for gaining insights regarding the selected analysis goal. Thereby, insights gained in Step 1 and company- and application sector-specific boundary conditions are considered to enable an application-aware analysis requiring manual code analysis (R_{Rat}).

5.2.3. Comparison and Results' Documentation of Additional PLC Software Parts or Projects Regarding the Selected Goal (Step 3)

In Step 3, further control software parts or projects are analyzed. An overview of the activities performed in Step 3 is depicted in Figure 11.

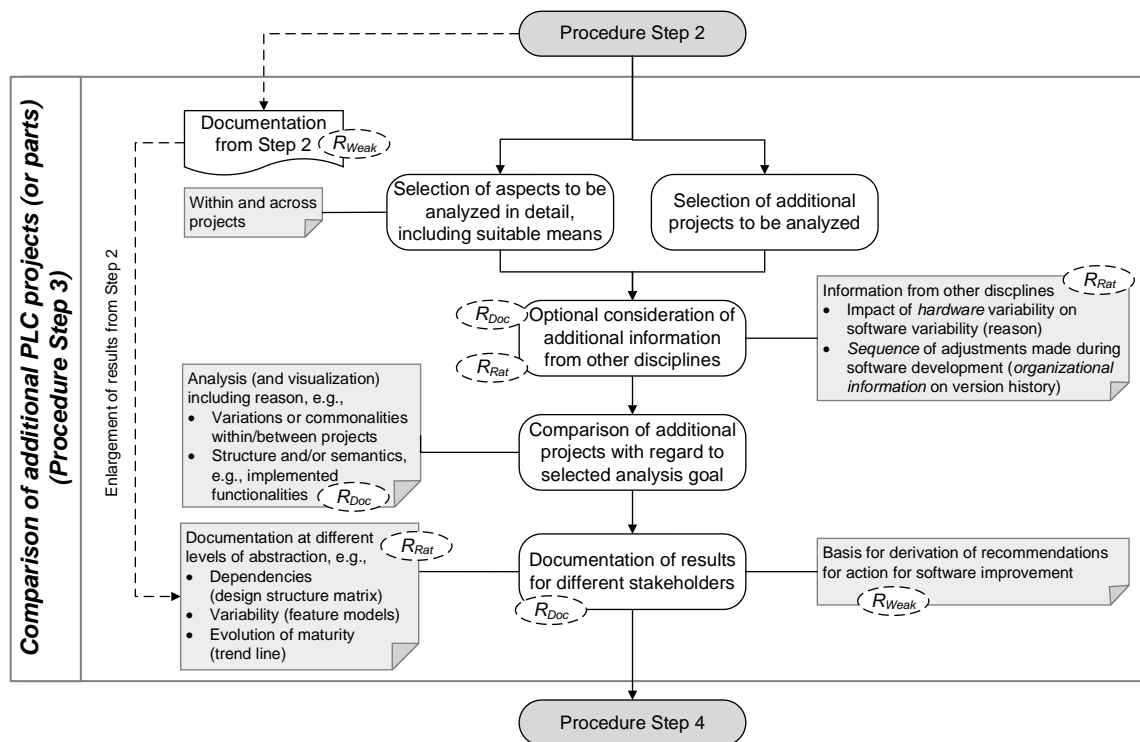


Figure 11: Details of Step 3 (Comparison of additional PLC projects or PLC project parts) of the quality assessment procedure.

Unlike in Step 2, the focus lies not on the structure and design decisions of a single project but on iteratively analyzing and comparing multiple PLC projects/project parts concerning the selected analysis goal. Accordingly, this procedure step is not intended as a repetition of the analysis described in Step 2. Instead, Step 3 uses the documentation from and, thus, insights gained in the previous analysis step to select the aspects to be considered in greater detail for assessing the selected analysis goal. Consequently, not all aspects that have been analyzed in the initial project need to be considered for the additional projects/project parts. Similar to Step 2, information from other disciplines is considered during the analysis. Moreover, the insights gained during the investigation of additional projects are documented. In the following, details regarding the individual activities in Step 3 are provided with respect to *Case Study A* and in general.

Selection of aspects and additional projects to be analyzed

During the expert interviews in Steps 1 and 2, the software developer of the warehouse has already mentioned typical variants of the warehouse hardware with effects on the control software. These variation points are used in Step 3 to select four additional projects, which represent all main storage variations and are, thus, suitable for gaining insights for the defined analysis goal. Since little documentation is available on the warehouse software design decisions, the developer is only aware of the rough effects of hardware variants on the software, but the details are unknown. Therefore, a detailed analysis of the commonalities and differences within the control software of different warehouse variants is targeted in Step 3. Further, to allow mapping the differences to their cause, warehouse variants, which differ in one identified variation point, are selected for the analysis. The selected control software variants are already in operation or shortly before commissioning to avoid differences resulting from incomplete software projects.

In *Case Study A*, the aim of Step 3 is to address the defined analysis goal *enhance planned reuse of variant-rich control software* by identifying POUs, which are common to all software variants, and POUs, which can potentially be parameterized for reuse, depending on the warehouse variant. By considering hardware variants and their influence on the control software, project-specific and project- or variant-independent software parts can be distinguished. Since the latter are suitable for planned reuse, in some cases even across different machines or plants, their implementation should be separated from project-specific aspects. Means for a detailed analysis of software project variants include coarse aspects like the amount of POUs (cf. Table 8, *Aspect 1*) to identify optional POUs and implementation details of the individual POUs (*Aspect 9*) to identify modified POUs.

Generally, by utilizing the documentation from Step 2, software parts linked to the chosen analysis goal, disadvantageous structures or conspicuous design decisions are identified for detailed analysis in additional projects. The checklist supports selecting relevant aspects to be targeted and

compared during the analysis of further projects, including suitable means for performing the analysis. Additional projects such as variants (as in *Case Study A*) or versions of the first project need to be selected, depending on the analysis goal and aspects to be considered in detail. In case of known challenges, the projects should include positive and negative examples, e.g., projects containing known, recurring issues during the commissioning. Moreover, projects representing frequently used variants, unique customer requirements or different optional functionalities are generally suitable for comparison regarding their implementation. Overall, the selected projects should give a representative overview of the targeted aspects. The selection of these additional projects ideally includes software developers who are familiar with the projects and indicate the projects' maturity based on their estimation and experiences.

Preparation of information from other disciplines

Prior to analyzing control software variants, the warehouse's variation points are documented from a mechanical and functional point of view. For this purpose, information from other disciplines is required. More precisely, the mechanical layout plans of the five selected warehouses are compared with the support of the software developer. The variation points in the mechanical view are extracted from the layout plans and summarized in a feature model, which contains mandatory, optional and alternative features of the considered system [Thü+14]. The general functions of the warehouse from a customer point of view are identified in expert discussions and documented in a second feature model, as proposed in [FV17].

The general functionalities offered by the warehouse remain unchanged despite variations such as different warehouse layouts, amount of storage cars or the optional use of protection boards. Independent of the storage variations, the functional feature model includes interface places to adjacent plant parts, handling board stacks, controlling the storage car(s) and ensuring security and safety. For example, whether the stack transportation is conducted by a single storage car or distributed on several cars is not relevant for the storage functionalities since the warehouse cars are all controlled identically. The number of storage cars is not a functional requirement but depends on the storage layout. Therefore, the general functions of the warehouse from a customer point of view are independent of the identified variation points and do not affect the warehouse control software. In contrast, the software developer's rough estimation of the effects on the control software showed that every identified variation point in the mechanical view of the warehouse, i.e., the number of interface places or the number of storage cars, has a direct influence on the software. For example, adding a further interface place requires an additional POU in the software. Overall, this analysis required domain knowledge from the software developer. [FVF15]

As illustrated with *Case Study A*, additionally to the control software itself, information and documents from other disciplines, e.g., functionality desired by customers or the hardware variability, need to be considered. They are essential when interpreting the reason or semantics behind identified differences in compared software parts, such as the project variants targeted in *Case Study A* (R_{Rat}). Moreover, insights gained during expert interviews and workshops are crucial and need to be included in the result documentation, as they provide reasons for changes or specific types of implementations. Additionally, organizational aspects and the performed steps during the software development process are helpful depending on the selected goal. For example, if the version history of a template-based project is compared to estimate the template's suitability, correctness, and completeness, the sequence of adjustments to the template across projects is relevant. Adjustments performed in various projects indicate potential points of improvement of the template. They must be documented for an expert discussion, ideally with the template developers and the application engineers using the template. Thus, depending on the focus of the analysis in Step 3 and the targeted goal, information about the software development process and from other disciplines is essential to plan the analysis, document the results and interpret them (R_{Doc} , R_{Rat})

Comparison of additional projects or project parts

After the impact of mechanical variations on the control software was confirmed, a detailed analysis of the control software variants was performed. For this purpose, five PLC software projects of different warehouses were compared. As a starting point, the software project known from Step 2 is compared with a second project differing in the mechanical variation point "amount of storage cars". First, the number of software parts (POUs, DBs and UDTs) in the two programs is determined to identify optional, variant-dependent parts. Next, software parts contained in both PLC projects are compared to identify parts, which are modified according to the number of storage cars. This requires a manual comparison of POUs on the network and even variable levels. A difficulty during this step was that the distinguished differences are not only caused by variants but are also evolutionary, such as adding and querying an extra sensor to avoid errors. [FVF15]

In order to identify and document the influences of the variation points on the control software, in total, six project comparisons are conducted. During the comparison, automatically generated call graphs are used as a starting point. These support the identification of optional POUs and dependencies. As illustrated in Figure 12, the call graphs of warehouse variants are highly similar regarding the overall structure, i.e., call dependencies and complexity distribution between POUs, which results from the applied reuse strategy of *copy, paste and modify*.

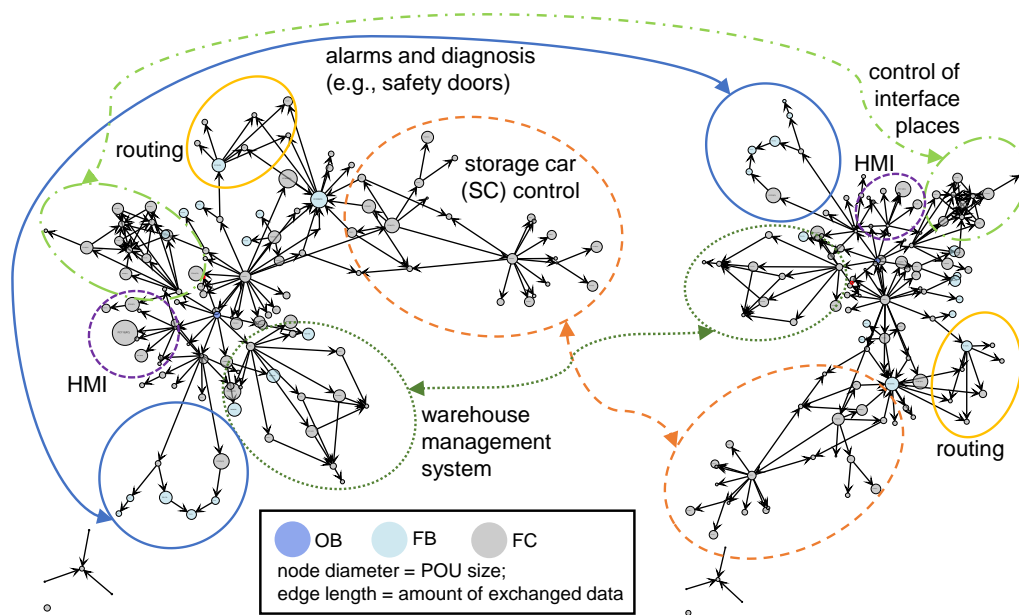


Figure 12: Call graphs of two warehouse software variants from Case Study A differing in the mechanical variation point “amount of satellite cars” (124 POUs each).

In some cases, such as the variation point “number of satellite cars”, no optional POUs are identified. Manual analysis is required to analyze modifications on the sub-POU level since available analysis tools could not provide results on the required level of detail. Nevertheless, available means in the Siemens STEP 7 development environment were used to support the manual code analysis. For example, deleting optional POUs in a software project and, subsequently, running consistency checks in STEP 7 (cf. chapter 15 in [Sie06]) eased the identification of POUs, which require modifications due to the considered variation point, since they use variables belonging to the deleted, optional POU. Apart from manually detecting modified POUs, a manual interpretation of these modifications is essential to separate the changes resulting from variation points and evolution (R_{Rat}). Although the detailed manual code analysis is time-consuming, it is the basis for distinguishing the invariable, unchanged core software parts, reusable without modifications, from the variant-dependent software parts requiring adjustments depending on the particular variant.

As illustrated with *Case Study A*, during Step 3, an iterative analysis of the PLC projects is performed. The selected analysis sub-steps, e.g., project comparisons according to a selected variation point on different granularity levels in *Case Study A*, are repeated. When comparing the analysis results of different projects regarding the selected analysis goal, aspects such as modifications, variations and commonalities need to be documented, including their scope, e.g., the structure or the functionality in general and application-specific parts of the investigated software (R_{Doc}). Furthermore, the documentation should include the reasons for the differences, e.g., a software change resulting from using different automation hardware. Consequently, documentation generated from utilized analysis tools, e.g., call graphs, is not always sufficient and often needs to be amended or

detailed with manual interpretation and analysis (R_{Rat}). Additional expert interviews should be performed if the analysis results gained during Step 3 contradict the software developers' experience knowledge. Depending on the interview insights, the analysis means need to be adapted or changed for the best possible result.

Documentation and results preparation for different stakeholders

Different means of documentation are used in *Case Study A*, with some requiring workshops with the software developer. For example, feature models are used to document the variation points of the warehouse from different views. Feature models are a means from the computer science domain to document mandatory, alternative and optional parts in variant-rich software systems. In a workshop with the software developer, the basic theory of feature models is introduced and subsequently, the feature models were jointly derived. Additionally, annotated call graphs and tables are used to document the conducted comparisons of the control software variants.

In total, six variation points of the warehouse software were identified and are listed in Table 9. Each variation point directly influences the warehouse software. Accordingly, six project comparisons were conducted and their results are summarized in a table similar to Table 9. Instead of illustrating the variability with commonly used feature models, a table was chosen due to the possibility of clearly and directly opposing the effects of the different variation points on individual software parts. All identified software parts, i.e., POUs, DBs and UDTs, of the warehouse software are listed in the first column. The remaining columns indicate how the different variation points affect the identified software parts. [FVF15]

Table 9: Summary of the software comparisons during manual static code analysis by means of selected software parts, published in [FVF15].

Software Parts	# Satellite Cars	# Storage Cars	Optional Protection Boards	# Interface Places	Optional Forklift	# Areas, Groups, Member
Part 1 (OB)	-	-	-	-	-	-
Part 2 (FB)	adapt	adapt	adapt	adapt	adapt	-
Part 3 (FC)	adapt	-	-	-	-	-
Part 4 (FC)	adapt	optional	-	-	-	-
Part 5 (FC)	-	-	-	-	optional	-
Part 6 (DB)	-	-	-	-	-	adapt
Part 7 (UDT)	adapt	-	-	-	-	-

Legend: “-“: variation point has no influence on the software part; “adapt“: variation point requires adaptations to the software part; “optional“: POU is optional depending on the variation point.

During Step 3, the documentation from Step 2 is enlarged to gain an overview of the current software state. The enlarged documentation is the basis for identifying weaknesses and improvement potentials in the control software (R_{Weak}). While the type of documentation depends on the analysis

goal, the aspects targeted and the methods used, it is generally essential to document the control software on different levels of granularity for identifying anomalies within the context of the considered software. Context-sensitive documentation and assessment of the results are essential: except for a few guidelines, e.g., SAIL in the intralogistics domain [VDI5100], there are barely any guidelines about commonly accepted rules to follow that support that control software has high quality. For example, there is no universally agreed-upon threshold indicating that control software is too complex [Fis⁺21b]. Similarly, as pointed out above (cf. Sub-section 5.1.1, p. 45), the suitability of a reuse approach highly depends on the characteristics of the control software and the background knowledge of the software developers [Fis⁺21c]. Therefore, the documentation is essential to identify weaknesses, including anomalies and outliers within the analyzed software.

Overall, the documentation of the analysis results should contain examples of identified strengths and weaknesses as a basis for an assessment of the control software. Subsequently, recommendations for improving the control software are derived from the documentation in Step 4. For this purpose, the documentation should support the estimation of the effort required for performing the derived software modifications and the expected long-term benefits (R_{Weak}). Especially the estimation of effort and benefit requires much information about necessary changes. For example, concerning the data exchange between POU's and identified violations of the (company-specific) programming languages, the documentation should highlight where the violations took place, the involved POU's and how many violations regarding the total amount of data exchange there are. This documentation serves to identify points to be improved and how much effort the improvement potentially requires. While the effort estimation is usually performed by application and module developers, group leaders or managers generally take the estimation and subsequent decision if the required effort will pay off. After assessing the control software, the documentation needs to support these stakeholders in their tasks regarding if and how derived improvement recommendations are to be implemented. For this purpose, documentation on different layers of abstraction is required for different stakeholders involved in the software development process, similar to [Fis⁺20a] for the visualization and documentation of software variability.

Various means of software visualization are available from GPL and PLC platform suppliers, targeting different stakeholders. Some examples, including the documentation content as support for deriving recommendations for actions, are listed in Table 10 and illustrated in Figure 13.

Table 10: Selected, exemplary means for visualization and documentation of analysis results (not intended to be exhaustive).

Number	Visualization	Content for documentation of analysis results
Visu 1	Call graphs (optionally annotated)	<p>Overview of the structure of the considered control software project (or parts), including dependencies via calls and (in-)direct data exchange on different architectural levels.</p> <p>Possibilities for annotations:</p> <ul style="list-style-type: none"> • Semantics indicator: functionality distribution • Structuring of objects in folders: may represent similar functionality or the machine layout • Combined with metric results: complexity, size (in the context of architectural levels and calls) or distribution of code duplicates <p>Similar level of detail: software cities, design structure matrix.</p>
Visu 2	Heatmaps	<p>Navigational instrument to move through different architectural layers and see selected aspects in the scope of the current layer (starts with an overview but enables navigation to details from outliers on top-level).</p>
Visu 3	Chord diagrams	<p>Dependencies between selected elements (due to the size of industrial control software projects, rather a selection than an entire project; but also complete project view might be helpful to identify highly interconnected parts).</p>
Visu 4	Diagrams, Charts	<p>Kiviat diagrams, bar and pie charts, e.g.,</p> <ul style="list-style-type: none"> • Bar charts for complexity (including sub-measures), • Trend lines for software maturity across POU versions or • Pie charts for distribution of network complexity within a POU
Visu 5	Code diff views	<p>Available for textual languages to see differences between software variants or changed duplicates. First approaches for graphical languages available: [NVO15].</p>
Visu 6	Sortable lists	<p>Different metric values on POU implementation level inside a table for comparison, i.e., to sort the POUs concerning a selected software property, e.g., complexity.</p>
Visu 7	SPLs and feature models	<p>Variability is documented from different perspectives and on different granularity levels, e.g., customer, mechanics or control software view (POU or sub-POU level).</p>

As listed in Table 10, various approaches are available to visualize the characteristics of control software. Some of these are illustrated in Figure 13, sorted according to the targeted granularity level and stakeholders. On an abstract level, call graphs and software cities help gain an overview of the general software structure, including classifying contained sub-elements or optional annotations (*Visu 1*). When targeting dependencies between entities on POU-level, a design structure matrix, chord diagrams and dependency graphs are suitable to represent inter-relationships between several entities (*Visu 1* and *3*). Especially chord diagrams (*Visu 3*) have an advantage regarding the illustratable amount of inter-relations. On a fine-grained level, sometimes even sub-POU level, different diagrams such as trend lines, bar graphs and pie charts are available (*Visu 4*). Finally, approaches such as a side-to-side view of variants with color-coding to compare software implementations and their variability exist (*Visu 5*). [Fis⁺20a]

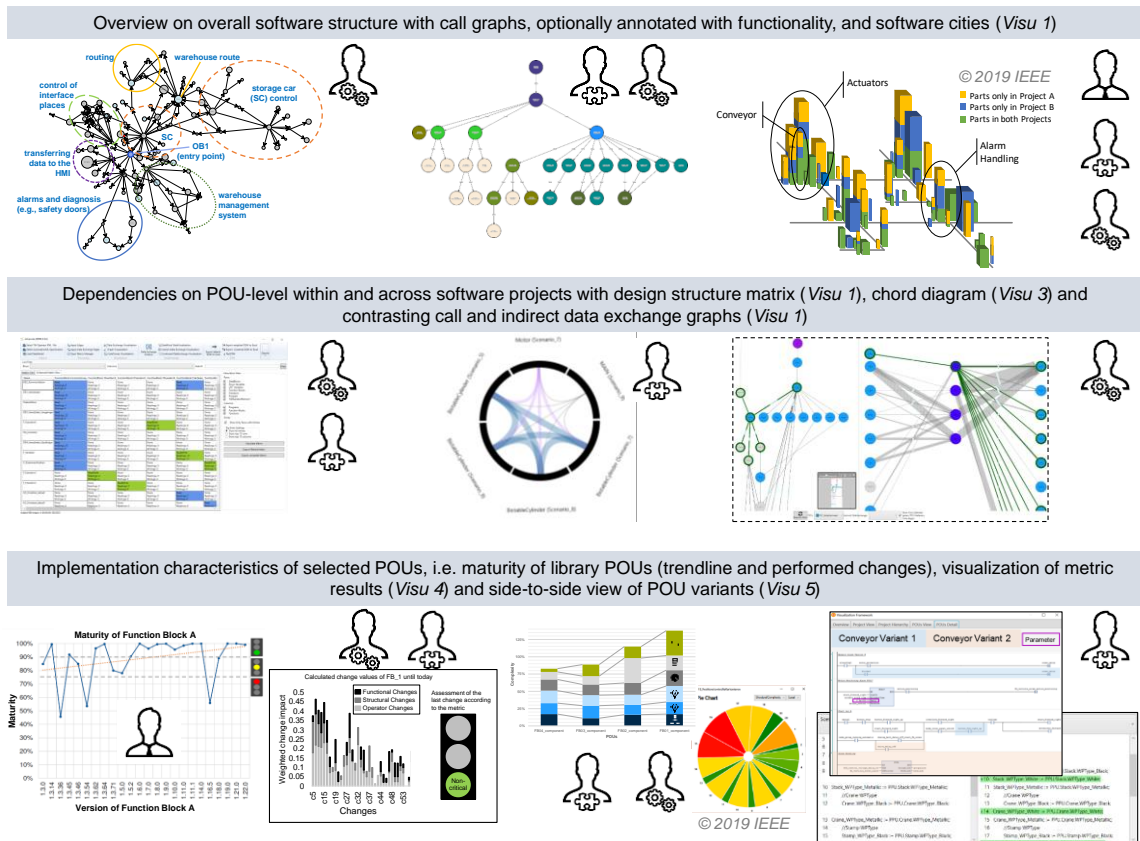


Figure 13: Different means of visualization and documentation of analysis results for different stakeholders such as managers (👤), application developers (👨‍💻), and module developers (👨‍🔧); summarized from [Bou⁺19; Fis⁺20a; Fis⁺21b; Neu⁺20b; VNF22; Vog⁺16; Wil⁺19; Wil⁺22].

Overall, by comparing multiple projects regarding selected aspects in Step 3, the documentation from Step 2 is enlarged. The documentation should be on different granularity levels to fulfill the needs of different stakeholders, include information relevant to the control software from different disciplines (e.g., hardware variants influencing the control software) and focus on the selected analysis goal. The documentation serves two general purposes: first, it is a means to document the current state of the control software (e.g., its structure, functionality distribution, implemented ways of data exchange or complexity) since available documentation is often outdated [Kir⁺16]. Moreover, the documentation needs to support the identification of outliers and anomalies. Due to the lack of generally applicable guidelines for achieving high-quality software, these should always be considered and assessed in the context of the analyzed project to select implementation parts with a high potential for improvement. The documentation is the basis for identifying strengths and weaknesses and the overall assessment of the control software regarding the selected analysis goal. Consequently, it is essential and determines the quality of the recommendations of actions that will be derived in the final procedure step. The documentation needs to be fine-grained enough to estimate the effort of proposed modifications, e.g., determine the amount of POUs that need to be changed or the risks of changing specific software parts.

5.2.4. Identification of Improvement Potentials, Including the Derivation of Recommendations for Action (Step 4)

Based on the documented analysis results, including project comparisons, an assessment of the control software regarding the selected analysis goal (R_{Goal}) is performed in Step 4. Thereby, strengths and weaknesses in the analyzed control software are identified. Subsequently, recommendations for actions are derived to enhance the software quality concerning the identified weaknesses and the selected analysis goal. The derived recommendations are compared regarding their estimated implementation effort (R_{Weak}), the expected benefit and the potential risk when performing the recommendation for action. Moreover, concerning the applicability of the derived recommendations for action, their inclusion in the development workflow is reviewed (R_{Work}). Consequently, this step's outcome is a list of recommendations for action, including software parts affected by the proposed change and where it should be made in the workflow. If required, available programming guidelines, used templates or even the development process itself need to be adapted. An overview of the tasks performed in this step is given in Figure 14.

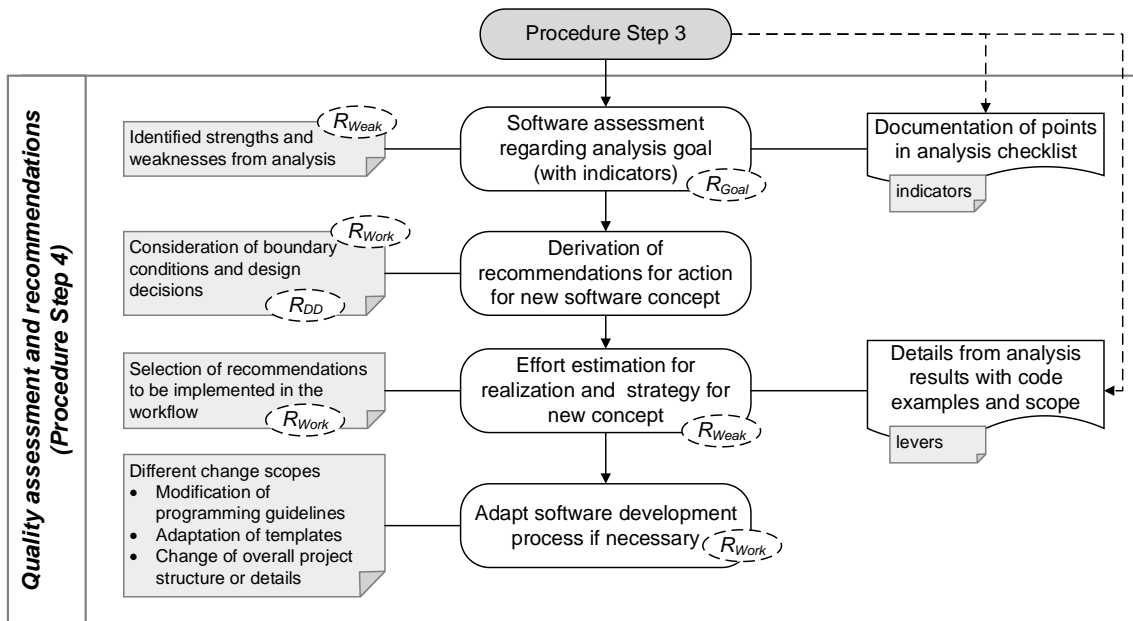


Figure 14: Details of Step 4 (Quality assessment and recommendations) of the assessment procedure.

The individual activities are introduced in detail below utilizing *Case Study A*.

Software assessment based on analysis results

The documentation gained from Steps 2 and 3 about the warehouse control software from *Case Study A* is interpreted for assessing the control software concerning the defined analysis goal. Since the targeted goal is to enhance planned reuse, aspects directly related to software reuse, e.g., the amount of duplicated code and the influence of hardware variants, are focused (R_{Goal}). From the comparison of five PLC software variants, three primary influences of hardware variation

points on control software parts are derived, i.e., no influence, requiring adaptations and optional software parts (cf. Table 9). Overall, the software parts can be divided into two main groups: a set of invariable software parts, which are common to all warehouses and not influenced by the hardware variation points, and variant-dependent parts, which require modifications or are optional depending on the specific variant (cf. [FVF15] for a detailed interpretation of hardware influences). In the variant-dependent group, both the POU and the related variables need to be modified according to the chosen warehouse variant.

These insights regarding the software structure are used to identify the strengths and weaknesses of the analyzed software. An apparent weakness is the use of *copy, paste and modify*, which leads to a high amount of duplicated software, which in turn decreases maintainability. Moreover, data exchange is mainly implemented via global variables stored in DBs. This might result from the high amount of FCs used in the warehouse software, but it potentially hinders planned reuse. However, also various strengths are identified. Due to the hardware-oriented structure of the control software, the influence of hardware variation points on the control software is limited. For example, the variation point “number of storage cars” affects the FCs for car control, general alarm FCs and visualization FCs, but no further POU. Furthermore, despite the extensive use of global variables, UDTs support structuring the variables with respect to the data they store, e.g., information about the storage car control. Consequently, POU and related UDTs for controlling warehouse sub-parts are identified, representing a basis for planned reuse. Another strength resulting from the overall software structure is that some POU, including their variables, are reusable without modifications and, thus, are suitable for developing library POU for planned reuse. Regarding the variant-dependent software parts, it is concluded that a suitable set of parameters is sufficient to describe individual warehouse variants entirely and unambiguously, including the control software. Since the unambiguous description of a variant and its effect on the control software are prerequisites for planned reuse strategies such as code configuration, this finding is highly relevant for the analysis goal.

As illustrated with *Case Study A*, the actual software quality assessment is conducted utilizing the acquired documentation of the different aspects analyzed in Steps 2 and 3, meaning the gained results for different projects with different methods. The main aim is to identify strengths and weaknesses, i.e., disadvantageous implementation parts or design decisions, in the control software projects regarding the targeted analysis goal (R_{Goal} , R_{Weak}). The disadvantageous software parts pose potential for improving the software quality, which requires further analysis. However, some of these parts might be unchangeable due to boundary conditions, requirements or deliberate design decisions. Therefore, the identified weaknesses need to be reviewed for their relationship

to these unchangeable parts, documented in Step 1 (R_{DD}). As a side effect, deliberate design decisions are associated with the analysis results during this task. Thus, they are evaluated: it is noticeable if a design decision results in many disadvantages regarding the analysis goal. If this is the case, a discussion with the software developers to reconsider these design decisions is proposed, taking into account their negative consequences. Overall, a manual interpretation of the documented analysis results, e.g., achieved metric values, is foreseen and required to identify strengths and weaknesses (R_{Rat}) to derive measures for improving the software quality [VFN20].

Furthermore, by analyzing the advantageous software parts regarding the selected analysis goal, implementation parts and design decisions, which positively impact the analyzed software quality characteristics, are identified. An example is the hardware-oriented structure of the warehouse software in *Case Study A*, which limits the effects of hardware variation points. When deriving recommendations for actions, these parts should not be changed as their modification poses a risk of turning them into less beneficial software parts regarding the analysis goal. Moreover, it needs to be considered that software parts, which are beneficial or disadvantageous for one goal and the related software quality attributes, might contradict others. An example is highlighted in [Neu⁺20a], where the conflict between reducing complexity by using OO IEC and its potential negative effects on performance efficiency is illustrated. In the example, applying OO IEC reduces software complexity by defining standardized module interfaces. At the same time, this leads to an increased cycle-time, i.e., a reduced performance efficiency, which is particularly disadvantageous for time-critical applications. Thus, a design decision that is beneficial regarding one software quality attribute might have a negative impact on another. Since the analysis is focused on a selected goal, this potential conflict of different software quality attributes should be taken into account during the assessment and subsequent derivation of recommendations for action.

Deriving recommendations for action

During the software assessment in *Case Study A*, two main findings regarding the enhancement of planned reuse are identified. These are subsequently used to derive recommendations for action. Targeting the software characteristic *planned reuse* (defined analysis goal) and its sub-characteristic *avoidance of duplicated software parts*, the first finding is a high amount of invariable, duplicated code parts across project variants. It indicates a weakness in the developed control software. However, despite the use of global variables, UDTs linked to these duplicated code parts are identified. To address this weakness and reduce the amount of copied software while enhancing planned reuse, the respective software parts should be merged and encapsulated in library POU. Thus, the derived recommendation for action regarding the first finding is to merge invariable software parts, including related variables, into library POU intended for planned reuse.

The second finding is that all possible warehouse variants can be identified with a suitable set of parameters. Moreover, a warehouse variant's control software is directly linked to the values of the chosen parameters [FVF15]. Targeting the defined analysis goal *enhancement of planned reuse* once again, parameter-based reuse of the variant-dependent software parts is proposed. More precisely, by separating the invariant and variant-dependent software parts, the configuration of the warehouse software is possible. The invariant software parts, which are intended for planned reuse as library POU's, are merged into a base project, i.e., a project template. From the analysis, the correlations among variable and optional software parts are known and, thus, can be specified in a configuration tool. With these actions performed, it is possible to develop the warehouse software of a specific variant by automatically configuring the source code of the variant-dependent software parts and adding them to the base project. Since the used PLC development environment does not yet support variant management and code generation from parameters, Excel is chosen as a potential starting point for an external configuration tool. In summary, the derived recommendation for action regarding *finding 2* is to use a parameter-based configuration to develop the warehouse control software, which partially builds on the first recommendation. Consequently, a stepwise enhancement of planned reuse is possible with the derived recommendations for action.

Generally, with the documentation from Steps 2 and 3, examples from the analyzed control software are available to point out implementation parts for illustrating the identified strengths and weaknesses. These can be used to derive recommendations for enhancing the software quality by addressing the identified weaknesses. As illustrated with *Case Study A*, linking the targeted analysis goal with the identified weaknesses supports deriving suitable measures to improve the software. Furthermore, approaches such as the goal-lever-indicator-principle can be used [VFN20]. The goal-lever-indicator-principle is based on the module model and module characteristics from [KfV04], who analyzed lever-action relationships between module characteristics and deduced a causal structure of three aspects, i.e., goal, lever and indicator. In the following, the principle's definition from [VFN20] is refined.

The principle defines a *goal* as a desirable quality characteristic of the control software concerning its modularity. In the scope of this thesis, the goal corresponds to the selected analysis goal derived in Step 1 for quality assessment and is not limited to modularity. Similar to the ISO 25010 [ISO25010], desirable software characteristics can be further divided into sub-characteristics, representing sub-goals. Means of influencing quality characteristics and, thus, a selected goal are called *levers*. More precisely, levers represent measures, e.g., design decisions or guidelines on different granularity levels concerning the overall software architecture or individual POU's. An industrial expert can take or follow these to improve the software to fulfill the desired goal to a greater extent. Thus, if a goal has not been reached entirely yet, a lever is the connection to derive

recommendations for action [VFN20]. Finally, *indicators* represent the quality-related software attributes measured during static code analysis to evaluate the selected analysis goal. Thereby, to measure and assess a software's quality characteristic fully, a set of attributes is often required [ISO25010] and, consequently, a single indicator is not sufficient. The relationship between goals, levers and indicators is illustrated with an example from *Case Study A* in Figure 15.

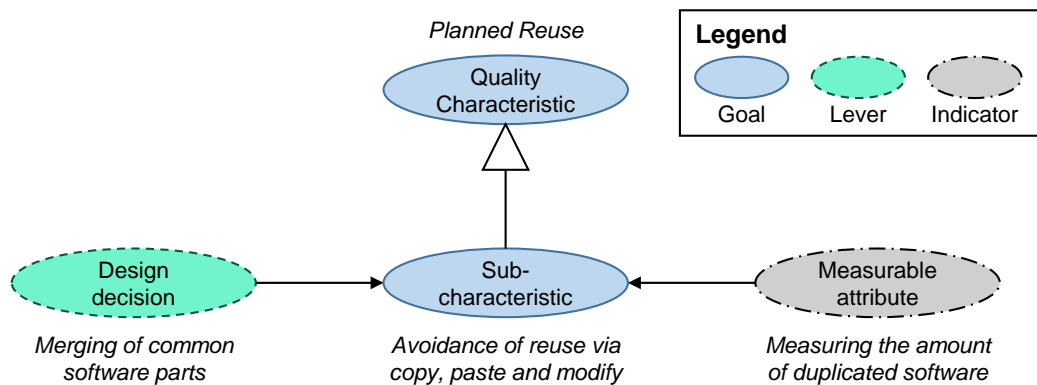


Figure 15: Goal-lever-indicator-principle at the example of planned reuse (adopted from [VFN20]).

In the depicted example, the targeted analysis goal is the quality characteristic *planned reuse*. One of its sub-characteristics is the avoidance of reuse via *copy, paste and modify*. A lever for this sub-characteristic is the design of reusable software modules by merging common parts. The extent to which the sub-characteristic is met can be assessed by measuring the amount of duplicated software (indicator) with approaches from static code analysis. Thus, the indicators are targeted by conducting static code analysis or calculating software metrics to assess the selected goal or quality characteristic. If a weakness is identified within the analyzed software, the lever represents the starting point for deriving a recommendation for action from the causal structure of goal, lever and indicator. Thereby, boundary conditions like used platforms, the current development workflow and deliberate design decisions should be considered (R_{Work} , R_{DD}). Furthermore, current software characteristics and the developer's background should be taken into account since the choice of suitable software concepts like reuse strategies highly depend on these [Fis+21c]. When deriving recommendations for action, conducting expert workshops is helpful as it provides background knowledge concerning the analysis goal. For example, when reuse is to be improved by merging code duplicates into mature library modules, the properties that a mature library module should satisfy and the typical design errors in the process should be known.

For documenting the derived requirements in an understandable format, the goal-lever-indicator-principle proposes a table-based structure with different criteria, as depicted in Table 11 [VFN20]. Besides the recommendation itself, additional information about its context is documented. For

example, the design principle targeted by the recommendation, including the related quality attribute, is stated. Moreover, the recommendation’s scope, i.e., the affected parts of the control software, including its functionality (optionally classified as proposed in [Wil+22]), and the expected benefit after performing the recommendation are documented.

Table 11: *Criteria of a recommendation for action derived from static code analysis results during procedure Step 4 (*representative list without claim for completeness); enlarged from [VFN20]*

Criteria	Characteristics of the criteria
Recommendation	Summary of recommendation listing actions suggested for enhancing the selected software quality attribute.
Design Principle*	Design decision, architectural aspect (e.g., as defined in [Neu+20c]), or guideline targeted by the recommendation. Examples include data exchange, modularity and hierarchy levels (link to analysis goal).
Quality Attribute	Addressed software quality attributes, e.g., according to the DIN EN ISO 25010 [ISO25010] (corresponds to the defined analysis goal)
Scope*	PLC project part affected by the recommendation: <ul style="list-style-type: none"> • Entire project (all POU’s and their interrelations) • All POU’s • Specific software parts (e.g., single PRGs, FBs, FCs, actions or groups thereof) • POU’s on specific hierarchy levels • Standardized POU’s in self-defined libraries • POU’s implemented in a particular language
Functionality*	Implemented functionality targeted by the recommendation, e.g., <ul style="list-style-type: none"> • Independent of functionality • Reusable functionality (potentially present in different POU’s) • According to the classification proposed in [Wil+22], including among others <ul style="list-style-type: none"> ▪ Sequence control ▪ Control of automation hardware (sensors and actuators) ▪ Communication (different types) ▪ Extra-functional software parts, e.g., <ul style="list-style-type: none"> ▪ Diagnosis and error handling ▪ Change of operation mode ▪ Connection to HMI ▪ Operating Data Collection
Reason	Explanation of how the recommendation supports achieving the selected goal.

The goal-lever-indicator-principle supports presenting the derived recommendation for action to the industry experts systematically, as suggested in Table 11. With this, the expert shall be enabled “to comprehend the analysis result and possible measures to be taken to improve the software based on the identified weaknesses” [VFN20]. While static code analysis and software metrics enable the assessment of control software regarding the selected goal, the principle goes one step further beyond the pure identification of weaknesses by detecting potentials for improvement concerning a defined goal and providing hints for achieving this goal. The direct use of the analysis

results is also the aim of procedure Step 4. The referenced principle is one possibility to relate the selected goal and the obtained analysis results to derive concrete measures for improvement.

Effort estimation for implementing derived recommendations

To estimate the required implementation effort, the derived recommendations for action and the documented results from the static analysis are consulted. From the static analysis in *Case Study A*, it is known that the control software can be divided into variant-dependent and invariant parts, including detailed documentation for all elements contained in the analyzed variants. The implementation effort for the first recommendation (planned reuse of invariant software parts) is rated low, especially since it can be realized incrementally: invariant POUs can be designed as library POUs stepwise and independent of each other. Further, the recommendation is expected to improve the control software quality by easing bug fixing or software maintenance and reusing well-tested library modules. Moreover, the software development time is reduced since the software developer does not have to modify these library modules concerning the hardware variability.

The second recommendation for action can be implemented in various ways. First, the base project can be defined as a template for new development projects. Thus, *copy, paste and modify* is already reduced to the variant-dependent software parts. Further, with means for parameterization, selected variable POUs can be revised into configurable POUs. The required link between the hardware variation points and the effect on the control software is documented in detail (cf. Step 3, Table 9) and directly useable for the revision. However, from the interviews, it is known that changes to the POUs implementing communication functionalities are critical. Consequently, it is advised to start with POUs with other functionalities. For a complete implementation of the recommended configuration in an external tool, a high effort is required. Since it can be achieved incrementally with direct benefits and the influence of the individual variation points on the software is limited, it is still considered feasible. Moreover, by using a configuration tool to develop the control software, the method *copy, paste and modify* and its disadvantages are entirely avoided. Since the chosen set of parameters is sufficient to describe a warehouse variant unambiguously, it represents a suitable basis for configuration and variant management on incremental levels.

In general, not all derived recommendations are equally suitable for enhancing the control software regarding the analysis goal. Therefore, the expected benefits, the estimated change effort, and risks of changes related to each of the derived recommendations for actions should be considered to select the ones to be implemented (R_{Weak}). In combination with the analysis documentation, levers can be used to estimate the effort of required changes linked to a recommendation of action. More precisely, the documentation provides a rough estimation of the scope of a planned change. For example, it illustrates if a recommendation solely affects the implementation within single POUs,

interfaces of a group of POUs need to be changed or an architectural design decision, e.g., regarding an extra-functional aspect, requires modifications. Thus, the documentation provides insights into the number of POUs and the connections between POUs affected by a planned change. Additionally, code examples for disadvantageous software parts are available from the analysis results to support the effort estimation.

Addressing some of the identified weaknesses might cause much effort, while only a slight improvement is expected. In these cases, it is not feasible to implement these in the first step. Moreover, in some cases, not all POUs affected by a recommendation for action need to be changed at once. As illustrated in *Case Study A*, incremental implementation of the recommendation can already be beneficial. In another example, intending to increase comprehensibility, the identification and subsequent refactoring of the POUs assessed as the most complex has the highest effect and is also incrementally possible [Fis⁺21b]. However, refactoring of POUs, which have been identified as slightly too complex, will not improve the comprehensibility to a great extent. Consequently, refactoring these should not be chosen as a top priority. Concluding, it needs to be decided to what extent a recommendation for action should be pursued.

Depending on the software change, the potential risks of introducing errors or causing incompatibilities between software parts should be considered. An available approach at the POU level is presented in [VNF22], where a maturity assessment of control software is performed. Thereby, 69 different change types in the implementation of individual POUs are distinguished and classified into three criticality categories. If a planned change is related to a high risk of introducing errors, tests of the respective control software parts or even the entire software project need to be repeated, especially if module interfaces or central modules are affected. This retesting is particularly cumbersome in application sectors such as medical applications (MedTech), where control software requires time-consuming re-certification after a change to ensure legal regulations are met. After comparing the estimated effort, expected benefits and risks, a strategy for software quality enhancement needs to be chosen, i.e., the recommendations for action to be implemented should be selected. Also, a suitable time for their implementation needs to be determined, which requires taking the software development workflow into account (R_{Work}).

In some cases, the effort to implement derived recommendations for action outweighs the expected benefits by far. However, the gained insights about the strengths and weaknesses identified are still useable since they should be considered in the design phase of new software concepts. This way, positive design decisions can be transferred into the new concept, while decisions leading to disadvantageous software parts can be avoided.

Optional adaptation of software development process

With the derived recommendations for action, means for the step-wise enhancement of planned reuse of the warehouse software are identified. These need to be integrated into the development process, which requires adapting the *copy, paste and modify* process to a development workflow utilizing library modules and, subsequently, a project template or even software configuration. Since only one developer is responsible for the warehouse software, separating module and application development into two departments is not feasible. However, quality gates should be introduced to ensure the maturity of new library POUs and the project template. Since the software developer has basic background knowledge about these reuse strategies, no additional expert workshop is required. Details of the software development workflow with a configuration tool are described in [FVF15]. The identified variation points in the warehouse hardware could potentially be used during the entire warehouse development process. However, it is sufficient to adopt the software development workflow to code configuration as a starting point. For this, no additional information from other disciplines is required since the MVL-list and layout plan are sufficient.

Case Study A demonstrates that apart from updating the control software itself by changing structures, selected sub-parts or interfaces, additional modifications might be required when performing a selected recommendation for action. Depending on the scope of the selected recommendation for action, changes include updates of currently applied reuse strategies, e.g., modifications of templates or existing library modules. Also, organizational aspects are targeted, depending on the recommendations for action to be implemented, which might result in adaptations of the development process or specific tasks performed during the software development. If the recommendation for action requires taking a new design decision or updating an available template, modifying programming guidelines might also be necessary.

In summary, in Step 4, the quality assessment of the control software regarding the selected analysis goal is performed based on the documented analysis results from previous steps. From the identified weaknesses, an approach like the goal-lever-indicator-principle is applied to derive recommendations for action to improve the control software concerning the analysis goal. The detailed documentation from Step 3 enables the effort estimation of the suggested improvements and supports the choice of actions to be performed. Thus, the procedure aims to support the industrial experts in using the gained insights from the quality assessment and identifying potential measures to improve the detected weaknesses. The result of this step is a concept with changes planned to be performed to enhance the software quality.

Summary concerning the entire quality assessment procedure

The presented quality assessment procedure is neither intended as a strict guide to be followed nor are its four procedure steps and the tasks performed therein always perfectly separable. In contrast, multiple iterations through the steps targeting sub-aspects of the selected analysis goal or even a mixture of the activities of different steps during the quality assessment are possible. The proposed procedure supports conducting the quality assessment in a structured, systematic and goal-oriented way by providing an overall schema and additional material, e.g., interview guiding questions and a checklist of aspects to be considered during the analysis. However, it has to be noted that all presented lists, e.g., the interview guiding questions, potential analysis goals and means of visualizing the results, have no claim to completeness. Instead, they are intended as examples to enable software developers to perform the quality assessment themselves.

The initial application of the proposed quality assessment procedure should be performed in close cooperation with the company's software developers or other involved personnel (depending on the selected goal). After conducting an analysis supported by external experts familiar with static code analysis, the company's software developers should subsequently perform the software analysis themselves. Thereby, ideally, the software developer conducting the analysis is not the one who has programmed the software but a colleague from the same department and machine type. A developer, who is not familiar in detail with the software being analyzed, has an outside view of the software. Thus, it is more likely that the developer questions design decisions or historically grown software structures during the analysis. Moreover, if the following analysis targets the same control software, familiarization with the software architecture and identification of an analysis goal are less time-consuming than during the first-time application of the procedure. Consequently, the overall application effort of the procedure is lower the second time (R_{Eff}).

6. Implementation

This Chapter gives a brief overview of the implementation of selected concept parts as support for a context-sensitive analysis using the proposed procedure and for documenting the analysis results. Most implementations are contained in the prototype of the research project *advacode* [Ins22], which was implemented by the programmer Thomas Mikschl and the student assistants Fabian Haben and Jan Wilch according to the concepts and specifications of the author of this thesis.

6.1. Prototypical Implementation of Context-sensitive, Configurable Static Code Analysis Concepts

In the research project *advacode* [Ins22], various means for an automatic static code analysis of Siemens TIA Portal control software have been implemented in a prototype. The input files for the *advacode* prototype are Siemens TIA XML files of the considered source code, which can be exported from the TIA Portal via TIA Portal Openness [Sie22]. In a preparation step, the files are parsed into an internal data model, including information on the contained elements, e.g., POUs, DBs and UDTs, and the dependencies between them, e.g., call and data exchange edges, for subsequent analysis. After this preprocessing step, various analysis means are available, ranging from call graphs, the calculation of software metrics and the identification of duplicated software parts. Selected analysis means, which are closely related to the quality assessment procedure, are shortly introduced in the following.

The proposed assessment procedure sets the framework for a goal-oriented, context-sensitive static code analysis, taking company-specific boundary conditions and deliberate design decisions such as programming guidelines into account (R_{DD}). An approach to enable the automatic identification of dependencies violating company-specific programming guidelines has been proposed in [Fis⁺22b] and implemented as a *Data Exchange Analysis* view in the *advacode* prototype. First, a classification scheme was introduced to distinguish different types of dependencies between software parts. Based on the classification, an editor to enable the configuration of rules was implemented to evaluate the concept with industry-sized control software (R_{DD}).

The *Data Exchange Analysis* view consists of three main parts, which are depicted in Figure 16. The available rules, which can be activated for use in the subsequent analysis, are listed on the left. After the analysis is conducted, the identified violations are displayed in table format on the right. The rule configuration is performed in a separate editor, which allows defining two types of rules, optionally including regular expressions (Regex) to enable filtering for or excluding specific

elements and taking company-specific naming conventions into account. Configured rules can be stored, exported from and imported to the *advacode* prototype and edited. [Fis+22b]

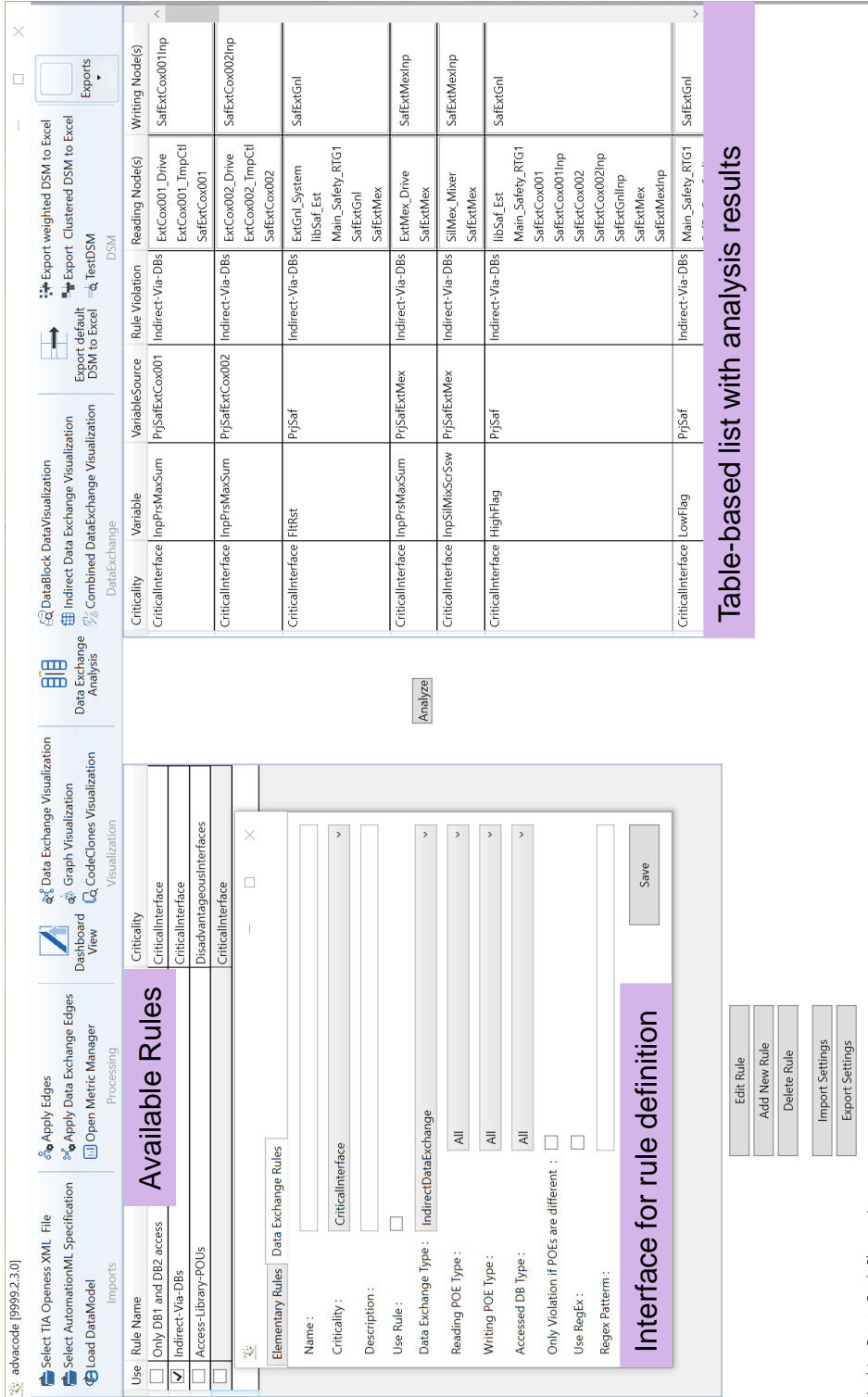


Figure 16: Screenshot of the data exchange view in the *advacode* prototype, including available, pre-defined rules (top left) with criticality level, rule editor (bottom left) and the table-based results view (right); content published in [Fis+22b].

When assessing if dependencies are implemented as intended, considering the functionality implemented in the dependent POU's is essential. Moreover, the functionality distribution in a software project is relevant for the reuse of its sub-parts (e.g., separation of application-specific control logic and standardizable hardware control). Currently, the identification of implemented functionality of individual POU's and, thus, the functionality distribution within a control software project, is mainly a manual task that requires understanding the intention behind the control code (R_{Rat}). A first approach enables the automated identification of the main functionality of individual POU's within a software project based on structural and semantic factors. These factors include the used programming language, a POU's hierarchy level in the call graph, complexity and naming conventions (cf. [Wil⁺22] for details). It has been implemented in the *advacode* prototype and evaluated with industry-sized control software [Wil⁺22]. Although the implementation does not substitute the manual comprehension of the functionality distribution, it eases its understanding (R_{Rat}).

Similarly, the *advacode* prototype contains different metrics, which can be calculated for all or selected POU's inside a software project. The metrics results are displayed in a sortable table. However, a manual interpretation of the gained metric values is required to assess the control software and derive recommendations for action from the analysis results (R_{Rat}).

6.2. Different Visualizations of Static Code Analysis Results

The results gained during the static code analysis in the *advacode* prototype are visualized in different formats. While table-based forms summarize metric results and identified violations, call and data exchange graphs with highlighting and filtering options are utilized to show the overall software structure or selected details in their context. For example, the results of the functionality distribution analysis are visualized color-coded in the call graph [Wil⁺22] (cf. Figure 17).

Apart from call graphs with color-coded functionality indications, additional color-coded information is available, e.g., the folder each POU belongs to in the development environment to analyze dependencies between project parts structured in folders. Moreover, the results of different software metrics can be visualized as the diameter of the nodes in the call and data exchange graphs. A *Combined DataExchange Visualization* view that contrasts the direct data exchange in the call graph and the indirect data exchange via global variables (in Siemens TIA Portal, i.e., via global DBs) was developed to support the dependency analysis. The *Combined DataExchange Visualization* view is depicted in Figure 18. It helps to understand the analysis results, as it displays the identified dependency violations and the affected POU's within their context, which is essential [GC15]. Furthermore, the amount of data a POU reads from or writes to global DBs is encoded as thickness in the edges between POU's and DBs.

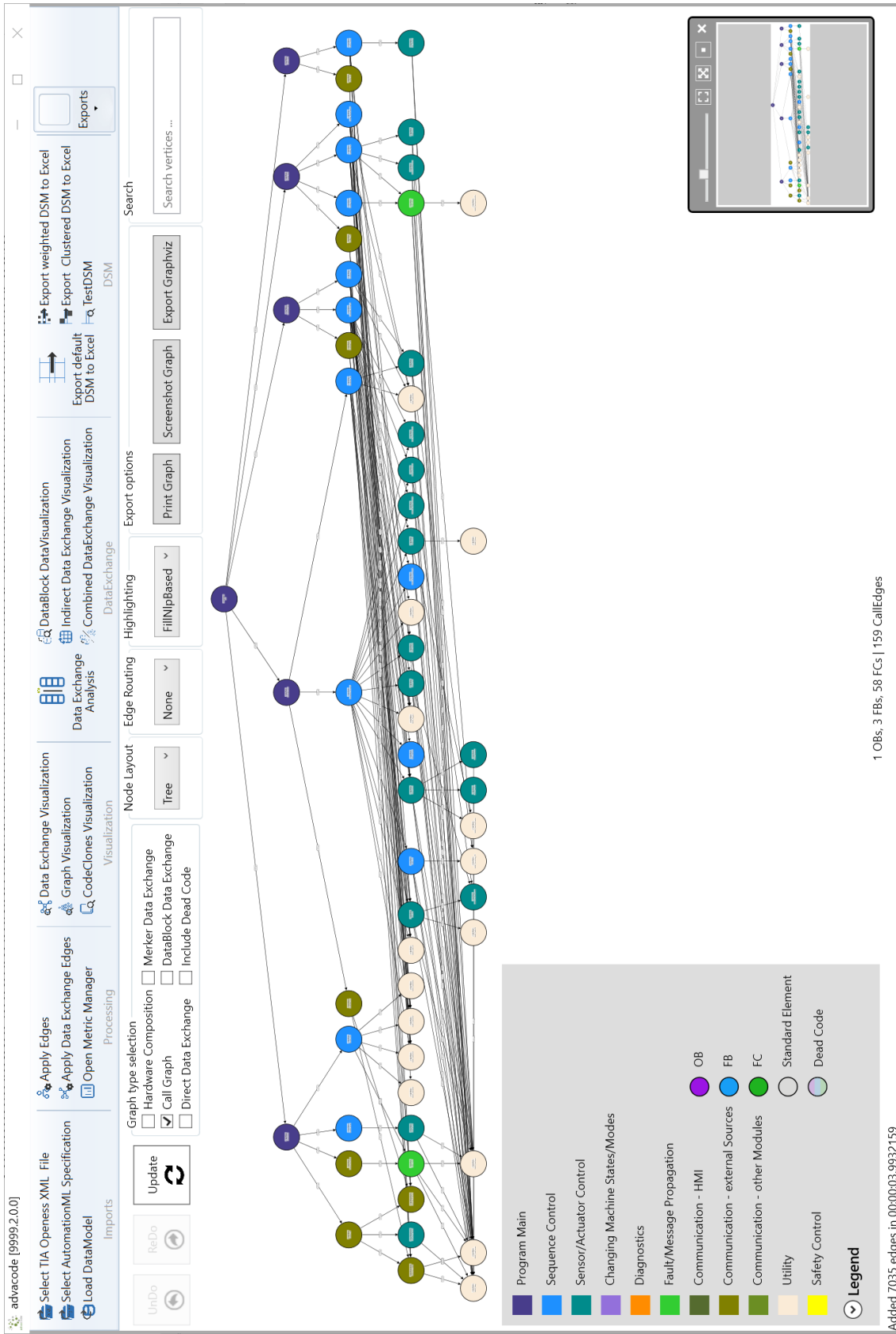


Figure 17: Screenshot of a call graph, including color-coding for the implemented main functionality of each POU in the advacode prototype.

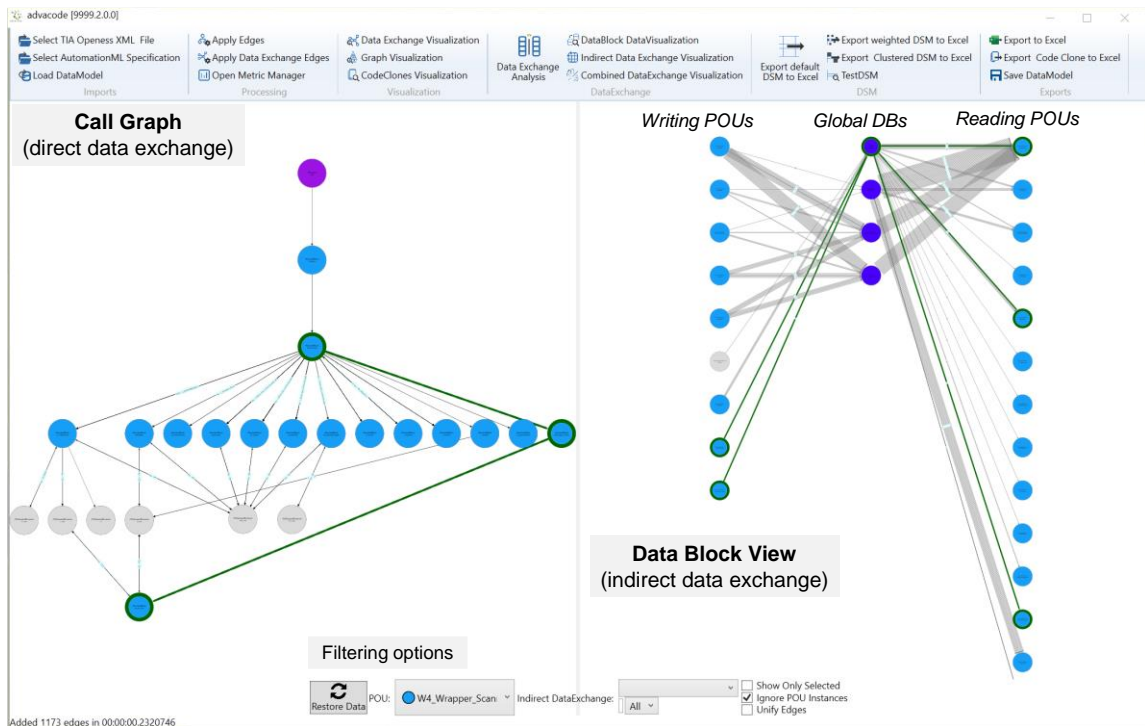


Figure 18: Screenshot of combined view with call graph (left) and data exchange via data blocks (right); dependencies to selected POU “W4_Wrapper_Scan” are highlighted in both views (content published in [Fis⁺22b]).

Apart from performing static code analysis, the *advacode* prototype also supports visualizing and exporting the gained insights, e.g., call and data exchange graphs, table-based lists of metrics results and violations of programming guidelines (R_{Doc}). Industrial experts, i.e., software developers, group leaders and managers from different companies, have confirmed the usefulness of the various documentation types. An expert confirmed that the combined view is promising to visualize the software’s modularity, but it is not yet convincing for an overall modularity assessment. One expert raised a concern regarding the suitability of the proposed call graph views to visualize OOP software, which has not been focused on in this thesis and needs to be investigated in future work. Different visualizations focusing on software variability have been prototypically implemented and evaluated in the DFG-funded project RED SPLAT [GEP22a]. These visualizations are tailored to the tasks of different stakeholders and show various granularity levels of the analyzed software project [Fis⁺20a].

The different analysis features of the *advacode* prototype have been applied to and evaluated with industry-sized control software in various studies (cf. [Fis⁺21b; Fis⁺22b; Pun⁺22; Wil⁺22] for details). Filtering options, e.g., in call and data exchange graphs [Fis⁺22b; Wil⁺22], highlighting possibilities, e.g., to display data exchange violations in their context [Fis⁺22b], and different visualizations, e.g., using a design structure matrix [Pun⁺22], support the assessment of the analysis results despite the size of industrial software projects (R_{Scal}).

7. Qualitative Evaluation

The proposed quality assessment procedure for legacy control software was evaluated using the requirements derived in Chapter 3. For this purpose, four different evaluation methods targeting different sub-sets of these requirements were used (cf. Table 12 for an overview).

- The overall applicability of the developed procedure was evaluated with three industrial *Case Studies B to D*, including industrial expert interviews (Sub-sections 7.1.1 to 7.1.3).
- Focusing on integrating the quality assessment procedure into different company workflows (R_{Eff}) and combining the proposed procedure with available means for automatic code analysis, the lab-sized demonstrator *Case Study E* was conducted. It was evaluated with a mixed industrial expert group using an online questionnaire (Sub-section 7.1.4).
- The suitability of the proposed quality assessment procedure to cope with different boundary conditions in the aPS domain and the procedure application by software developers to control software on different platforms, including adaptations to company-specific constraints, was targeted with an industrial expert group in the scope of an industry working group (WG) meeting. Due to their different backgrounds, the WG members are considered a representative group for the characteristics of industrial control software development. The evaluation included a web-faced questionnaire and industrial expert discussions in sub-groups (Section 7.2).
- Finally, the applicability of the procedure in an application sector with specific rules and regulations was evaluated in a workshop conducted with developers and group leaders from a German plant manufacturing company in the food and beverage sector. The evaluation targeted the conduction of the quality assessment and the use of the analysis results, including deriving recommendations for action, with an online questionnaire and industrial expert discussions (Section 7.3).

Table 12: Evaluation methods per requirement with reference to the relevant Sections.

Evaluation method	Case Studies					Expert evaluation in WG-meeting (applicability in aPS domain)	Expert workshop in food and beverage sector
	Industrial case studies	Case Studies			Lab-sized demonstrator case study		
Section	Sec. 7.1.1	Sec. 7.1.2 (Summary in Sec. 7.1.5)	Sec. 7.1.3	Sec. 7.1.4	Sec. 7.2	Sec. 7.3	
Evaluation Element	Case Study B	Case Study C	Case Study D	Case Study E	WG	Workshop	
Targeted Requirements							
<i>R_{PLC}</i> – Platform Independence	x	x	x	x	x (WG#13)		
<i>R_{Pro}</i> – aPS as Product	x	x	x	x	x (WG#3, 7)		
<i>R_{Use}</i> – User	x				x (W#8)	x (W#7)	
<i>R_{Sec}</i> – Application Sector	x	x	x	x	x (WG#11, 12)	x (W#4)	
<i>R_{PP}</i> – Pain Points	x	x	x	x	x (WG#6)	x (W#2)	
<i>R_{Work}</i> – Workflow Integration	x	x	x	x	x (WG#3)	x (W#8)	
<i>R_{DD}</i> – Design	x	x	x	x	x (WG#7)	x (W#3)	
<i>R_{Goal}</i> – Analysis Goal	x	x	x	x			
<i>R_{Scal}</i> – Scalability	x	x	x				
<i>R_{Eff}</i> – Application Effort	x	x	x				
<i>R_{Rat}</i> – Rationale	x	x	x	x			
<i>R_{Weak}</i> – Weaknesses and Change Effort	x	x	x	x		x (W#6)	
<i>R_{Doc}</i> – Documentation	x	x	x	x		x (W#5)	

Legend:

x: requirement targeted by evaluation, empty cell: requirement not targeted by evaluation;

CS: case study; Q: questionnaire; E: feedback from industrial experts in interviews/discussions; WG/workshop questions referred to as WG/W#[question number].

7.1. Qualitative Evaluation with Case Studies

Four case studies have been conducted in addition to Industrial *Case Study A* (cf. Chapter 5.2) for evaluation. Three case studies target industrial control software from German machine and plant engineering companies; one utilizes a lab-sized demonstrator and is assessed with an industrial

expert group. An overview of the conducted case studies is provided in Table 13 (in contrast to Table 2 on p. 17, only aspects regarding the analyzed control software projects are listed).

Although the proposed quality assessment procedure has been applied in five case studies, *Case Studies B* and *C* are focused mainly. Apart from *Case Study A*, these were the first applications, and thus, the most insights about the procedure were gained. Before introducing the details of the case studies in the following Sub-sections, *Case Studies B* and *C* are contrasted in Figure 19.








Procedure Step	Case Study B (independent project variants of a plant part)	Case Study C (dependent versions of a software project)
 Step 1) Familiarization with application	Expert interviews: <ul style="list-style-type: none"> - Machine functionality and general hardware structure - Development workflow (copy, paste and modify) Analysis goal: Documentation of software variants for their planned reuse (incl. hardware influences)	Expert interviews: <ul style="list-style-type: none"> - Machine functionality and general hardware structure - Development workflow (template, library modules) Analysis goal: Assessment of project template regarding modularity, comprehensibility and suitability for reuse
 Step 2) Static code analysis (manual & automatic)	Aim: Understanding structure of control software by <ul style="list-style-type: none"> - Identification of design decisions - Analyzing the link of software parts to the controlled hardware and functionality Analysis focus: Control of mechanical modules , including process logic (interlocks) , e.g., <ul style="list-style-type: none"> - Functionality distribution in the software - Dependencies in the software, e.g. interlocking conditions 	Aim: Understanding applied modularization principle <ul style="list-style-type: none"> - Link of software modules to hardware and functionality - Familiarization with programming guidelines and library modules Analysis focus: Understanding software structure and dependencies in final version of selected project <ul style="list-style-type: none"> - Analyzing data exchange between modules - Starting point: Central process, extra-functional tasks
 Step 3) Analysis of additional software parts including documentation	Goal: Modularization and reuse concept	Goal: Assessment of project template's reusability
 Step 3) Analysis of additional software parts including documentation	Analysis focus: How does change in hardware / layout affect control software?	Analysis focus: Which changes from template to final version are performed and when are they implemented?
 Step 3) Analysis of additional software parts including documentation	Additional projects: variants of considered plant part	Additional projects: version history of project from Step 2
 Step 3) Analysis of additional software parts including documentation	Method: <ul style="list-style-type: none"> - Mapping of hardware variants to control software - Documentation of variability (hardware, software and functionality) in feature models - Analysis of data exchange between software modules via flag variables (including type of passed information) - Continuous discussions with software developer to ensure correctness of findings - Structural patterns (not helpful due to flat call hierarchy) 	Method: <ul style="list-style-type: none"> - Analysis of changes from template to final version - Focus on data exchange of selected modules - Documentation of deviations from template - Analysis of programming progress (implementation additions, including violations and their consequences) - Classification of violations to analyze to which extent they could be automatically identified using static code analysis
 Step 4) Quality assessment	Assessment results: <ul style="list-style-type: none"> - Suggestions for modularity concept (restructuring of control software) - Reuse strategy with invariable and variable parts - Documentation of variability and link between functionality, hardware and software 	Assessment results: <ul style="list-style-type: none"> - Evaluation of the project template - Classification of violations (metrics for their identification) - Recommendations for action to improve usability of template

Figure 19: Comparison of Case Study B and C linked to the four quality assessment steps.

Details of the conducted *Case Studies B* to *E* are presented in the following Sub-sections. In Sub-section 7.1.5, insights gained from all case studies are summarized and linked to the derived requirements.

Table 13: Overview of conducted industrial Case Studies A, B, C and D and lab-sized demonstrator Case Study E.

	Industrial Case Study A (internationally operating)	Industrial Case Study B (internationally operating)	Industrial Case Study C (internationally operating)	Industrial Case Study D (internationally operating)	Lab-sized Demonstrator Case Study E
Product/aPS type	PM	PM	MM (SPM)	PM (with SPM)	Lab-sized demonstrator
Application sector*	Woodworking	Intralogistics	Automotive engineering (supplier)	Automotive engineering (supplier)	Intralogistics / factory automation
PLC development environment*	Siemens SIMATIC Manager (STEP 7)	Siemens SIMATIC Manager (STEP 7)	Siemens TIA Portal	Siemens TIA Portal	Beckhoff TwinCAT 3
Company Size	About 1,970 employees in total	About 600 employees in total	About 1,100 employees in total	About 800 employees in total	n.a.
Software developers*	About 30 software developers for PLC, HMI and static process control	About 20 PLC software developers (also work as commissioners)	About 30 PLC application developers; three PLC module developers	About 35 PLC module and application developers	n.a.
Amount of software projects considered (incl. number of POU(s))*	Five complete, independent software projects controlling variants of a plant part (already operating or shortly before commissioning); projects with about 130 to 180 POU(s)	Seven complete, independent software projects controlling variants of a plant part (five considered in detail); projects with 33 to 51 POU(s) and 363 POU(s)	Two complete customer projects shortly before commissioning; Final versions of projects with 225 and 363 POU(s)	Two complete, independent customer projects (187 and 135 POU(s)). Two training projects for new software developers to familiarize with the standard (base project: 34 POU(s), example project: 107 POU(s))	Four versions (evolution scenarios), which contain variants of hardware elements, e.g., conveyor belts; Projects with 19 to 21 POU(s)
Applied reuse strategies*	<i>Copy, paste and modify</i> of historically grown legacy software (developer reuses own projects)	1) Mainly <i>copy, paste and modify</i> of historically grown legacy software (developers reuse own projects) 2) Few standardized POU(s) (with version control, no libraries)	Project templates combined with library modules (software development divided into standardized and application-specific development)	Template-based reuse in combination with library modules, code generation	<i>Copy, paste and modify</i> of projects; OMAC state machines used on ISA 88 hierarchy levels
Used programming guidelines*	(x) very basic guidelines; company-wide unique equipment IDs	x structure, functionality distribution, naming conventions, reusable POU(s)	x structure, intended dependencies for data exchange, naming conventions	x mature standard, including an example project for a testbed	no company-specific guidelines; software is structured according to ISA 88 with OMAC actions
Information from interviews (Step 1) regarding pain points and current targets	1) Use of <i>copy, paste and modify</i> lead to historically grown software structures with little documentation on detailed software design decisions and influences of hardware variation points on the control software; only one software developer responsible for the plant part	1) Modules dependencies are crucial and mainly implemented indirectly; no suitable modularization strategy; functionality distribution is challenging due to variation points 2) Current targets: standardization, planned reuse, ease integration of reusable artifacts into overall PLC re-program functionality	1) Not all project-specific changes conform to the template, which leads to difficulties when updating the software, finding bugs or reusing modules in a different context and reduces understanding 2) Application engineers sometimes re-program functionality already	Programming standard and library modules are in use; the company currently aims to increase the reusable, standardizable software parts Standardization is challenging due to high customization	OMAC standard and ISA 88 used for software structure → expectation: software is modular; impression of a developer: despite recurring hardware, no reuse of respective control software (many code duplicates)

	Industrial Case Study A (internationally operating)	Industrial Case Study B (internationally operating)	Industrial Case Study C (internationally operating)	Industrial Case Study D (internationally operating)	Lab-sized Demonstrator Case Study E
	2) Current targets: documentation of variation points; planned reuse; training of a second programmer	software project, increase in modularity 3) Plan: function-oriented approach for company-wide use (ongoing)	implemented in library modules in new POU's, which are already included in library modules; 3) Current targets: Increase the proportion of reused software		
Analysis goal	Influence of hardware variants on control software (planned reuse taking variability into account; separation of application-specific and standardizable/parameterizable parts) modules	Documentation of available hardware functionality (incl. combinations) and its effects on the control software to develop a new modularization strategy and, thus, enhance planned reuse, e.g., with standardized library modules	1) Assessment of template: focus on interfaces to check conformance with guidelines in the final project and its version history; 2) Estimation of standardization potential in application software	Maturity assessment of programming guidelines, including base project and library modules, to assess the standard's modularity and improvement potential (focus on functionality distribution and interfaces between library modules and application-specific parts)	Identification of code duplicates and variants to derive library POUs for planned reuse
Design Decisions taken into account (R/D)	Organization of warehouse areas into DBs, data exchange with WMS	Timing requirements; few existing reusable POUs; connection to superordinate WMS	Data exchange via global DBs; specific POUs in each module dedicated to communication	Hierarchical modularization with dedicated FBs for controlling plant parts and extra-functional tasks; DBs as interfaces in PLC and to HMI	Project structure and development workflow considered for the definition of software metrics
Workflow integration	Decoupled from daily work; shortly before commissioning/after startup	Decoupled from daily work; commissioned projects after startup	During the development process and shortly before commissioning	Decoupled from daily work; base & commissioned projects after start-up	Decoupled from daily work; commissioned projects after start-up
Involved industry experts in quality assessment	Two software developers (one familiar with the control software, a new one under training)	Two software developers, a group leader, a manager (participated partially)	A module software developer, a manager	Two software developers, a group leader	A software developer; (ten industrial experts with different backgrounds)
Unique for case study	Effects of hardware variations included in software development assessment; development of parameterizable software parts	Modularity workshop enabled software developers to apply quality assessment independently to other software parts; hardware variants considered (feature models)	Applied at two stages of the development cycle; version history considered	Group leader confirmed helpfulness and relevance of documentation (programming guidelines and software architecture understandable for management level; enables risk assessment for changes)	Combination of manual and automatic static code analysis; ten industrial experts confirm that approach can be integrated into their company workflow

Legend: *: at case study time and concerning the analyzed software projects/parts; MM: machine manufacturing; PM: plant manufacturing; SPM: special purpose machines; WMS: warehouse management system.

7.1.1. Industrial Case Study B: Variability Analysis in the Intralogistics Sector

Industrial Case Study B was conducted in a German plant manufacturing company (R_{Pro}) from the intralogistics sector (R_{Sec}). A few peculiarities characterize this application sector: according to a recent study, the hardware modules, i.e., conveying elements, are highly reused [Spi⁺17]. These reused conveying elements have clearly defined interfaces to connect to neighboring modules and enable material transport. Due to this high degree of reuse, the control software can be modularized according to the used hardware, which results in a high reuse potential for the software (cf., for example, [AFV22]). *Case Study B* was conducted with an intralogistics part of a corrugated cardboard manufacturing plant, which transports product stacks from the cardboard manufacturing machine to further processing stations. For its control, mainly historically grown PLC software is used. The company used Siemens PLCs, programmed with the SIMATIC Manager (STEP 7) when conducting the case study (R_{PLC}).

Information on Procedure Step 1 with Case Study B

The **expert interviews** were conducted with three experts from the company, namely a manager, a group leader from the software development department and a PLC software developer. The experts briefly introduced the entire plant with a film and the targeted intralogistics part with mechanical construction plans. Apart from currently applied reuse strategies and the development workflow, pain points and recently targeted changes in the development were discussed.

The targeted plant part conveys produced cardboard stacks (cf. Figure 20). The entry points are so-called stacker stations, where the produced stacks are collected for subsequent transportation to a storage system, packaging machinery or other conveying elements. Additionally to the actual conveying area, the intralogistics system optionally contains a rework area to manually correct defects in individual product stacks. The amount of stacker stations varies and influences the layout of the intralogistics system. Each stacker station is connected to a straight conveying line called lane, whereby both have the same number. Furthermore, each lane consists of multiple numbered conveying elements, which vary significantly due to the functionalities they offer in addition to pure transportation. The used numbering system allows for determining a conveying element's position inside the conveying area. Depending on the desired functionality, several lanes can be combined into one or a lane can be split into several parallel conveying elements in a so-called special conveying area (SCA, cf. Figure 20).

The selection of conveying elements, which depends on the desired functionality and the production hall's layout, leads to a high variability of the considered intralogistics system, which also needs to be managed in the control software. During the initial interview, the experts stated possi-

ble combinations of two different conveying elements (hardware variations) influencing the control software. The variations in the hardware were mainly caused by the logistics functionality, e.g., pure conveying or conveying with combining, dividing, or turning the transported stacks.

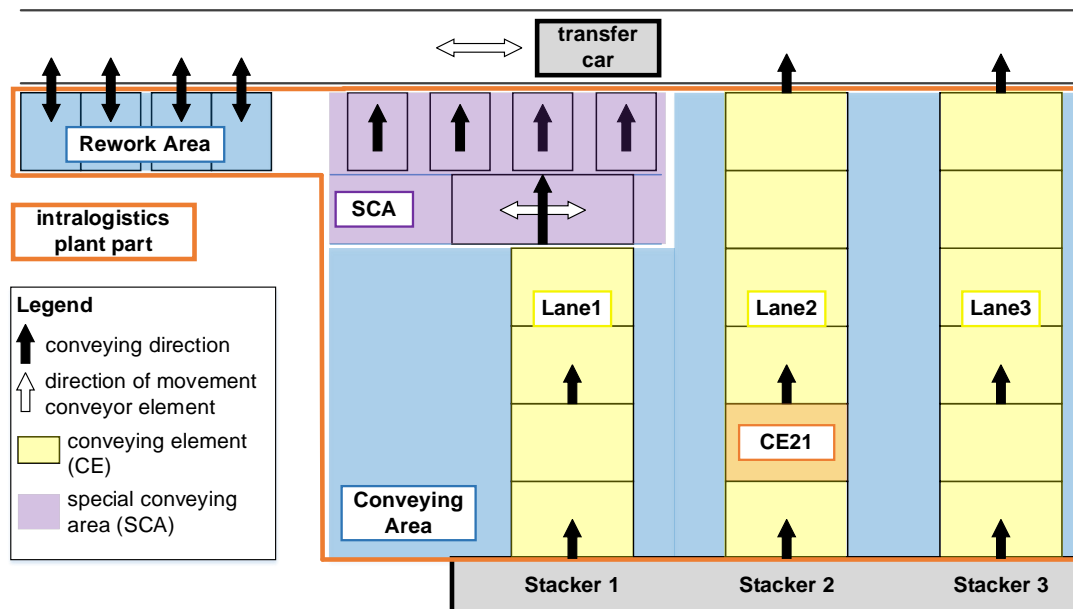


Figure 20: Considered intralogistics plant part of Case Study B, including interfaces to adjacent plant parts and organization in lanes consisting of conveying elements (CEs); adapted and modified from [Ber19].

As typical in the aPS domain, the control software's structure is highly influenced by the position and number of control panels, which varies, e.g., depending on the location of emergency stop switches. Each panel area supports two operating modes (manual and automatic) and, consequently, influences the required interlocks and the process logic in the respective control software. Standardly, the plant part is run in automatic mode, but manual control of individual conveying elements from an operating panel is possible. During the first interview, **the company shared various materials**. The mechanical layout plans of two plants, including the targeted intralogistics part, were provided. An image film containing the intralogistics system and the conveyed goods was also presented. During the analysis, further information was shared, including mechanical and functional descriptions of the conveying elements, excerpts from circuit diagrams and functional plans of the intralogistics system. Moreover, information on the company-wide, cross-disciplinary numbering system was provided.

When conducting the case study, the control software for the considered plant part is developed almost entirely using *copy, paste and modify* (cf. Figure 21). Despite the reuse of the hardware modules, which the company itself manufactures, very limited planned reuse of the respective software is performed. Only a few recurring functionalities are standardized in reusable FCs and

FBs. For these POU, documentation exists, but they are not organized in libraries. The software is programmed according to company-specific **programming guidelines**, including general remarks on the programming style, naming conventions and information on standardized, reusable POU. Moreover, the call structure and functionality distribution across POU, including memory areas of flag variables assigned to specific functionalities, are contained. Finally, the implementation of extra-functional tasks and different logistics functionalities are addressed. The legacy PLC programs are mainly implemented in FBD, with a few POU programmed in IL. The programming guidelines were provided for the analysis.

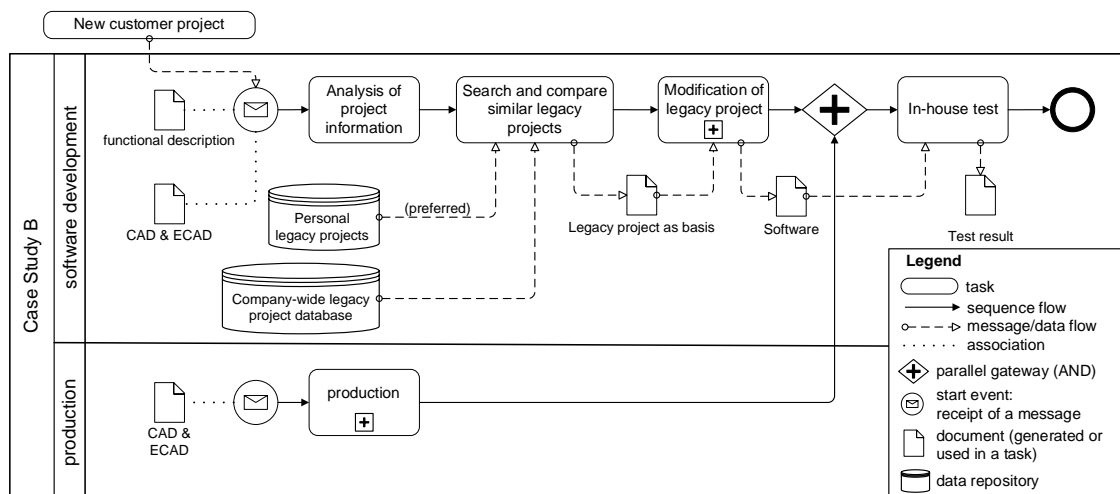


Figure 21: Software and product development workflow applied in Case Study B at the company site using BPMN.

The overall software structure follows a coarse-granular modularization on the level of lanes. The control of an individual lane is divided into several POU in a function-oriented manner. Thus, controlling a single conveying element requires more than one POU. Individual tasks such as the extra-functional task *error handling* are implemented by dedicated FCs. Extensive information exchange exists between the PLC software and a superordinate control, visualization and management system, which monitors the entire plant and coordinates the material flow. Process-wise, very strict timing requirements must be kept and the information exchange between PLC and the superordinate system is set (R_{DD}).

During the interview, the software developer and group leader mentioned the communication between modules as a known **pain point**. Overall, identifying a suitable modularization strategy as a prerequisite for enhancing the planned reuse of the control software is a challenge (R_{PP}). A modularization strategy for the control software must consider the high variability of the mechanical modules caused by their different functionalities. Currently, the company targets establishing a company-wide, function-oriented planned reuse approach. Since the influence of hardware variants on the control software is not documented, this is challenging. Moreover, interfaces between

modules and functionality distribution within the control software were stated as challenging. From these pain points, the analysis goal was defined: deriving a modular reuse concept for historically grown software variants to reuse recurring logistics functionalities and control variant-rich automation hardware (R_{Goal}). Based on the gained insights and after reviewing the received material, the analysis of the first project was planned and conducted.

Information on Procedure Step 2 with Case Study B

Following the proposed assessment procedure, the initial analysis targets the overall software structure and principle design decisions with the support of the software developer and provided material. With a focus on the set analysis goal, the link of the control software to the automation hardware is considered in detail. Consequently, the analysis is performed after completing the PLC project and start-up of the respective plant part (R_{Work}). The static analysis was started tool-based utilizing a prototype [Fuc⁺14] to generate the software's call graph and data exchange graphs (cf. Figure 22). However, the flat call hierarchy (cf. Figure 22, left) and the high amount of indirect data exchange via flag variables (cf. Figure 22, right) hampered the comprehension and the tool-based analysis was not sufficient. Thus, manual code analysis was conducted to understand the dependencies between the POU's (R_{Rat}). Iterative exchanges with the software developers were organized to clarify questions arising during the analysis.

The **first analysis targeted three different aspects**, which were analyzed sequentially from a coarse-grained to a fine-grained level following the analysis checklist (cf. Table 8, p. 65): initially, the aim was to gain an overview of the entire software project, including the contained POU types (cf. *Aspect 1*), their dependencies (cf. *Aspects 2* and *8*), structural design patterns (*Aspect 3*) and included library elements (*Aspect 4*) on a coarse level. To a great extent, this step was performed with the analysis prototype. Subsequently, the functionality distribution in the software (*Aspect 7*), including the link to controlled automation hardware and superordinate control system (*Aspect 10*), was analyzed manually. For this purpose, the layout plan and programming guidelines were used. Finally, since module communication, i.e., interfaces between the respective software parts, was mentioned as a challenge, it was analyzed in greater detail (cf. *Aspect 9*). An overview of the examined aspects and targeted findings is provided in Table 14.

Table 14: Overview of targeted aspects during the first project analysis in Case Study B.

Focus	Aspects and followed procedure	Targeted findings
Overall structure	<i>Aspects 1, 2, 3, 4</i> and <i>8</i> Contained POU's (including type) Call graph, indirect data exchange graph Used library POU's and structural patterns	Overall project structure (hierarchy levels) Overview of amount of direct and indirect dependencies between POU's Contained library elements

Focus	Aspects and followed procedure	Targeted findings
Link to automation hardware	<i>Aspects 6 and 10</i> Detailed, manual implementation analysis for 1) Identification of POUs linked to panel areas, superordinate system and others 2) Identification of implementation within POUs for controlling the conveying elements	Functionality distribution (on POU and sub-POU level) Link of control software parts (on network level) to controlled hardware
Communication between POUs	<i>Aspect 9</i> 1) Dependencies between POUs (indirect data exchange via memory areas of flag variables) 2) Analysis of according data flow	Amount and type of exchanged data Intention behind data exchange; reason for dependency

The first analysis results **were documented and visualized** using manually annotated call and data exchange graphs (including size metrics, cf. Figure 22), in a table-based format and, additionally, in a presentation as a combination of code screenshots and comments (R_{Doc}).

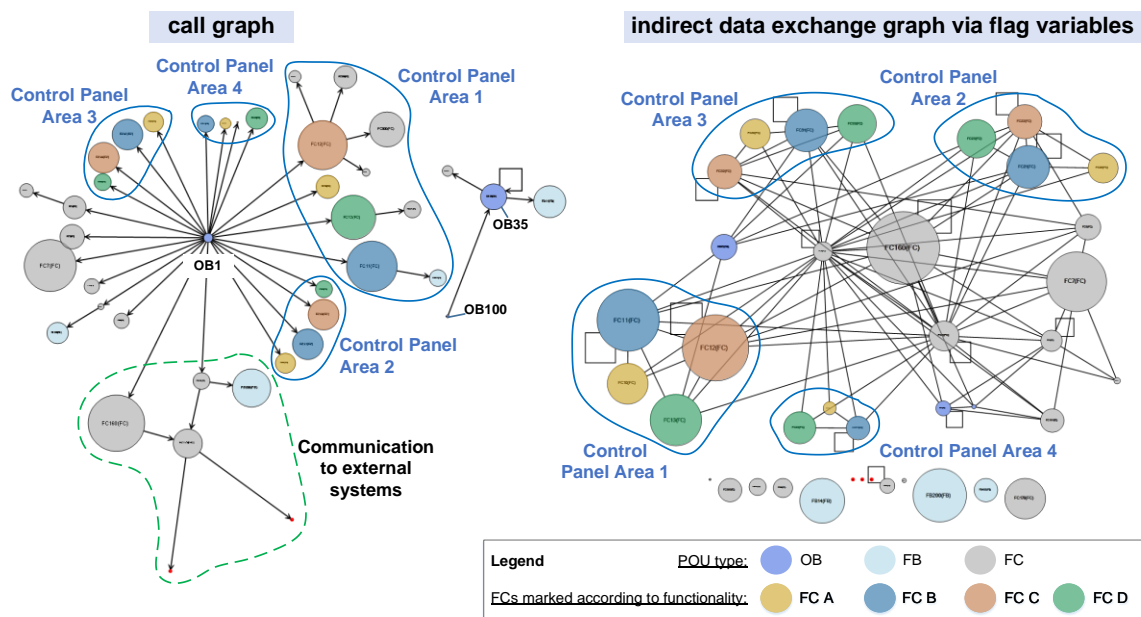


Figure 22: Generated, manually annotated call graph (left) and indirect data exchange graph via flag variables (right) of Case Study B's control software in Step 2; adopted from [Ber19].

The software structure in this case is relatively flat, with a maximum of four hierarchy levels. Apart from the tree pattern, no other structural patterns could be identified. The functionality is mainly implemented in FCs and, consequently, there is no reuse through multiple instantiations of a FB. However, some Siemens library POUs are used. Three groups of POUs were identified during the functionality analysis, namely general POUs, POUs belonging to a specific operating panel (cf. panel areas in Figure 22) and POUs for communication. Each lane is controlled by four FCs, modularized in a function-oriented manner. Sub-parts of these FCs can be linked to the control of individual conveying elements within the respective lane. The dependencies between POUs

are implemented mainly indirectly via flag variables. Although the control of different lanes is similar, the four POUs associated with a lane need to be adapted to the contained conveying elements and their offered logistics functionalities, which affect the required interlocking conditions.

The gained insights were discussed with the company's developers. Due to the pain points and challenges, it was decided to analyze further PLC projects focusing on the POU interfaces and links between the control code to the hardware modules. An overview of the dependencies and exchanged data is essential for the analysis goal, i.e., choosing a suitable modularization strategy.

Information on Procedure Step 3 with Case Study B

The software developers selected five additional **independent software project variants** of the plant part targeted in Step 2 to identify and document their variation points. Due to lacking tool support, it was planned to conduct a manual analysis. Thus, only a limited number of projects could be compared and a representative selection based on the developers' experience was essential. For ease of understanding, the variability analysis was started with a one-week stay at the company's site to enable clarification of questions on short notice.

According to the defined goal (*enhancing planned reuse*), the analysis of the additional projects was performed in two parts: first, the selected variants were compared regarding the contained conveying elements. Since variations in the automation hardware were expected to impact the respective control software, they were documented to consider them in the targeted modularization and reuse concept. Second, the functionality was analyzed as the company aims to apply a function-based modularization strategy across all disciplines. From a functional/hardware perspective, the identified common, alternative and optional parts were documented as feature models.

Subsequently, the effects of the identified hardware- and functionality variation points on the six control software projects were targeted on different granularity levels. These are summarized in Figure 23. A comparison of the call graphs showed recurring structural patterns, e.g., a group of POUs responsible for communication via the bus system (cf. Figure 24). However, only little variability was identified on the project and POU level. Thus, a manual analysis of the sub-POU level was required.

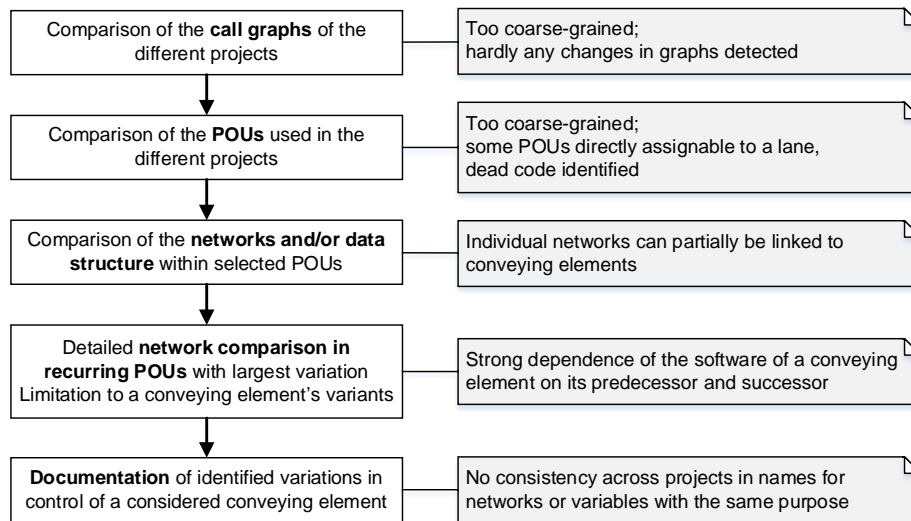


Figure 23: Performed steps during the automatic and manual control software analysis of project variants concerning influences of hardware variations (left), including gained insights (right).

The four FCs used for controlling the conveying elements in a lane are recurring multiple times in all projects. They have high variability concerning their implementation, but the used variables are almost identical. Thus, considering the company's naming conventions, these POUs are compared in a very fine-granular analysis on the network level. Some networks and their variants are directly traceable to hardware specifics or the required functionality. With a focus on POU parts linked to hardware control, the communication between POUs was targeted. According to the programming guidelines, it is mainly implemented as indirect data exchange via flag variables. However, the causes for the variations are not documented. Thus, it was analyzed which flag variables belong to which conveying elements, including their variations. As a result, the implementation parts and flag variables required to control individual conveying element variants were documented. By comparing the control software of identical conveying elements with different pre- and successors, it was noted that a conveying element's control software strongly depends on its physically connected neighboring elements, e.g., regarding interlocking conditions.

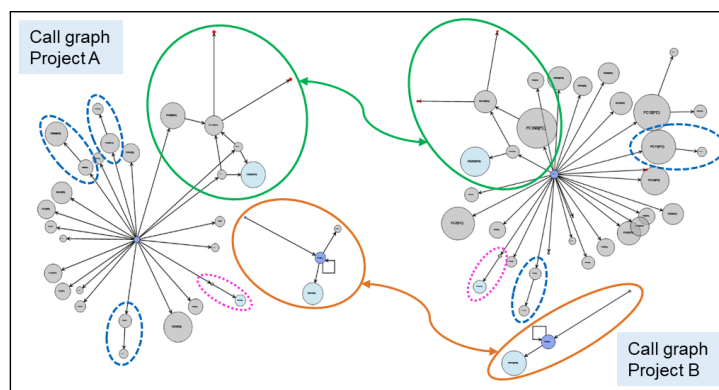


Figure 24: Comparison of two call graphs regarding prominent features and commonalities in their structure as a pre-step for their detailed manual comparison on the sub-POU level.

During the comparisons, some challenges arose, which required manual interpretation of the targeted control software (R_{Rat}). For example, despite naming conventions, FCs implementing the same functionality did not always follow a uniform numbering and naming. Similarly, naming conventions for network titles or variables with the same functionality were not used consistently across different projects, which required manual interpretation to identify variable and common parts (e.g., correspondent variables in different projects) in the comparisons. Moreover, some projects contained unused dead code, i.e., unused POUs resulting from a *copy, paste and modify* error, which created wrong dependencies within the software. Finally, the programming preferences and styles of different software developers lead to additional variants. These had to be eliminated before defining a new modularization strategy, which required the support of the software developers. Also, due to the high number of variants, the company experts' help was required to identify the actual variants, including their cause (R_{Rat}).

The identified variation points were **documented** with feature models from a functional and hardware viewpoint (R_{Doc}). The identified variation points affect the control software differently, mainly requiring software changes within existing POUs, i.e., changes on network level. These effects were documented on different granularity levels (POUs, networks and variables) using a table-based format. Also, the dependencies of flag variables to POUs and functionalities were reported in a table-based form. The resulting table provides an overview of the POU interfaces, including data required from other POUs (potential input variables), data provided to other POUs (potential output variables) and data on the material flow. Overall, the documentation is a basis for deriving a new modularization principle with clearly defined, standardized interfaces between the software parts controlling the conveying elements.

Information on Procedure Step 4 with Case Study B

During the quality assessment and derivation of recommendations for action, **deliberate design decisions** such as POUs that are already reused, the connection to the superordinate system and the strict time requirements were taken into account (R_{DD}) and integrated into the new concept.

Overall, the **analysis results**, i.e., the documentation of variability in functionality, hardware and control software, confirm a high potential for increasing planned reuse. The control software is already modularized; however, the level is too coarse-grained, which leads to **weaknesses** hindering planned reuse (cf. Table 15 for an overview). The identified three to a maximum of four architectural hierarchy levels in the control software indicate lacking encapsulation. The detailed implementation analysis confirmed that the hardware control and process logic, i.e., process-independent and -dependent software parts, of all conveying elements inside a lane are mixed in four POUs per lane. This mix reduces the reuse potential of those POUs since their reuse in a different

context requires application-specific adaptations (*WB-1*). However, a few POUs are already reused in a planned way. The second identified weakness is also closely linked to the applied modularization strategy. Currently, the data exchange between POUs is mainly implemented indirectly via flags, which requires much expertise to reuse one hardware module's control software, including all dependencies to neighboring modules, in a different context (*WB-2*). In addition, the use of flag variables is error-prone because read and write accesses are challenging to trace. Finally, the varying programming styles of different programmers lead to additional variants, which complicate the planned reuse of existing functionality implementations (*WB-3*).

From these weaknesses and the documented analysis results, **recommendations for action** are derived, including an **estimation of their effort** (R_{Weak}). The goal-lever-indicator principle is applied to address the weakness *WB-1* with a suitable recommendation. As depicted in Figure 25, the quality characteristic *modularity* and its sub-characteristic *separation of concerns* can be measured by analyzing the *functionality distribution*. A recommendation is derived from the related lever, namely the *separation of application-specific parts* such as interlocking conditions and *standardizable parts* like the control of recurring automation hardware.

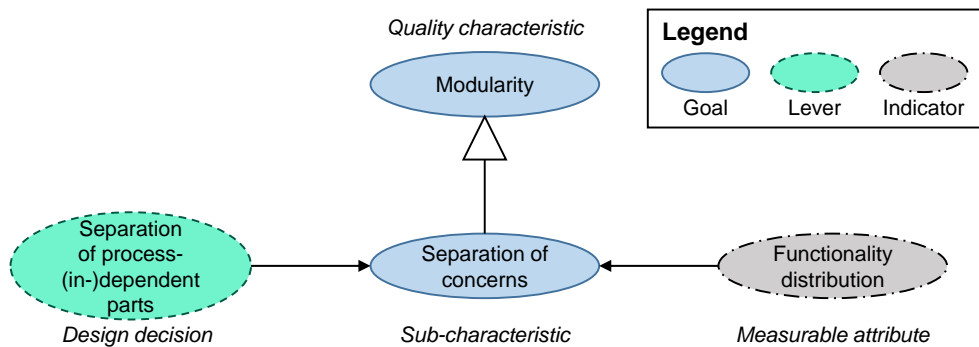


Figure 25: Goal-lever-indicator principle applied to the identified weakness *WB-1* to derive a recommendation for action (following the style of [VFN20]).

To achieve the recommended separation of process logic and hardware control, the followed hardware-oriented modularization strategy needs to be implemented more fine-granular on the level of conveying elements. This recommendation aims to enable planned reuse of the hardware control POUs of recurring conveying elements and limit the amount of POUs, which need to be adapted according to application-specific combinations and resulting interlocking conditions.

Regarding the *modularity's* sub-characteristic *loose coupling*, the *analysis of (in-)direct dependencies* between the POUs shows indirect communication between neighboring modules via flag variables (*WB-2*). Thus, reducing the dependencies and implementing them directly or via DBs is recommended following the lever *uniform and lean data exchange*. The use of flag variables should be avoided entirely. For combining and standardizing information to be exchanged between

POUs, the development of UDTs is an option to reduce interface variables. Finally, to address the weakness *WB-3*, it is suggested to strengthen the programming guidelines to avoid unnecessary variants by introducing uniform variable names, network titles and numbering for POUs.

Although the change efforts for addressing weaknesses *WB-1* and *WB-2* are estimated as high, since they affect the entire software structure, they are expected to lead to high benefits by enabling planned reuse. Moreover, the detailed documentation of the analysis results eases the process. As the activities go hand in hand, they can be carried out in parallel, decreasing effort. The resulting POUs for hardware control, if defined according to the new modularization strategy, can directly be implemented as library modules and, thus, address the defined analysis goal.

Table 15: Summary of identified weaknesses, derived recommendations for action and the estimated change effort in Case Study B.

Identified weakness	Recommendation for action	Estimated change effort
<i>WB-1</i> Functionality distribution according to lanes reduces reusability (modularization is too coarse-grained)	Separation of hardware-control and process logic; modularization of POUs on the more fine-grained level of conveying elements	High (change of entire software structure required for separation of concerns, but documentation is available from the analysis)
<i>WB-2</i> Indirect data exchange via flag variables	Redefine POU interfaces: direct data exchange; lean interfaces; elimination of flag variables	High (change of entire data exchange required; but: POU implementation parts and linked flag variables were documented)
<i>WB-3</i> Available programming guidelines in parts too vague	Revision in identified points leading to unnecessary, additional variants	Low (identified examples can be directly included)

With the derived recommendations for action, means for the step-wise increase of the software's modularity are identified. These need to be integrated into the development process, which requires adapting the *copy, paste and modify* process to a development workflow utilizing library modules. A new modularization concept was developed and prototypically implemented based on the identified weaknesses. In addition, a two-part modularity workshop was conducted with the company's software developers to provide them with the necessary background knowledge to implement improvements and new modularization strategies without support of external consultants. Apart from solution alternatives, the workshop included limitations and boundary conditions along the entire software lifecycle. In a short meeting after the quality assessment had been finished, the experts confirmed that they applied the principles discussed during the joint PLC code analysis and workshop to individual parts of a machine currently under development themselves (R_{Use}).

Lessons learned from Case Study B

The insights gained during the application of the quality assessment procedure in *Case Study B* are summarized with a link to the respective procedure steps in Table 16.

Table 16: Summary lessons learned from procedure application in *Case Study B*.

Procedure step & activity	Challenges	Required input	Lessons learned about the proposed quality assessment procedure
Step 1) Interview with experts	One interview was not sufficient; continuous feedback by experts was required	Domain knowledge was required during the analysis on different granularity levels (concerning functionality, hardware and software).	Expert interview in Step 1 is essential but not enough. Instead, continuous feedback loops with company experts are required to clarify arising questions during the analysis.
Step 2a) Static code analysis with tool	Due to flat call hierarchy, little knowledge gain with available code analysis tool (no structural patterns)	Tool could not identify aspects like: link of software to controlled hardware, intention behind software parts.	Available tools and automated means for static code analysis are not enough. <i>Case Study B</i> confirmed the importance of manual code analysis (cf. R _{DD}).
Step 2b) Manual analysis	Gaining overview and understanding of software architecture and relevant parts	Feedback/input from industry experts required to understand programming rationale.	Quality of the results after applying the procedure depends on feedback from industrial experts.
Step 2b) Manual analysis	Time-consuming task with unknown result (risky if the effort is worth the result)	Estimation of software developers regarding challenges in daily work to select software parts for detailed, manual analysis (closely linked to selection of analysis goal).	Due to the historically grown software structure with little planned reuse, improvement potential was expected and confirmed with analysis results. Expert interview(s) are essential for estimating if the manual analysis effort will pay off.
Step 3) Documentation of variability analysis	Linking commonalities and differences in software variants to their cause (for variability) could not be performed by static code analysis only	Documentation on hardware variants was essential to understand software variations. Dead code or errors due to incomplete <i>modify</i> -step cause false variants that require domain expertise.	Domain knowledge and information from other disciplines are essential to identify causes for software variability, e.g., variations in functionality, hardware, customized aspects and others. Finding: Software seldom varies due to programming style.
Step 3) Expert involvement in documentation	Software developers are not always familiar with means used to document analysis results (e.g., feature models for variability)	Software developers need to participate and be motivated to understand the means used for documenting analysis results.	Motivation of involved experts is crucial for long-term success (enable experts to apply procedure themselves) and for usefulness of generated documentation.
Step 4) Expert workshop	Industrial experts need to be enabled to understand the assessment and the gained insights and to draw the conclusions themselves	Two-part workshop with industrial experts targeting different aspects of modularity, including potential conflicting sub-goals of modularity.	Training of software developers is essential to enable them to apply the assessment procedure. Scope and content of training depend on the background of the involved experts.
Step 4) Assessment (and interpretation) of results	Background information about theoretical aspects is required to draw conclusions about the analysis goal	Workshop with software developers regarding modularity, which went beyond the considered example to provide general background knowledge.	Without the workshop targeting general background knowledge, the developers would not have been able to independently apply the approach to a different machine part. Limitation: If a different software/analysis goal is targeted, developers might lack required background knowledge.

The proposed quality assessment procedure was successfully applied in *Case Study B*. It confirmed the importance of expert interviews, domain knowledge and information from other disciplines to understand the intention behind the control software or identify reasons for variations. Furthermore, the involvement of industrial experts is crucial to enable them to apply the quality assessment procedure themselves successfully. However, the scope and structure of such a workshop require further research and need to be tailored to company-specific boundary conditions.

7.1.2. Industrial Case Study C: Analysis of Version History and Estimation of Reuse Potential in the Automotive Sector (Component Assembly)

The industrial *Case Study C* was conducted in an internationally operating German special-purpose machine manufacturer (R_{Pro} , cf. Table 13 for an overview). During *Case Study C*, the quality assessment procedure was applied to a machine for the automated assembly of products used in automotive engineering (R_{Sec}). The machine's control software is programmed in Siemens TIA Portal (R_{PLC}). Due to customer requirements, already developed machines, including control software, are usually not reusable without changes. The company follows a modular design principle to stay globally competitive and achieve a high degree of reuse. Accordingly, software development is divided into creating reusable, project-independent and project-specific software parts. Partially *Case Study C* has been published in [Fis⁺21a].

Information on Procedure Step 1 with Case Study C

In Step 1, two **expert interviews** were conducted with a software developer responsible for standardized, project-independent software parts and a manager familiar with the machines. During the interviews, the software developer briefly introduced the considered machine-type, applied reuse strategies and rough pain points. Further, he provided details of the machine controlled by the software project chosen for the quality assessment.

The selected machine automatically assembles two parts of a product, including several preparation and post-processing steps. The machine is divided into mechanical modules that provide functions regarding the workpiece processing or the workpiece flow. First, a robot loads the machine with raw parts. These parts are assembled on a turning table with several adjacent stations that perform different assembly processing steps. After each step, the turning table rotates to move the parts one station further. Consequently, it requires close coordination with the adjacent stations. In the end, the robot places the parts on pallets for further transport.

The company **shared material** on the operating concept of the machine, its mechanical layout plan, details on the individual mechanical modules and the performed processing steps to ease

Overall, the control software has a hardware-oriented, modular structure, e.g., a robot or a turning table are controlled by a software module each. The software modules are designed function-oriented and consist of several POUs, which implement specific functionalities [Fis⁺21b]. To achieve a high degree of modularity, the design specifies only a few to no calls between the software modules (R_{DD}). Instead, data exchange between modules is foreseen indirectly via global DBs. UDTs are used to standardize the exchanged information. Extra-functional aspects like communication to the HMI, error handling and safety are standardized in project-independent library POUs.

During the two interviews and email exchanges, the software developer mentioned **challenges and known pain points** in the workflow (R_{PP}). The project-specific template is provided to the application developers, who adapt it to the required, usually unique, machine functionality. Due to time pressure and customer requirements, violations of the programming guidelines, the intended project structure and the functionality distribution are introduced. Consequently, reusing new modules in a different context becomes very costly. However, this contradicts the company's plans to increase reuse steadily and, thus, decrease development time by standardizing parts from commissioned projects. During the second interview, this pain point was refined. The reuse of software modules is prevented by many dependencies on POUs of other modules, which contradict the intended, standardized interfaces. Consequently, the **derived analysis goal** is to examine the software development of the modular, template-based control software during project-specific adaptations to assess the template's suitability for the integration of reusable modules (R_{Goal}). Accordingly, changes made to the template during the development are analyzed, focusing on dependencies between the template, library modules and project-specific POUs.

In a short discussion after the first interview, the criteria for **selecting the first project** to be analyzed were jointly defined: the software developer must be familiar with the PLC project, it needs to contain application-specific adaptations, which are not implemented in an ideal way, and it should be (close to) finished to ensure all project-specific adaptations had been made and the overall structure and design decisions in the project are comprehensible. The software developer selected the final version of a recently finished project. During the project-specific software development, many challenges had to be resolved, which are estimated as representative by the developer and lead to many dependencies violating the company's programming standard.

Information on Procedure Step 2 with Case Study C

As intended by the assessment procedure, the initial analysis first targets understanding the overall software structure and principle design decisions with the support of the software developer and provided material. According to the software developer, most changes and difficulties in the project-specific adaptations occur in the modules and their dependencies. Therefore, it was planned

to focus on how modules are integrated into and adapted within the template. When analyzing the first project, several meetings and emails were scheduled and exchanged with the software developer to clarify questions arising during the analysis iteratively. This led to a **refinement of the analysis goal**: to assess the template's suitability, understanding the dependencies between modules is targeted, whereby data exchange within the modules was not considered important. More precisely, it was decided to examine if dependencies are implemented as intended or contradictory to the design principle. To assess the refined analysis goal, the software developer recommended focusing on the turning table due to its central character and high dependencies on the adjacent stations. Further, the coupling of the robot to the turning table was highlighted since it requires a relatively complex sequence, which is known to be challenging.

Following the set analysis goal, the **quality assessment** was planned **at two different points of the development workflow** (R_{work}). The first analysis is conducted after the software development to understand the software structure and assess the template's modularity and the application-specific parts, including their conformance to the programming guidelines, focusing on dependencies. The second part of the analysis targets the changes performed during the application-specific development process, which requires comparing different project versions. The aim is to identify if and when violations of programming guidelines are implemented.

During Step 2, mainly manual static code analysis was performed since, apart from a prototype in its early stages, there was no analysis tool available when the analysis was conducted. However, the folder structure in the TIA Portal project explorer, an in-program tool that lists the call hierarchy and the cross-reference data were used to obtain information on the software architecture. Corresponding with the analysis checklist (cf. Table 8, p. 65), the manual analysis is performed from coarse- to fine-grained aspects. Initially, the folders in the PLC project explorer and the POUs organized within these are analyzed to understand the overall structure, i.e., the division of the project into modules consisting of POUs (cf. *Aspects 1 and 5*). During this step, the naming conventions of the folders and software elements (e.g., POUs, DBs) are used to link the considered modules to the respective hardware and, thus, comprehend their functionality. Next, the call graph was analyzed with an in-program tool in the TIA Portal (cf. *Aspect 2*). Finally, the organization of module data was analyzed, targeting the structuring of the information, POUs exchanging data and identification of data exchange patterns across different modules (cf. *Aspects 3 and 8*). Concerning the refined analysis goal, the implementation of central functional processes was analyzed in detail. Furthermore, standardized POUs implementing central generic functionalities, namely selection of operation mode and error handling, were examined to understand the dependencies between the application-independent and the project-specific software parts (cf. *Aspects 6, 7 and 9*). An overview of the analyzed aspects and targeted findings is contained in Table 17.

Table 17: Overview of targeted aspects during the first project analysis in Case Study C.

Focus	Aspects and followed procedure	Targeted findings
Overall structure	<i>Aspects 1, 2, 3, 5 and 8</i> Contained elements (modules, POUs) Organization of elements in folders Call graph Indirect data exchange Structural patterns	Functionality distribution (incl. link to hardware) Modules (and POUs) intended for a planned reuse Direct dependencies between elements via calls Linking functionalities/elements to hierarchy levels Organization of module data Accesses to information (linking elements with data)
Central module (high amount of dependencies)	Detailed implementation analysis (central module) 1) Understanding implemented production process 2) Identification of code parts linked to other modules (e.g., variables in interlocking conditions) 3) Analysis of according data flow and involved modules, POUs and DBs	Modules and POUs with dependencies to central module (incl. hierarchy levels and functionality) Concerned module interfaces (any standardization, recurring patterns in modules with similar functionality) Amount and type of exchanged data Intention behind data exchange; reason for dependency
Extra-functional task error-handling	<i>Aspects 6, 7 and 9</i> 1) Directory for error data in a module 2) Tracing of signal chain 3) Reaction of a module to errors (internally and externally)	Internal identification and reaction to an error (interlocking, halt of operation sequence, change of operation mode) Communication of module error to superordinate levels (any hierarchical patterns) Error propagation to other modules (uniform interface) Separation of (extra-)functional parts; separation of application-(in)dependent parts

In summary, two main aims were targeted in the analysis in Step 2. Firstly, understanding the project's general structure and functionality distribution, including the link to controlled automation hardware. For this purpose, the mechanical layout plan and programming guidelines were used. Secondly, analyzing module interfaces, including dependencies of central modules and between extra-functional, generic software implementation parts with project-specific modules. The support of the software developer was essential for the module selection and comprehensibility.

The **results** from the initial analysis were **visualized and documented** mainly manually (R_{Doc}). Regarding the overall software structure, the integration of modules into the template project and design decisions targeting extra-functional aspects such as error handling, sketches were used to illustrate the functionality distribution across different hierarchy levels and the identified dependencies (cf. Figure 27). Moreover, the dependencies between different modules were documented in various tables, including classifications of the dependencies. Additionally, manual sketches of the data flow between selected modules were created.

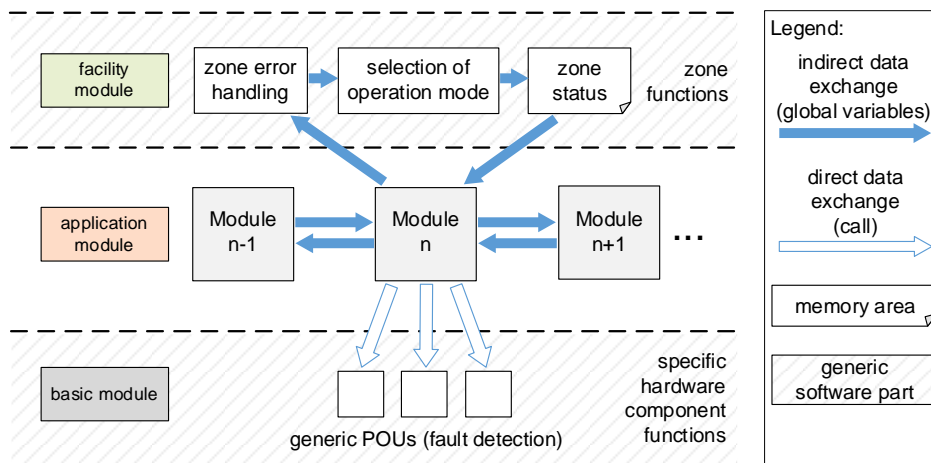


Figure 27: Module integration into the template project in the style of [Vog⁺15a], adopted from [Hub20].

Finally, a **summary of the insights from Step 2** regarding the general software architecture and the targeted analysis goal was created. As depicted in Figure 27, the software is divided into three hierarchy levels. The facility and the basic module level contain standardized, project-independent POUs for extra-functional tasks and hardware control. On the facility module level, the zone coordinates the project-specific, hardware-oriented modules located on the application module level. Data exchange, including coordination between physically neighboring modules, is mainly implemented indirectly via global variables, which is described in the programming guidelines.

Since the software developer highlighted module interfaces as critical and error-prone, these were analyzed in detail. The intended interfaces and dependencies were documented, including the semantics behind each dependence. Three basic types of indirect data exchange were identified, which are utilized for different tasks, e.g., information exchange for *process coordination* or *work-piece data transfer*. Details are described in [Fis⁺21a]. During the clarification of questions with the software developer, several weaknesses, e.g., dead code and violations of programming guidelines, were identified, including data exchange between modules implemented in POUs, which are not foreseen to realize dependencies to other modules.

Information on Procedure Step 3 with Case Study C

Four prior **versions of the commissioned project** analyzed in Step 2 were **selected for examination in Step 3** to identify changes during the project-specific adaptations to the template and analyze their impact on modularity and reusability. Starting with the customer- and order-specific template project, software versions at intervals of one month were received. Thus, it was decided to analyze the changes performed continuously from the template to the commissioned version of the project. Moreover, a second, well-implemented project of a comparable machine and, thus, with a similar structure, following the same development process, was selected to compare the implementation of module interfaces in both projects.

In correspondence with the set analysis goal and the insights gained in the previous two procedure steps, **selected aspects were analyzed in Step 3** to assess the reusability of the template. As defined with the software developer in Step 2, the analysis focused on the central modules and their dependencies on other modules since these are the main adaptations required to the template. More precisely, the five versions of project 1 were analyzed manually to identify software changes concerning interfaces of the central module that may lead to difficulties and impede reuse in the long term. Thereby, it was documented in which version violations to the template or the programming guidelines were introduced, including consequences for further project development. The interface analysis was also conducted for the second project, focusing on a similar central module. Moreover, interfaces targeting the *workpiece data transfer* were analyzed. The **comparison of project versions** was conducted as depicted in Figure 28.

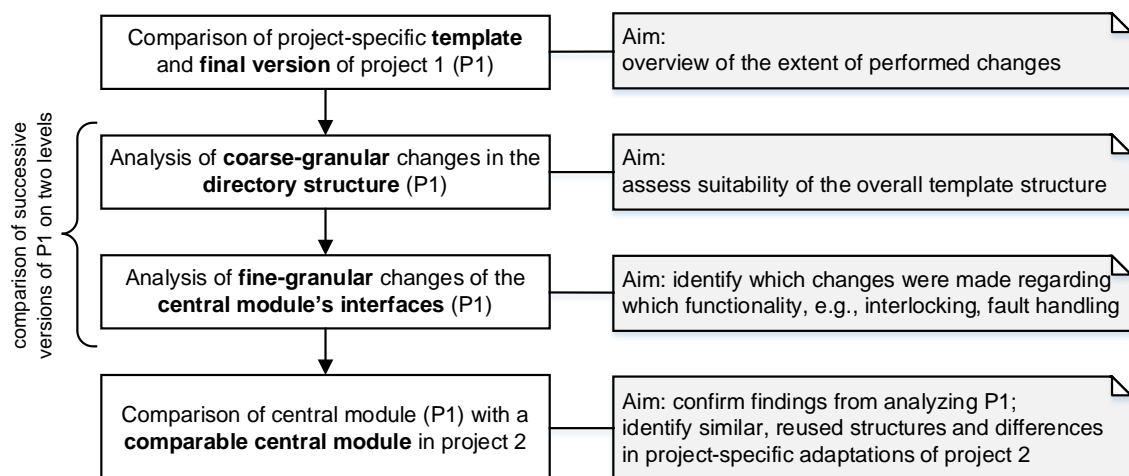


Figure 28: Comparison of project versions in Case Study C, focusing on interface changes.

The overall aim of the comparisons was to identify reasons for unusual and significant changes over different versions. The identified changes were classified as intended or as violations of programming guidelines. More precisely, it was analyzed to what extent intended interface structures in the template are used and adapted and, further, if violations are dependent on each other, e.g., if one violation results in additional ones throughout the development process. Company-tailored software metrics were developed to enable a quantitative analysis of the module interfaces (cf. [Fis⁺21a] for details). During the comparison of the project versions, a significant challenge was identifying the reason for a change, which required manual interpretation and support by the software developer (R_{Rat}). Especially when comparing the final versions of projects 1 and 2, it was challenging to distinguish between the reasons for deviations in the implementations of similar functionalities. These did not always result from violations of the programming guidelines but were also caused by different programming styles or hardware variations.

For **documentation** of the analysis results, mainly tables were used to track the changes made between the project versions (R_{Doc}). Similar to the excerpt in Table 18, these included the semantical reason for the dependency (interface task), the dependency type, its change history and if the change was intended, meaning compliant with the programming guidelines and principles. Additionally, the version in which it occurred was documented for each interface change. For a quantitative assessment of the implemented interfaces, interface metric tables from [Fis⁺21a] were used, which support the identification of disadvantageous module interfaces.

Table 18: Excerpt of the findings from the manual analysis concerning implemented interface tasks with their dependency types and unplanned changes in Case Study C.

Interface task	Dependency Type			Change history	Unintended change
	Type 1	Type 2	Type 3		
Coordination of the zone			x	A	
Error propagation to a higher level	x			A	
Transfer of workpiece data	x	x	x	B	x
Coordination between central modules in the workpiece flow	x	x	x	B	x
Movement release coordination	x	x		C	
Use of function in an external module	x	x		C	x

Legend: Type 1: read access; Type 2: write access; Type 3: copy operation; A: prepared interface structures are used; B: prepared interface structures from template are not used or changed; C: new interface structures are implemented.

Furthermore, insights gained about the order of the project-specific adaptations were documented in a textual form, supported with screenshots of code excerpts. Overall, the structure in the template and the final version are similar; project-specific adaptations are mainly realized by modifying or adding code in the project-specific modules. In contrast, the implementation of standardized POUs, e.g., for error handling or hardware control, is rarely changed. The required degree of customization of project-specific POUs varies significantly. The version analysis reveals that difficulties in the project-specific adaptation of the control software lead to unplanned changes that impair reusability. Prepared interface structures are often changed or not used at all, which leads to the implementation of new interfaces affecting the reusability of the corresponding modules (cf. Table 18, interface tasks with change history “B” and “C”). However, the template offered sufficient adaption possibilities in most cases, and many issues could have been avoided.

Comparing the two central modules in projects 1 and 2 revealed significant differences in their interfaces. While the module interfaces in project 1 show various types of violations leading to critical, disadvantageous interfaces, the central module in project 2 shows no significant issues

(cf. [Fis⁺21a] for details). However, interfaces between project-specific modules and the standardized software parts in the template are implemented identically. Overall, this result matches the experience-based assessment of the industrial experts.

Information on Procedure Step 4 with Case Study C

In the quality assessment, **deliberate design decisions** in *Case Study C* are taken into account, e.g., the intended communication between project-specific modules via dedicated POUs and global variables instead of calls (cf. *R_{DD}*). Hence, indirect data exchange is only rated as disadvantageous if it is implemented contradictory to the programming guidelines or if foreseen interfaces are not used, but new ones are implemented instead.

Overall, the **assessment** of the two final project analysis results confirms that the template enables the reuse of standardized functionalities, e.g., extra-functional tasks or hardware control, on two hierarchy levels with minimal or no modifications required. Moreover, standardized interfaces between these reusable software parts and the hardware-oriented, project-specific modules enable the integration of modules into new projects with little effort. The high modularity of the software design supports testing, adapting and reusing project-specific modules individually. However, dead code, violations of naming conventions and lacking documentation of changes indicate high time pressure during the project-specific adaptations. Furthermore, several **weaknesses** regarding the dependencies between project-specific modules were identified. The four main weaknesses are listed in the following and summarized in Table 19.

Generally, the prepared interface structures within the project-specific template support the implementation of interfaces between project-specific modules, as identified by analyzing project 2. However, in some cases, they are modified or not used at all (project 1), which reduces the comprehensibility and reusability of respective modules. The analysis findings suggest that this weakness is caused by insufficient knowledge of application developers about the template's customization options (*WC-1*), which the module software developer confirmed. Additionally, module pairs with several interfaces between each other, sometimes even several accesses from one module to the same variable of the other module, were identified. According to the software developer, these modules are unsuitable for planned reuse without changes since the high amount of dependencies increases the adaptation effort (*WC-2*). The assessment of the project-specific development process illustrates that module interfaces violating the encapsulation principle are introduced under time pressure, which hampers the module's understandability and reusability (*WC-3*). Finally, additional interfaces are caused since the software modules follow the modularization strategy applied by the mechanics' department, which does not always correspond with the functional view from a software development perspective (*WC-4*).

Table 19: Summary of identified weaknesses, derived recommendations for action and the estimated change effort in Case Study C.

Identified weakness	Recommendation for action	Estimated change effort
WC-1 Application software developers are unfamiliar with template's customization options (no use of prepared interfaces)	Enhance documentation; additional training	Low (challenges documented in detail, including code examples)
WC-2 Multiple interfaces between two modules, some accessing the same variable	Reduce interfaces between modules	Low (detailed documentation available)
WC-3 Violations of the encapsulation principle during the development	Continuous quality control of interfaces	Medium (version management available; metrics are known, but no automated means available)
WC-4 Suboptimal allocation of hardware components to modules	Function-oriented design principle in all disciplines	High (requires coordination with other disciplines, e.g., mechanics)

From these weaknesses and the insights gained during the static analysis, **recommendations for action** are derived, including an **effort estimation**. An enhancement of the template documentation and additional training for the application developers is suggested to address *WC-1*. Since detailed positive and negative examples are documented from the static analysis, the effort is estimated as low. Regarding the number of module interfaces (*WC-2*), the identified issues should be revised to implement leaner interfaces and increase reusability. Like *WC-1*, the effort is estimated as low due to the available, detailed documentation. To identify violations of the encapsulation principle early (*WC-3*), a continuous quality control regarding interfaces during the development is recommended. From the analysis, the critical interfaces are known and metrics have been defined to quantify the violations (cf. [Fis^{21a}] for details). The development workflow already includes version management. Nevertheless, the effort is estimated as medium since no automated means for metric calculations are available yet, which requires the company to implement it themselves. However, recent approaches such as [VNF22] and [Neu^{20b}], which automatically rate the maturity of library modules by tracking and quantifying the changes between different versions, show the feasibility of this recommendation. Finally, to reduce additional dependencies caused by the mechanical modularization (*WC-4*), it is suggested to apply a function-oriented design principle in all disciplines, which requires high effort as it involves other departments.

Overall, the expected benefits correspond to the estimated effort. Even though continuous quality control requires minimal adaptations to the development workflow, the software developer confirmed its usefulness as it would provide quick feedback by indicating critical interfaces at an early stage. The gained and documented knowledge about typical interface violations was rated beneficial, as it supports their recognition. The software developer and the manager confirmed two application scenarios of the conducted quality assessment: a continuous interface analysis during the

project-specific development process to identify violations at an early stage and post-processing of finished projects for standardizing project-specific modules (cf. [Fis⁺21a]).

Lessons learned from Case Study C

The quality assessment procedure application in *Case Study C* confirmed the results from *Case Study B*. Moreover, additional insights were gained, which are summarized in Table 20.

Table 20: Summary lessons learned from procedure application in Case Study C.

Procedure step and activity	Challenges	Required input	Lessons learned about the proposed quality assessment procedure
Step 1) Interview with experts	One initial expert interview was not sufficient	After joint identification of challenges and pain points, a suitable machine example was required.	Interview was split into two parts to enable software developers the selection of a suitable application example in an internal discussion.
Step 2) Static analysis of first project	Refinement of initially selected analysis goal	Indication of challenging software parts known to the software developer from his experience.	High relevance of continuous feedback from software developers during the analysis to refine the chosen analysis goal.
Step 2) Choice of first software project	Conducting the analysis efficiently with a focus on software parts most relevant for the defined analysis goal	The software developer pointed out software parts where most violations were expected for a detailed implementation analysis.	Confirmed the need and benefits of choosing a project containing known difficulties and examples of gathered pain points for an efficient first analysis (requires software developer's input).
Step 2) Understanding software architecture	High amount of interfaces in the PLC project complicated the general understanding of software architecture	Detailed, manual analysis for comprehending different types of dependencies and their intentions was required	Spending too much time on details unrelated to the defined analysis goal should be avoided to make the quality assessment as efficient as possible.
Step 2) Manual and tool-based code analysis	Formalizing company-specific guidelines for automatic conformance checking is not trivial, but tool-based identification of general dependencies was not sufficient	Manual effort and feedback from software developers required to comprehend the intention behind module dependencies, including underlying functionality and semantics of exchanged data	Manual interpretation is essential to understand the rationale behind dependencies in control software and to identify violations against programming guidelines (cf. <i>RDD</i>).
Step 3) Documentation of rationale	Reason for changes between versions of the analyzed project could not be derived automatically	Manual interpretation of changes in version history, e.g., spelling mistakes in added code and comments, indicated time pressure as a reason for violations in changed parts.	Tool-based, static code analysis is insufficient to comprehend the rationale behind software changes.
Step 3) Knowledge from other disciplines	Control software alone is not enough for understanding underlying intentions	Module definition from mechanics department was required to understand the intentions behind some interface changes.	The importance of knowledge from other disciplines to assess aPS control software was confirmed.
Step 3) Selection of additional projects	Selecting a suitable time interval for versions to be manually analyzed required a compromise	Versions selected at time intervals that enable identification of change sequence and causal relations between introduced violations while	To analyze changes in the version history of a project, a time interval has to be selected, enabling an

Procedure step and activity	Challenges	Required input	Lessons learned about the proposed quality assessment procedure
		avoiding laborious manual work (caused by too short time intervals)	efficient but still detailed comparison (choice depends on company's development workflow).
All Steps	Background of the involved software developer is highly relevant for an efficient, goal-oriented analysis	Input on software's overall design strategy and recurring difficulties required and provided by module developer from standardization department. Application software developers might not have sufficient knowledge about the template's scope.	For successful application of the assessment procedure, it is crucial to include different stakeholders, if arising questions cannot be answered by the initially involved software developer.

In summary, *Case Study C* confirmed that iterative discussions with the software developer throughout all steps are essential for the procedure's successful application, including refining the defined analysis goal. Furthermore, *Case Study C* demonstrated that the proposed procedure enables the integration of the software quality assessment continuously during development or once at specific development steps (R_{Work} , cf. [Fis+21a] for details). Albeit barely any means for an automatic static code analysis were available during the time *Case Study C* was conducted, the procedure supported the control software assessment. It could identify approaches for partially automating the performed analysis in the future. Overall, the application of the quality assessment procedure enabled a goal-oriented, systematic analysis, including the derivation of recommendations for action, which were estimated as helpful by the software developer and the involved manager. Furthermore, it showed that the procedure is successfully applicable to mature software already implementing means for planned reuse.

7.1.3. Additional Insights from Industrial Case Study D

For confidentiality reasons, no details about *Case Study D* are described. Moreover, the insights gained about the proposed procedure are very similar to the previous *Case Studies B* and *C*. *Industrial Case Study D* was conducted in a company within the automotive sector (mounting and testing of components) for a maturity assessment of the company's programming guidelines and template project (cf. overview in Table 13). The control software parts are created by several software developers and merged within one project at the end. During the application of the quality assessment procedure, the involved group manager highlighted the benefits of utilizing (annotated) call graphs to visualize dependencies and functionality distribution within the project. More precisely, he stated that the documentation enabled him to understand the applied software architecture entirely. Further, it supports the identification of crucial, central POUs, which need to be

checked in detail regarding their quality. Moreover, the visualization also enables the identification of POUs that play a central role in integrating the different software parts created by different developers into one project, which is essential for the company due to the distributed software development approach. In summary, software developers for standardized and application-specific parts and the group manager confirmed the suitability of the annotated call graphs to understand the software architecture and identify critical dependencies.

7.1.4. Insights from Case Study E with a Lab-sized Demonstrator

Case Study E targets the similarity estimation of POUs inside PLC software variants of a lab-sized, factory automation demonstrator called extended Pick and Place Unit (xPPU) [Vog⁺14]. The case study aimed to identify copied code parts as a pre-requisite for their planned reuse after refactoring. It was developed in the scope of the DFG-funded project RED SPLAT [GEP22a] with TU Braunschweig (TUB) and includes results from Alexander Schlie [Sch⁺19] and Kamil Rosiak [Ros⁺21a]. The concept for defining similarity metrics for control software and their subsequent calculation has been developed by TUB [Sch⁺19]. TUB's *Variability Analysis Toolkit* (VAT) [Ros21b] was used to automatically perform the similarity estimation within *Case Study E*. VAT supports identifying similar software parts with software metrics tailored to the respective programming style. The activities performed in *Case Study E* are linked to the four procedure steps in Figure 29. The defined analysis goal is the identification of code duplicates and variants to derive library POUs for planned reuse. The xPPU project variants are programmed in Beckhoff TwinCAT 3 and structured according to the ISA 88 hierarchy levels. In Step 2, static code analysis is performed to comprehend the software structure and programming style. Based on the insights, similarity metrics tailored to the identified programming style are defined in VAT by selecting attributes, their weights and thresholds. In Step 3, four xPPU project variants are compared in VAT. The results are stored in a family model, which classifies mandatory, alternative and optional code parts (cf. [Ros⁺21a] for details). Additional visualizations tailored to the needs of different PLC software stakeholders have been implemented and evaluated, such as a chord diagram (cf. [Fis⁺20a] for details, R_{Doc}). The results of the tool-based similarity estimation are reviewed manually and, if needed, the similarity metrics are refined. Finally, a manual interpretation of the results in Step 4 leads to deriving recommendations for action, e.g., merging mandatory or alternative POUs into parameterizable library modules. The refactoring workflow proposed in [Fis⁺20b] can be applied for performing the library module definition.

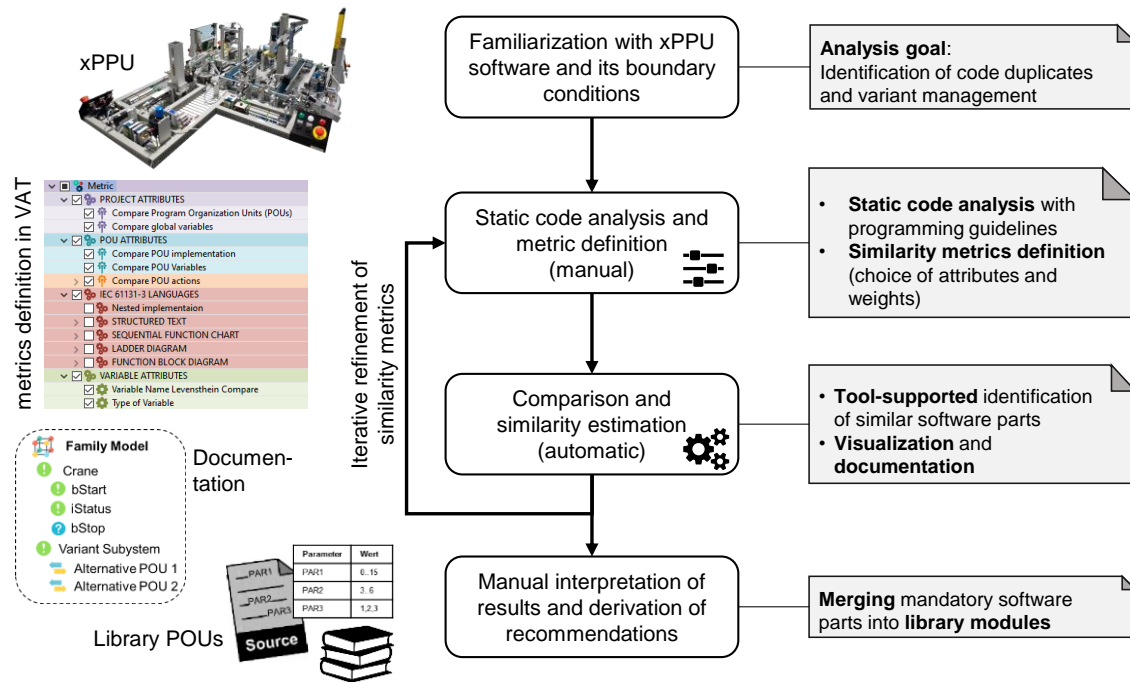


Figure 29: Procedure steps for defining customized similarity metrics and the subsequent semi-automatic identification of reusable software parts; details published in [Fis⁺20a; Fis⁺20b; Ros⁺21a].

The similarity analysis procedure presented in *Case Study E* was evaluated by ten industrial experts in an interactive, three-hour online workshop conducted during the *Automation Software Engineering Congress 2020*. The 18 workshop participants were industrial experts from different domains, including machine and plant manufacturers and PLC developers. The xPPU served as a simple example so that the experts could understand the procedure in the short time of the workshop. Since the control software is programmed according to industry standards (ISA 88 and OMAC state machines), the results are expected to be transferable to industrial-sized systems.

In the beginning, a presentation with theoretical background regarding static code analysis, software metrics and planned reuse of control software modules was given by the author. Subsequently, divided into three groups, the participants applied manual code analysis to identify variants in legacy control software and estimate their similarities. For this task, software variants of the xPPU's conveyor belts were provided. After discussing the experiences and challenges during the manual static code analysis for similarity estimation, a live demo of the semi-automatic similarity assessment workflow in *Case Study E* (cf. Figure 29) was given: the definition of similarity metrics and their calculation to derive a family model in VAT [Sch⁺19], exploration of the derived family model with visualizations tailored to the needs of different stakeholders [Fis⁺20a] (R_{Doc}) and subsequent refactoring of a selected, mandatory POU and its identified variants [Fis⁺20b]. Subsequently, ten participants voluntarily filled out a questionnaire consisting of ten questions.

In the questionnaire, 80% of the participants confirmed that variant management and planned reuse are (very) significant challenges they face in their companies, representing a pain point (cf. R_{PP}). Furthermore, 80% indicated that the semi-automatic workflow could be entirely or partially integrated into their development workflow, confirming R_{Work} . As the main challenges for applying the presented workflow in an industrial context, the participants indicated in a free-text question the time to familiarize themselves with the tool (VAT), the acceptance of the software developers and the use across different PLC platforms. Moreover, 40% of the participants rated the workflow as helpful and 50% as partially helpful to support the planned reuse of variant-rich software and, thus, address their challenges. All participants agreed that the semi-automatic workflow has benefits compared to a purely manual analysis because it is faster, easier to perform and the metrics ensure a consistent similarity assessment.

In *Case Study E*, manual static code analysis was required in Step 2 to customize the VAT similarity metrics to the programming guidelines and practices applied in the targeted control software since no universal metric applicable to all programming styles can be defined. Instead, the design decisions of the software developer need to be taken into account (cf. R_{DD}). Once the similarity metrics have been defined, an automated comparison and similarity estimation of software variants can be performed. This shows that the presented quality assessment procedure can be flexibly extended with semi-automatic methods as required, i.e., manual static code analysis means should be combined with automatable, tool-supported means such as VAT wherever possible. Despite the high potential of automatically deriving variation points in control software, the similarity metrics do not replace manual, human interpretation of the gained results. For example, expert knowledge is required to distinguish recurring variants that should be merged into a reusable library module from highly customized solutions (cf. R_{Rat}).

7.1.5. Summary of Insights Gained Through Case Studies

This Section links the insights gained from the five conducted case studies (industrial *Case Studies A to D*, lab-sized *Case Study E*) to the related requirements as a basis for the concept assessment in Chapter 8. Overall, control software of different maturity was analyzed during the case study evaluations. More precisely, the quality assessment procedure was successfully applied to historically grown control software, which was developed using *copy, paste and modify* and with a high amount of indirect data exchange (*Case Study B*). The procedure was also successfully applied to highly mature control software following a function-oriented modularization principle and reuse strategies such as library modules and templates (*Case Studies C and D*).

The five successful procedure applications showed that the proposed analysis steps are not always precisely separable from each other. Instead, in some cases, it was required and possible to perform

a mixture of the procedure steps, e.g., analysis in Steps 2 and 3, or perform iterations concerning sub-aspects. This finding aligns with the proposed analysis concept: for a successful quality assessment, the procedure needs not to be strictly followed. Instead, it shall provide a framework to assist software developers in conducting the analysis in a systematic, goal-oriented manner.

The comparison of the case studies confirms that the developed quality assessment procedure is not limited to specific standards. Instead, it allows the integration of application sector- or company-specific guidelines. Moreover, the procedure provides a structure to gather relevant information and perform a manual software analysis with the option of incorporating available methods and tools for automatic code analysis. Thus, the analysis methods are flexibly selectable and the assessment procedure is adaptable to the situation. Different pain points, analysis goals, and design decisions have been targeted in the case studies. The proposed procedure provides a framework for the analysis that does not require or prohibit any process or application sector properties. However, it has not been evaluated with continuous processes in the process engineering sector yet.

From the case studies, two challenges are identified: the provided interview guiding questions and list of potential analysis goals can support the industrial experts to apply the procedure independently (R_{Use}). However, basic background knowledge about control software design and reuse is required for a successful application. The procedure does not contain automated means to suggest improvements for the analyzed control software. Consequently, to ease applicability, the procedure should be combined with refactoring guidelines or best practices, which are, however, not yet available. Second, if the analyzed software is unstructured and monolithic, without comments, it might not be suitable to be used as a basis for improving individual aspects. Instead, a greenfield development of the respective software might be better. However, even if software of low maturity is analyzed, the gained documentation can help avoid repeating the disadvantageous design decisions in the new concept, summarize critical design decisions required due to boundary conditions or provide an overview of implemented functionalities or interfaces.

A summary of the insights gained from the case studies concerning the derived requirements is presented in the following Table 21.

Table 21: Summarized examples from the case study evaluations targeting different requirements.

Requirement	Examples from conducted industrial and lab-scale case studies with remarks	Limitations
<i>R_{PLC}</i> – Platform Independence	<ul style="list-style-type: none"> Case Studies A, B: Siemens SIMATIC Manager (STEP 7) Case Studies C, D: Siemens TIA Portal Case Study E: Beckhoff TwinCAT 3 	Case Studies limited to the field of factory automation; Focus on classical control software (OO-IEC not targeted)
<i>R_{Pro}</i> – aPS as Product	<ul style="list-style-type: none"> Case Studies A, B, D: Plant manufacturers Case Studies C: (special purpose) machine manufacturer Case Study E: Lab-sized demonstrator 	Procedure was not applied to a serial machine manufacturer
<i>R_{Use}</i> – User	Case Study B: Software developers applied the procedure themselves to a different plant part after participating in a two-part training workshop.	To avoid “operational blindness”, a software developer should not assess his own software. Benefits of an unbiased “view from the outside” were confirmed by industrial experts from Case Studies A, B and C.
<i>R_{Sec}</i> – Application Sector	<ul style="list-style-type: none"> Case Study A: Woodworking Case Study B, E: Intralogistics Case Study C, D: Supplier in automotive engineering 	No consideration of application sectors with special legal regulations, e.g., food and beverage, or pharma and medicine
<i>R_{PP}</i> – Pain Points	<ul style="list-style-type: none"> Case Study A, B, C and D: Challenges in the software development were identified in expert interviews, which enabled the definition of the analysis goal. Case Study E: Industrial experts confirmed variant management and planned reuse as their challenges. 	
<i>R_{Work}</i> – Workflow Integration	<ul style="list-style-type: none"> Required workflow integration depends on the defined analysis goal. Case Study A, B, D: After commissioning (“as build” after development and after start-up of aPS) Case Study C: Continuously during software development; one-time analysis after start-up Case Study E: Industrial experts from different companies confirmed that integration into their industrial workflow would be possible 	Manual workload as hindering factor for regular application of quality assessment procedure; lack of tools for continuous support
<i>R_{DD}</i> – Design Decision	<ul style="list-style-type: none"> Case Study A: Data exchange with WMS, storage car control and other PLC must not be modified Case Study B: Strict timing requirements must be kept (module communication); information exchange with the superordinate system; standardized FCs should not be changed; a combination of hardware- and function-oriented modularization is kept in the new concept Case Study C: Intended global data exchange between modules (according to specified rules, to ensure that modules are reusable in different contexts); no direct calls between modules Case Study D: Hierarchical modularization with dedicated FBs for controlling plant parts and extra-functional tasks; DBs as interfaces within PLC software project and to HMI Case Study E: OMAC State machine and ISA 88 were considered 	
<i>R_{Goal}</i> – Analysis Goal	<ul style="list-style-type: none"> Considered goals ranging from variant management including influence from automation hardware (Case Studies A, B) over a reduction of code duplicates (Case Study E) up to quality assessment of project template (Case Study C) and programming guidelines (Case Study D); (cf. details in Table 13) Analysis goals derived with interview guiding questions and pain points (all case studies) Successful refinement of analysis goal during procedure application (Case Study C) 	Case Study C: Pain point was clear in first interview, but second interview and emails were required to define the analysis goal Experience: When reviewing the first analysis results, causes for subjective pain points emerge; subsequent refinement of analysis goal is possible and reasonable

Requirement	Examples from conducted industrial and lab-scale case studies with remarks	Limitations
<i>R_{Scal}</i> – Scalability	<p><i>Case Studies A, B, C</i> and <i>D</i> targeted industrial-sized control software projects in the range of, e.g., [Vog⁺17] (cf. overview in Table 13, p. 100), with POUs written in SCL with up to 1500 Source Lines of Code</p>	Effort reduction does not apply to an analysis of software of two utterly different machine types, possibly even at different locations of the company and with other guidelines
<i>R_{Eff}</i> – Application Effort	<ul style="list-style-type: none"> • <i>Case Study B</i>: After procedure application, hardware variants and exchanged variables are known and documented → updating documentation with new variants is less effort than initial analysis • <i>Case Study C</i>: After first application, template and software structure are known and critical parts are identified (violations quantifiable with metrics); similar analysis of a project following the same reuse strategy is expected to be less effort; continuous application of analysis is expected less effort than initial analysis (critical interfaces are known, additional ones can be added) • <i>Case Study D</i>: After procedure application, functionality distribution and interfaces between library modules and application-specific parts are documented, which reduces effort for Step 1 in further applications • <i>Case Studies A and B</i>: Linkage between hardware modules and software required to derive the reason for the variability in the control software (needed for functionality-oriented modularization in <i>Case Study B</i>) • <i>Case Study B</i>: Comments, variables and POUs have different symbolic names across projects (partially caused by programming style/personal preferences) → requires manual interpretation of the comments to identify “false” variations and corresponding parts in the control software • <i>Case Study C</i>: Reason for changes identified in metadata (missing comments, miss-spelled words) or by considering information from other disciplines (modularization strategy from mechanics department causing challenges in control software modularization) • <i>Case Study C</i>: Interpretation of differences between POU interfaces in two independent projects (considering modules implementing similar functionalities) requires the support of an industrial expert • <i>Case Study C</i>: Selection of central modules for a detailed analysis saved laborious manual work and was required to gain insights efficiently (not possible without the software developer’s input) • <i>Case Study D</i>: Examples are available, but no details are provided due to confidentiality reasons • <i>Case Study E</i>: Expert knowledge is required to distinguish recurring variants suitable for reuse as library modules from highly customized solutions 	<p>From the analysis, it is usually known, if a planned change affects the entire software structure (high effort) or specific parts (lower effort). Identifying the “worst/top ten” POUs regarding a selected weakness allows starting with the software parts that have the greatest expected benefits and support a continuous improvement process.</p>
<i>R_{Weak}</i> – Weaknesses and Change Effort	<p>Analysis results enabled identification of weaknesses and estimation of required change effort, for example:</p> <ul style="list-style-type: none"> • <i>Case Study A</i>: Software can be divided into variant-dependent and -independent POUs (precise number known from Step 3); effort for a planned reuse of invariant parts is low but expected to improve the control software quality; for variable parts, the link to the hardware is clear and the documentation supports the software developer in his modification tasks. • <i>Case Study B</i>: Change effort is high since the entire software structure needs to be changed. Especially logic and hardware control need to be separated and POU interfaces need to be redefined (flag variables should be eliminated). The expected benefits for reuse are also high and detailed documentation is available. To avoid variations resulting from different programming styles, identified examples can be directly integrated into the programming guidelines with a low effort. 	<p>Identification of weaknesses requires suitable documentation of the analysis results (for different stakeholders, which also provides the base for change effort estimation). From the analysis, it is usually known, if a planned change affects the entire software structure (high effort) or specific parts (lower effort). Identifying the “worst/top ten” POUs regarding a selected weakness allows starting with the software parts that have the greatest expected benefits and support a continuous improvement process.</p>

Requirement	Examples from conducted industrial and lab-scale case studies with remarks	Limitations
<ul style="list-style-type: none"> • <i>Case Study C</i>: Insights from the analysis include interface types and their rationale (good and bad examples available), dependencies changed and violated during project development (including how they are violated) and modules with a high number of interfaces (optimization potential). Medium effort is expected for revising specific critical interfaces (examples available, training and enhancement of guidelines with identified weaknesses). High effort but also high benefit is expected from continuous change tracking to identify critical interfaces at an early stage (metrics available). Interdisciplinary exchange to avoid sub-optimal code structures requires high effort. • <i>Case Study D</i>: Examples are available, but no details are provided due to confidentiality reasons. • <i>Case Study E</i>: Identified mandatory software parts can be merged into library modules with low effort. Merging alternative parts requires medium to high effort (defining parameterizable library modules or universal modules). 		
<p><i>R_{Doc}</i> – Documentation</p> <ul style="list-style-type: none"> • <i>All use cases</i>: different levels of granularity and viewpoints were used for documentation (Steps 2 and 3); gained documentation was beneficial to rate change effort (cf. <i>R_{Weak}</i>). • <i>Case Study A</i>: Software architecture and main functionalities, including error handling (lack of documentation about software design decisions was a mentioned challenge), are documented with annotated call graphs in Step 2; table-based summary of links between hardware variation points and affected POUs. • <i>Case Study B</i>: Examples of programming guidelines' weaknesses documented (Step 3); influence of functionality and mechanical layout on the respective control software documented in feature models (coarse-grained) and table-based, with annotated screenshots and examples (fine-grained); documentation of functionality distribution; table of required module interfaces (derived from different variants). • <i>Case Study C</i>: Graphical, textual and table-based documentation highlights critical parts in the template, including correct implementations (project 2) and violations (project 1). • <i>Case Study D</i>: Confirmed usefulness of documentation for understanding overall software structure, functionality distribution and dependencies (helpful for estimating the risk of changes). • <i>Case Study E</i>: Software variants are documented and visualized for different stakeholders, including their location, size, and quantified similarity value. 		<p>Examples of documentation styles on different granularity levels are provided. However, no complete list of available means for documentation is given. Further, no set of rules for selecting suitable documentation is defined. Documentation types are not linked to potential analysis goals.</p> <p>The procedure does not provide support for the correct interpretation of the different means of documentation. It is only a framework with examples and the interpretation is individually/use case-specific.</p>

7.2. Evaluation with Industrial Experts in Industry Working Group

The evaluation with 20 German industrial experts from the aPS domain was conducted in the scope of an online meeting of the industry working group (WG) “Modular machine and software (Modulare Maschine und Software)” on June 18th, 2021. Some of the industrial experts have participated in the WG for several years and join the WG meetings regularly to exchange views, thoughts and ideas on the planned reuse of control software. Thus, the WG participants acknowledge the importance of modular software architectures and the need for strategies for planned software reuse. They are aware of the current challenges and obstacles hindering both. Furthermore, the participants have different backgrounds ranging from software developers to managers. Thus, the WG is very suitable for evaluating the proposed quality assessment procedure.

The evaluation was carried out within the WG meeting’s 2.5-hour morning session: after a short welcome of the 25 participants, the WG meeting started with a 40-minute keynote presentation entitled *Quality assurance using static code analysis and software metrics* presented by the author. First, a general overview of static code analysis methods was provided. Subsequently, the quality assessment procedure was derived utilizing the *Self-X Material Flow Demonstrator* from the AIS institute (originally an industrial testbed) as a use case. Following the keynote, the participants could ask questions regarding the presentation. The only question raised was whether the presented methods had already been applied to industrial control software, which was confirmed.

Subsequently, the participants were asked to individually fill in an online questionnaire regarding the presented quality assessment procedure before exchanging their thoughts in a virtual break. Participation in the questionnaire was voluntary. After a short break, current challenges and solution approaches for using static code analysis were discussed in four sub-groups, each moderated by a member of the AIS institute. Afterward, each moderator presented the group’s results to the other participants and the morning session was closed with a summary.

7.2.1. Evaluation of the Industrial Applicability of the Quality Assessment Procedure with Online Questionnaire

The participants’ background and the main findings from the online questionnaire, which was voluntarily and anonymously answered by 20 of the 25 participants, are presented below. The original German version of the questionnaire and all answers can be found in Appendix B. The asked questions are referred to as *WG#[question number]*, e.g., question 1 is identified as *WG#1*.

Since participation in the questionnaire was anonymous, the distribution of the participants’ backgrounds is not known in detail. In the WG meeting, 25 industrial experts from 16 German machine and plant manufacturing companies from seven application sectors (cf. Figure 30 for an overview

gained through question *WG#11*) with varying boundary conditions participated. In general, aPS from plant manufacturing are more customer-specific regarding size and location. However, the assignment between machine and plant manufacturing is not trivial [Vog⁺21a]; the difference between special-purpose machine and plant manufacturers is not strictly defined. Of the 16 German companies, eleven are considered machine manufacturers, ranging from serial to special purpose machines, and five are considered plant manufacturers by the author. The participants had different positions within their company, e.g., managing directors, department and team leaders, and control software developers (module and application developers) were present. Moreover, 75% of the participants who responded to the questionnaire indicated that at the time of the WG meeting, no static code analysis was used in their development workflows (cf. *WG#9*).

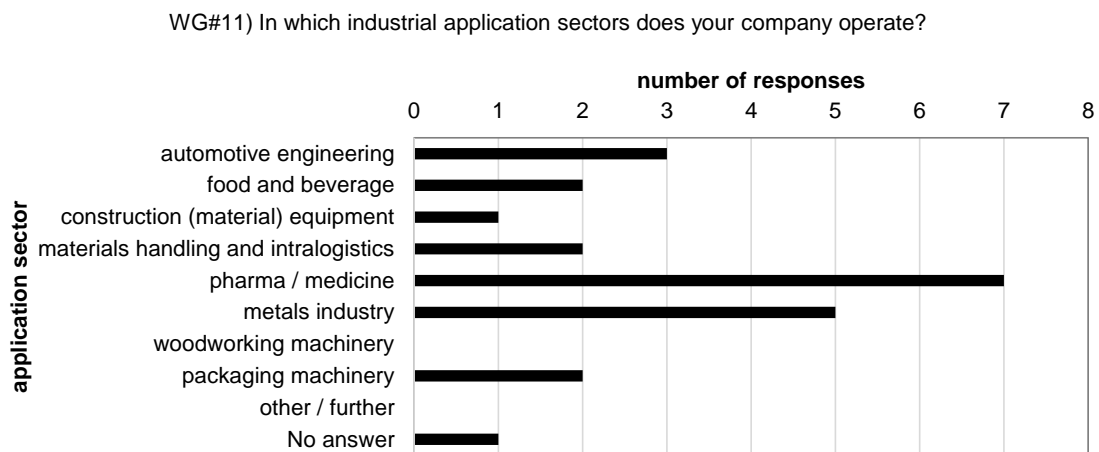


Figure 30: Background of participants regarding application sector (question *WG#11*, multiple-choice, answers: 20 participants).

The *industrial Case Studies* presented in Section 7.1 were conducted with control software implemented for Siemens PLCs. Therefore, to assess the requirement R_{PLC} , industrial experts using different platforms were questioned in the WG meeting. The high market share of Siemens also shows in the WG participants. However, of the ten experts indicating the use of Siemens as a platform, all but two use at least one other platform in their company. Figure 31 gives an overview of the used platforms, including IEC 61131-3 based platforms such as Beckhoff, B&R and Schneider Electric. Overall, the background of the participants is mixed, including at least seven different platforms, which is considered representative.

The participants formed a heterogeneous group regarding their position in the company, application sector and used programming platforms. Thus, their answers are considered representative of the aPS domain. To assess requirement R_{Use} , question *WG#8* targeted the application of the proposed procedure by the industrial expert him-/herself. Overall, 65% of the participants confirmed

that they would at least be able to partially apply the procedure independently after training. Another 30% indicated they are unsure since this strongly depends on the training amount. Considering that the procedure should be used by software developers and not necessarily by group leaders or software managers, who also participated in the questionnaire, overall, this confirms that the industrial experts would be able to apply the procedure after an adequate amount of training (R_{Use}). Moreover, in question *WG#12*, 45% of the participants indicated that the assessment procedure would be applicable in their application sector, and 55% agreed that it would be partially applicable. In combination with *WG#13* (cf. Figure 30, application sectors the companies operate in), the requirement R_{Sec} is considered fulfilled.

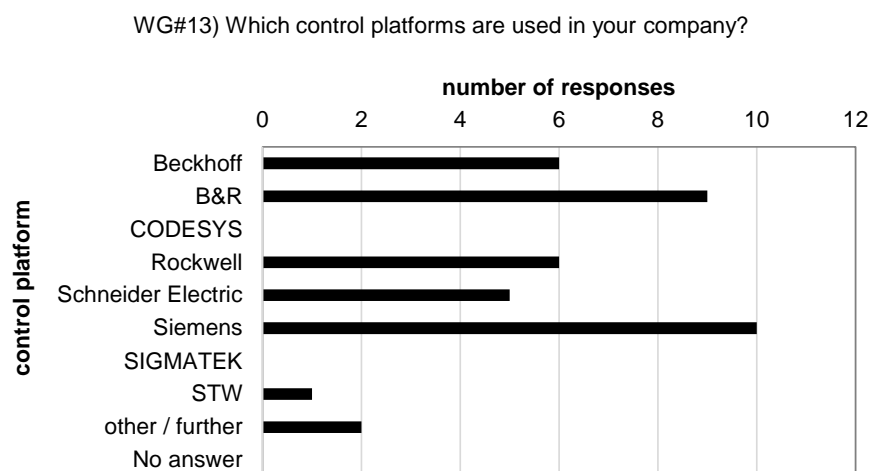


Figure 31: Background of participants regarding used programming platforms within their company (question *WG#13*, multiple-choice, answers: 20 participants).

Regarding the interview guiding questions presented during the keynote, question *WG#6* targeted their suitability for identifying the analysis goal and additional information, e.g., from other disciplines, for conducting the analysis. In the answers, 40% of the participants confirmed that the interview guiding questions are suitable and 55% indicated they are partially suitable to identify the analysis goal. However, identifying pain points was not targeted in question *WG#6*. Consequently, the requirement R_{PP} is rated as partially fulfilled.

The questions *WG#3* addressed the integratability of the presented quality assessment procedure into the company workflow. While 20% of the respondents agreed that the entire procedure could be integrated into their development workflow, 55% of the participants indicated that it could be partially integrated (cf. Figure 32). Since the proposed procedure does not necessarily need to be integrated completely but only the parts that are helpful and relevant for the company's defined analysis goal, the requirement R_{Work} is considered fulfilled.

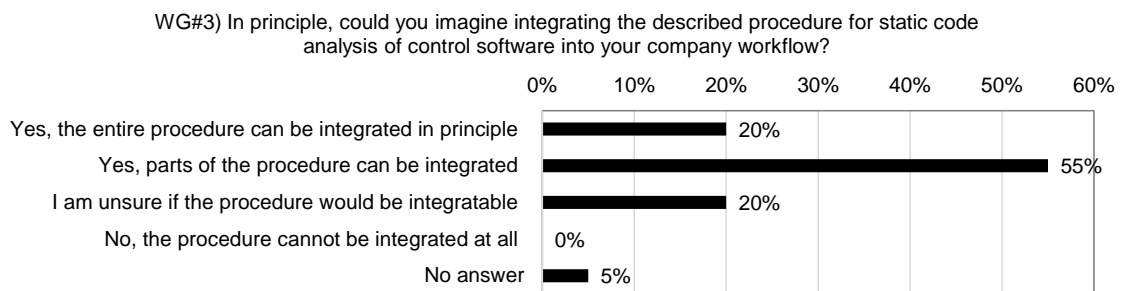


Figure 32: Estimation of the integratability of the quality assessment procedure into the company workflow (question WG#3, single choice, answers: 20 participants).

Concerning the procedure's applicability to company-specific boundary conditions such as unchangeable design decisions, 20% of the participants agreed that it is definitely applicable and 80% indicated that it is partially applicable (cf. questions WG#7). Accordingly, requirement R_{DD} is considered partially fulfilled. The participants of the WG have a mixed background, including serial machine, special purpose machine and plant manufacturing. Further, the majority confirm that the procedure can be integrated at least partially into their workflow (cf. question WG#3) and that it is capable of at least partially taking the company's design decisions into account (cf. question WG#7). Thus, the requirement R_{Pro} is considered fulfilled.

Finally, the summary of the free-text answers to question WG#4 targeting challenges in applying the procedure in an industrial environment is depicted in Figure 33. The summary shows that the challenges most frequently mentioned independently by several participants are boundary conditions such as the tool integration, the developers' mindset or the high implementation effort under time pressure. However, only a few stated challenges are caused by the proposed procedure itself, e.g., one participant mentioned the insufficiency of the interview guiding questions, which were shortly presented in excerpts, as a challenge hindering the procedure's application.

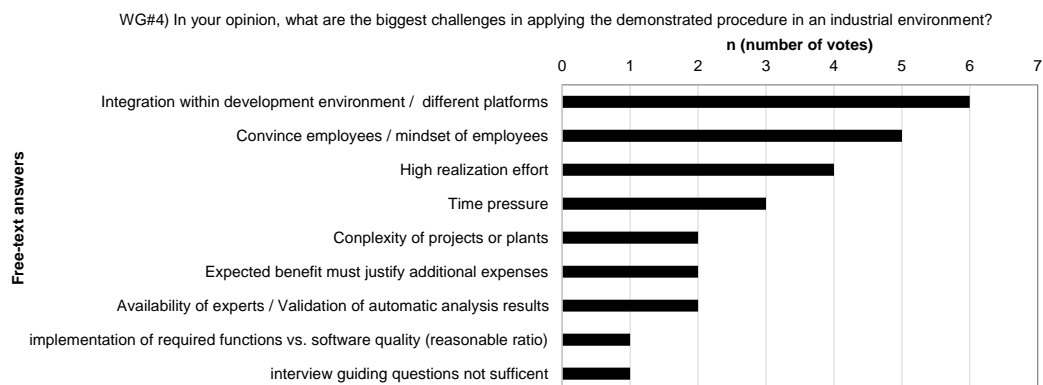


Figure 33: Challenges hindering the procedure application in an industrial context (question WG#4, free-text question, answers: 19 participants).

Finally, in question *WG#5*, all participants agreed that the presented procedure is at least partially helpful to integrate means and methods for quality assurance with static code analysis into their development process.

7.2.2. Group Discussions on Challenges of Application in an Industrial Context

For the second part, participants were divided into four groups to discuss current challenges and boundary conditions with one group, consisting of seven participants, focusing specifically on the presented procedure. Mentioned challenges from all groups are summarized in the following.

Several challenges concern the applicability of analysis tools; e.g., one participant highlighted the importance of a meaningful toolchain for conducting static code analysis in an industrial context. More precisely, an analysis tool must be easy to use, flexible towards boundary conditions and embedded in the development workflow since “an independent tool would not be used due to time constraints, no matter how good it is”. Although tool-based static code analysis is only a part of the developed quality assessment procedure, this statement confirms the importance of integrating the quality assessment at an appropriate point in the company’s development workflow (cf. *R_{Work}*). Furthermore, it highlights the need to consider company-specific boundary conditions (cf. *R_{DD}*).

In several groups, application and module developers emphasized the assessment of the refactoring effort, including its expected benefit, as essential to convince the management level of the necessity to perform time-consuming software maintenance tasks. The benefit, which opposes the effort of the analysis, must be communicateable, e.g., by statements like “the software development of the next plant is 20% cheaper”. However, currently, no approaches to assess or quantify the long-term effects of “good and bad software” are available. By documenting the analysis results for subsequent estimation of the required change effort and expected benefits (cf. *R_{Weak}*), the proposed procedure supports assessing the refactoring effort after the analysis has been conducted. However, measuring long-term effects requires analyzing aspects such as technical debt as targeted in [VB21], which are not included in the procedure.

Moreover, various external factors were mentioned in the discussions, such as the need for a holistic modularization principle including hardware and software, required coordination between different disciplines and the currently lacking cross-disciplinary understanding of (software) quality. An industrial expert from a plant manufacturing company was concerned regarding incomplete development models and data received from other involved departments/disciplines as “a far greater challenge”, which needs to be solved before optimizing the control software. With a focus on static code analysis, the acceptance by the software developers was mentioned as a prerequisite. Furthermore, the software developers stated time pressure from management to have high-quality,

“finished” software as a challenge since the developers consider software improvement as a continuous process that is never finished. The need for best practices and support when starting with static code analysis was also indicated as a challenge. This challenge confirmed the general need for a systematic, goal-oriented quality assessment such as proposed by the presented procedure, including interview guiding questions and checklists. However, best practices for developing control software are not targeted by the presented procedure and require further research.

Analyzing the software structure through call graphs was highlighted as most important in discussing analysis methods, especially when assessing brown-field projects. Another mentioned strength of the structural analysis is “finding an entry point into the software”, e.g., for its optimization or extension. Further, means for identifying code duplicates and software metrics to identify large, extensive software parts for a subsequent refactoring step were rated as beneficial. This confirms the order and importance of aspects in the analysis checklist (cf. Table 8, p. 65 and Appendix A.2, p. 191).

During the discussion, the industrial experts stated various application scenarios and goals where they expect benefits from the proposed procedure. While one participant questioned the usefulness of the procedure when applying it to well-defined control software already programmed according to company-specific guidelines, another emphasized that the presence of guidelines “does not mean that everything runs perfectly”. Instead, in his eyes, an analysis of the quality of the programming guidelines is necessary. One expert sees potential for the procedure’s application in the current changeover from IL to ST in his company, e.g., to analyze the dependencies between the modules. Another expert mentioned conducting an analysis focusing on improving the software quality, as new developers often have little experience with implementation, while management demands ever-greater efficiency. To integrate new employees more quickly, the usability and interfaces of the modules are of increasing relevance so that they can be used without knowledge of their internal details. The modules could be improved by applying the procedure. Finally, another mentioned application scenario was to check for software changes during commissioning. However, according to the industrial experts, this analysis would have to include the accompanying circumstances, i.e., the reason for the change, which confirms the importance of manual interpretation of changes (cf. R_{Rat}). A developer of a plant manufacturer mentioned in this context that no two plants are the same in his company. In addition, adjustments to the customer premises are always necessary. Therefore, the meaningful application of code analysis and quality assessment is limited to the time during software development. Overall, these different application scenarios confirm that the proposed quality assessment procedure can support different analysis goals at different points of the development workflow from the industrial experts’ point of view.

Despite the mentioned challenges, the participants confirmed that the presented procedure is generally helpful and applicable for integrating analysis means and metrics into the development workflow for the quality assessment with different boundary conditions and in various scenarios.

7.3. Expert Workshop with an Industrial Focus Group in the Food and Beverage Sector

The quality assessment procedure and selected aspects of its four steps were discussed and evaluated in the scope of an online workshop with experts from a German, internationally operating plant manufacturing company in the food & beverage and intralogistics sector. Overall, the company has over 16,000 employees worldwide. It mainly uses automation hardware, including programming platforms, from Siemens, Rockwell Automation, and B&R Industrial Automation. PLC projects are programmed in graphical and textual programming languages while considering company-wide programming guidelines and utilizing library POU's, code generation and templates. The invited experts were primarily from the software development department, but some also had an electrical engineering background. Most of the invitees were PLC software developers, with a few participants from the management level. These different backgrounds of the participants enable an evaluation of the proposed procedure from various stakeholders with different tasks. The results of the expert workshop were originally published in [Fis⁺22a].

The workshop was scheduled as a web meeting for 2.5 hours. After a short welcome, the quality assessment procedure was introduced in a timeslot of 1.5 hours to the participants as a mixture of presentations and live demonstrations utilizing the *advacode* prototype [Ins22]. The procedure explanation was divided into four blocks (each with 15 minutes of presentation by the author and 5 minutes for participants to voluntarily answer one or two single-choice question(s)). The blocks targeted a general introduction to the procedure, details on the interview guiding questions, conducting the analysis under consideration of company-specific boundary conditions and, finally, utilizing the result documentation to derive recommendations for actions. Afterward, the workshop participants were divided into three groups to clarify potential questions about the presented information and, subsequently, to join an interactive part as a basis for an in-depth discussion about the applicability and usefulness of the proposed procedure.

Due to the online format of the workshop, invited participants were able to join and leave the online meeting while the workshop was being conducted. In total, 46 participants joined the workshop meeting, whereby some of these stayed for (parts) of the presentation and prototype demonstration only. Of all participants, 36 remained in the web conference for the entire workshop duration. During the workshop, nine single-choice questions were asked via a polling application of

the used web meeting tool Microsoft Teams. Answering the questions was voluntary and anonymous. The findings obtained during the workshop are presented in the following sections. Certainly, the results cover only the responses of experts from a single company. However, due to the mixed background and the number of workshop participants, qualitative results can be derived, which provide additional insights beyond the previous evaluations regarding its applicability in an application sector with specific rules, regulations and boundary conditions.

7.3.1. Evaluation of the Applicability of the Quality Assessment Procedure

As described above, the introduction of the quality assessment procedure and selected subparts was divided into four topic blocks, each scheduled for 20 min. First, an overview of the means for static code analysis and an introduction to the proposed assessment procedure was given. The introduction to static code analysis was essential since 65% of the participants indicated in question *W#1* that they do not apply any means of static code analysis at all during the software development (cf. Appendix C.2 for complete answers to the workshop questions).

Secondly, details regarding the preparation step were provided, focusing on choosing an analysis goal as a prerequisite for conducting the code analysis goal-oriented, thus, enabling the subsequent derivation of recommendations for action from the gained results. An excerpt of the developed interview guiding questions was presented in theory and utilizing the application example *Self-X Material Flow Demonstrator*. Furthermore, criteria for selecting a suitable PLC software project for the analysis in the second procedure step were introduced. In the following question *W#2*, 92% of the participants confirmed that the interview guiding questions are helpful or partially helpful for identifying the analysis goal. Thus, R_{PP} can be considered fulfilled in this case.

The third topic concerned conducting the static analysis under consideration of company-specific boundary conditions. For example, the analysis and rating of data exchange types according to company-specific programming guidelines was introduced in theory and demonstrated with the *advacode* prototype. Subsequently, the participants were asked if they thought the procedure could be successfully applied regarding the boundary conditions such as design decisions in their company (Question *W#3*). While 62.5% of the participants think that the procedure is at least partially applicable, over one-third were unsure (cf. Figure 34). Since the participants included management personnel who do not work closely with control software in their daily tasks, R_{DD} is nevertheless considered partially fulfilled.

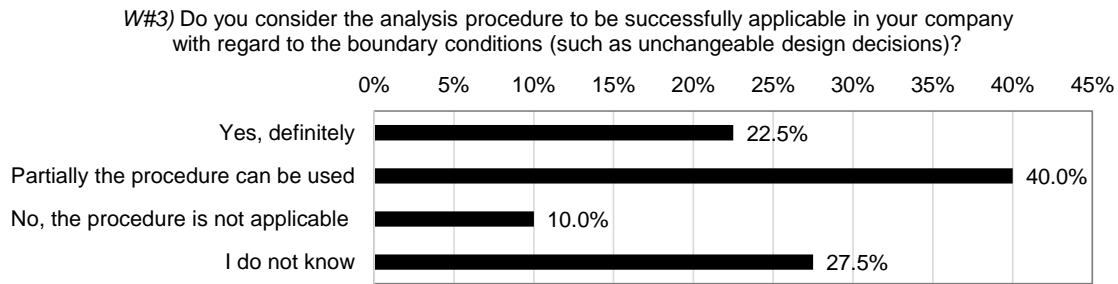


Figure 34: Answers to W#3 concerning the applicability of the analysis procedure under consideration of company-specific boundary conditions (total answers $n = 40$).

The participants were further asked in question W#4 if they think that the procedure can sufficiently address the constraints of their application sector (food & beverage) and would therefore be applicable. While 47% think it would be applicable at least partially, with 45%, almost half of the participants are unsure. It might result from the fact that not all participants had a background in PLC programming or that the short concept presentation did not sufficiently show the possibilities for adaptations. Concluding, R_{Sec} is only partially fulfilled and additional research is required to identify the critical boundary conditions and to what extent the procedure could address them.

In the fourth topic block, different documentation types for the analysis results on varying granularity levels were exemplarily provided. These are, for example, the call graph on a coarse-grained level to visualize the overall software structure and peculiarities within, e.g., violations to the intended hierarchy levels or structural patterns. On a fine-grained level, the documentation with software metrics provides quantitative values for an objective comparison of a selected software characteristic to identify the “worst ten” POUs, e.g., the ten most complex POUs. It was further highlighted that especially anomalies in the context of the own software are relevant due to a lack of commonly accepted programming guidelines. With the *advacode* prototype, exemplary available means to document different aspects such as data exchange, metric values and call graphs were demonstrated. Subsequently, 39% of the participants answering question W#5 confirmed that the documentation on different levels in the context of their software is helpful to identify anomalies and disadvantageous software elements and 46% stated it is partially helpful. Thus, R_{Doc} is fulfilled. Regarding the derivation of recommendations for action, including a rough effort estimation (Question W#6), only 20% think that the documentation enables this, while 60% state it is partially helpful. The hesitation might result from the lacking experience of the company experts with static code analysis and the required effort for manually creating detailed documentation in case tool support is not yet available. However, future research is required to identify the reasons for the hesitation and the requirement R_{Weak} is considered partially fulfilled.

As a summary for the first workshop part, the overall assessment procedure and its four steps were introduced for a second time to link the previously presented and demonstrated aspects to the different steps and highlight their relations. Next, the participants were asked if they would be able to apply the presented procedure to their control software after a first application together with an analysis expert (Question W#7). As depicted in Figure 35, about half of the participants confirm that they would be able to (partially) follow the procedure to apply static code analysis themselves (49%) and 35% think that it depends on the amount and scope of training. Therefore, R_{Use} is considered partially fulfilled. Moreover, measures to tailor the proposed training, including workshops, to the company's boundary conditions, e.g., the background knowledge of the developers, need to be investigated in a future step to ensure that the procedure application can be performed without external experts. Additionally, developing means for automated support during the procedure application, e.g., an assistant system, is expected to reduce required training.

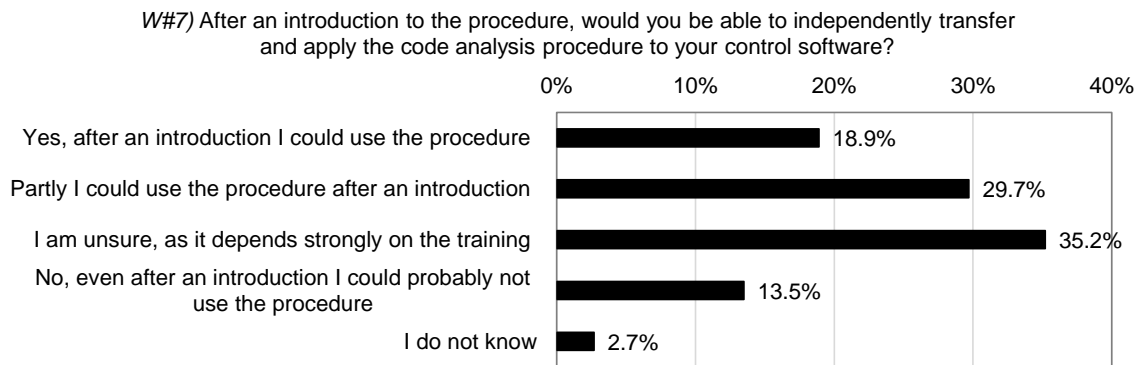


Figure 35: Answers to W#7 regarding the independent application of the analysis procedure by the participants themselves (total answers $n = 37$); pseudo accuracy to avoid rounding error.

Furthermore, the participants were asked if the procedure could be integrated into the development workflow they currently follow (Question W#8). 68% of the participants indicated that the procedure could be integrated at least partially. Since the procedure is supposed to support the application of static code analysis for a goal-oriented, context-sensitive quality assessment, it need not necessarily be included completely. Thus, the requirement R_{Work} is fulfilled.

7.3.2. Challenges Regarding the Applicability in an Industrial Context

Afterward, the participants were divided into three groups (Group 1 with 13 participants, Group 2 with 11 participants and Group 3 with 15 participants), each moderated by a member of the AIS institute. First, the remaining questions regarding the presentation were clarified to enable the participants to assess the applicability and usefulness of the procedure voluntarily. A summary of the qualitative ratings is depicted in Figure 36, whereby the answers from the different groups are illustrated in different colors.

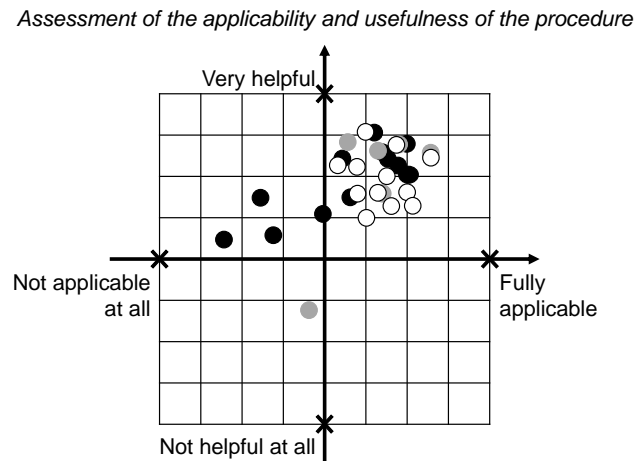


Figure 36: *Qualitative assessment of the applicability and usefulness of the procedure for integrating quality-assuring means into the software development process (Group 1 = black (12 replies), Group 2 = grey (6 replies), Group 3 = white (12 replies)), adapted from [Fis⁺22a].*

In the subsequent group discussions, experts indicated challenges hindering the procedure's applicability and ways to improve its usefulness. Since Group 2 mainly discussed tool-related aspects, no additional insights regarding the proposed procedure were gained and it is not explicitly mentioned in the following. The main tool-related challenge raised independently in all three groups is the mix of programming platforms used in the investigated company, including the analysis of the correctness of the data exchange between these platforms. Thereby, especially concerns regarding a tool capable of analyzing control software across different platforms were raised. The current organization within the company's software development department was seen as an additional obstacle. For a successful application of static code analysis, the participants highlighted the need for assigning a team responsible for analysis, documentation and subsequent software improvement. There is a concern that, otherwise, there is no capacity available in a software developer's day-to-day operations to resolve the identified weaknesses. Despite these organizational concerns, the participants of Group 1 confirmed that an integration of the assessment procedure into their current development workflow would be possible and even indicated an appropriate point for this. Since Group 1 rated the applicability and usefulness most skeptical of all groups (cf. black dots in Figure 36), the requirement R_{Work} is considered fulfilled.

Moreover, the participants of Group 1 stated that they have little experience with static code analysis, but they once tried to analyze their software's structure in the Siemens TIA Portal. Due to the different stakeholders involved, e.g., developers, designers, and commissioners, and their different requirements concerning the software, it was challenging for the experts to interpret the results and distinguish between positive and negative software ratings. Similarly, an industrial expert from Group 3 thinks that defining the software architecture and clear goals in advance is a prerequisite to setting evaluation criteria and focal points for the quality assessment with static

code analysis. Otherwise, different stakeholders will come to different results. However, once a company has defined these, the expert rates the presented procedure as “very strong” to support software quality management. This statement confirms the importance of a structured, goal-oriented quality assessment procedure and the need to target a specific analysis goal while considering the company’s design decisions and boundary conditions (cf. R_{PP} , R_{DD}).

Regarding the usefulness of the procedure and the analysis results, the experts in all groups highlighted that (automatically) analyzing and documenting the software’s conformance to the company’s programming guidelines would be beneficial. However, the current guidelines are partly formulated flexibly, which leads to different interpretations among the software developers. A suggestion of the experts to overcome this obstacle is to differentiate between software parts, where the rules must be strictly followed, and less critical parts. These strict rules could then be inserted via an input mask into an analysis tool for the subsequent automated checking, which would also enable a cross-departmental comparison (e.g., between machine types) of guideline conformance. Additionally, Group 1 indicated that the comparison of a machine’s control software versions would be helpful to detect incorrect use of the template. This application scenario of the assessment procedure with a focus on dependencies between POUs is described in [Fis⁺21a].

Concerning relevant aspects and limitations of the presented procedure, the experts in Group 1 indicated that analyzing, visualizing and documenting the dependencies between software parts are essential to estimate the impact, i.e., required effort and cross effects, after a software change. Thus, during static analysis, two aspects of modularity should be considered: reuse and the possibility to add or remove modules to or from the system. Consequently, the analysis documentation should support an assessment of the impact when a module is removed from the software, especially if the software is still functional. In Group 3, the technical applicability of the quality assessment procedure is rated high as it is „helpful to get a gut feeling“. However, the industrial expert does not expect automated suggestions from an analysis tool since it lacks the required domain knowledge. According to another expert, identifying the highest pain points and appointing resources for their refactoring is a strategic question that cannot be answered automatically. These expert opinions confirm the importance of domain knowledge during the quality assessment of control software, which is addressed by combining automatic analysis and manual interpretation in the proposed quality assessment procedure (cf. R_{Rat}).

A limitation of the presented prototype was highlighted in Group 1. According to the participants, hardware dependency plays a significant role in software assessment. However, module definitions across these disciplines do not always match since each has different perceptions about mod-

ule boundaries. From the expert’s point of view, this aspect is still missing in the analysis. Similarly, a participant in Group 3 highlighted that the analysis focus is on a technical review of code and that the essential aspect of *functional correctness* is not considered. To resolve this, the expert suggests including a model-based description of the automation hardware to amend the analysis with information regarding the required functionality.

To improve the usefulness, the experts in all groups expressed great interest in best practices, which would be helpful for initially designing high-quality software. To increase the applicability of the presented prototype, an expert from Group 1 suggested that “the analysis could identify used design patterns in a program, represent them abstractly and compare them with the planned structure.” Furthermore, performing the static analysis within the PLC’s IDE to provide feedback while editing the control code would be helpful. Finally, including a user interface to insert company-specific programming guidelines in the code analysis tool for a subsequent conformance check was suggested. Meanwhile, the *advacode* prototype was extended to enable conformance checks to company-specific rules in the data exchange (cf. [Fis+22b] for details).

Despite the experts’ concerns regarding the applicability of the procedure in an industrial context, in a final question at the end of the workshop (Question W#9), they confirmed that the proposed procedure eases the application of static code analysis (cf. Figure 37). Despite the specific boundary conditions and regulations in the food & beverage sector, the industry experts confirmed the proposed procedure’s suitability to support applying static code analysis for the quality assessment of control software in an industrial context.

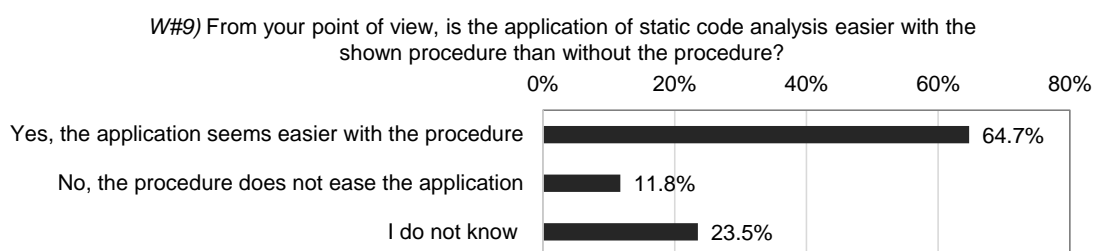


Figure 37: Answers to W#9 regarding the application of static code analysis with and without the proposed procedure (total answers $n = 34$); pseudo accuracy to avoid rounding error.

Comparing the answers to the asked questions (cf. Table 22), it shows that the percentage of responses selecting “*I don’t know*” varies from 5% (W#2, helpfulness of interview guiding questions) to 45% (W#4, consideration of sector-specific constraints). A reason for this might be the time spent on introducing the different topics and the required knowledge of control software and process details for answering the respective questions. The interview guiding questions were introduced with specific examples, including their aim, and do not require expert knowledge of the

programming of control software to be answered. Thus, workshop participants with an electrical engineering background and from the management level could answer the question. On the contrary, details of the control software of the *Self-X Material Flow Demonstrator* were used to show the adaptability of the proposed procedure to different boundary conditions and constraints. Thus, knowledge of PLC programming, including the transfer to the control software projects of the company and their boundary conditions, was required to answer the respective question *W#4*. It is likely that not all participants from management or electrical engineering had the necessary background knowledge of the software, including details of the constraints in the food & beverage sector that affect the control software, to make the transfer and answer the question. Similarly, the high amount of participants, who estimate that they could not independently apply the procedure themselves or are unsure about it (51% of the responses to *W#7*), is in line with expectations: it requires detailed knowledge of PLC software programming and code analysis, but 65% of the participants do not yet apply static code analysis during the development (*W#1*).

Table 22: Comparing the company experts' answers to the questions asked during the workshop.

Answer	W#1	W#2 <i>R_{PP}</i>	W#3 <i>R_{DD}</i>	W#4 <i>R_{Sec}</i>	W#5 <i>R_{Doc}</i>	W#6 <i>R_{Weak}</i>	W#7 <i>R_{Use}</i>	W#8 <i>R_{Work}</i>	W#9 <i>overall</i>
Yes / partially	27.0%	92.3%	62.5%	47.3%	84.6%	80.0%	48.6%	67.6%	64.7%
No	64.9%	2.6%	10.0%	7.9%	2.6%	11.4%	13.5%	2.7%	11.8%
unsure	-	-	-	-	-	-	35.2%	29.7%	-
I don't know	8.1%	5.2%	27.5%	44.8%	12.8%	8.6%	2.7%	0%	23.5%

In summary, the assignment of human resources to and the organization of software quality control as well as the mix of platforms used are considered the most significant obstacles to successfully integrating the analysis procedure for quality assessment. Future steps concerning the procedure should focus on the required amount and depth of training to enable software developers to apply the procedure themselves. Furthermore, additional research is needed regarding the procedure's applicability to the boundary conditions of different application sectors. Suggestions for improving the presented quality assessment procedure and demonstrated prototypical tool include conceptual-wise the integration of automation hardware modules to assess the fulfilled functionalities within the software. Tool-wise, a user interface to insert company-specific guidelines and additional support in highlighting dependencies to estimate the impact of changes would increase the procedure's usefulness and applicability in an industrial context, according to the participants. Overall, the industrial experts confirmed the need to combine tool-based, static code analysis with human interpretation of the results, taking domain knowledge into account. The effort for applying the procedure to monolithic legacy software is estimated as very high. However, once a company has defined clear design strategies and rules, the presented quality assessment procedure is rated as "very strong" to support software quality management in an industrial context.

8. Assessment of the Fulfillment of the Requirements

The previous Chapters 6 and 7 discussed the findings of the prototypical implementation, the conducted case studies and expert evaluations (WG and workshop) and the fulfillment of the derived requirements (Chapter 3). These results are summarized in Table 23 to assess to what extent the proposed quality assessment procedure addresses the identified research gap. All requirements were evaluated positively with the conducted case studies. However, the industrial experts raised concerns regarding the independent application of the quality assessment procedure, the consideration of deliberate design decisions and the support to derive recommendations for action. Thus, future research should target the scope and amount of required training, the consideration of company-specific design decisions and stronger integration of approaches to derive recommendations.

Table 23: Summary of the evaluation of the presented procedure with respect to the requirements.

Requirement	Implementation (Section 6)	Case Studies (Section 7.1)*	Expert Evaluation (WG) (Section 7.2)	Expert Workshop in food and beverage sector (Section 7.3)	Overall Evaluation
R_{PLC} – Platform Independence		+	+ (WG#13)		+
R_{Pro} – aPS as Product		+	+ (WG#3, 7)		+
R_{Use} – User		+**	o (W#8)	o (W#7)	o
R_{Sec} – Application Sector		+	+ (WG#11, 12)	o (W#4)	+
R_{PP} – Pain Points		+	o (WG#6)	+ (W#2)	+
R_{Work} – Workflow Integration		+	+ (WG#3)	+ (W#8)	+
R_{DD} – Design Decision	+	+	o (WG#7)	o (W#3)	+/o
R_{Goal} – Analysis Goal		+			+
R_{Scal} – Scalability	+	+			+
R_{Eff} – Application Effort		+			+
R_{Rat} – Rationale	+	+			+
R_{Weak} – Weaknesses and Change Effort		+		o (W#6)	+/o
R_{Doc} – Documentation	+	+		+ (W#5)	+

Legend:

“+”: fully satisfied; “o”: partially satisfied; “-”: not satisfied; empty cell: not available.

*refer to Table 21, p. 129 for details regarding the fulfillment of the requirements by the conducted case studies.

** requirement is targeted and fulfilled by Case Study B only.

From the prototypical implementation and the conducted evaluations, an assessment of the proposed quality assessment procedure is performed. The procedure is designed platform-independent (R_{PLC}) and should be applicable regardless of the produced aPS type (R_{Pro}), which is confirmed by the conducted case studies and the WG evaluation. Although the four industrial case studies were conducted with companies using Siemens PLCs (SIMATIC STEP 7 and TIA Portal), *Case Study E* demonstrated the applicability of the quality assessment procedure to TwinCAT 3 control software. Moreover, half of the WG members answering the questionnaire indicated that they do not use Siemens PLCs at all in their company. Of the six WG participants answering that they would be able to use the procedure after an introduction ($WG\#8$), three do not use Siemens PLCs as platforms, one indicated the use of Siemens and IEC 61131-3-based PLCs and two use only Siemens PLCs in their companies. Thus, R_{PLC} is considered fulfilled, but as *Case Study E* is a lab-sized case study, at least two additional case studies with non-Siemens platforms should be conducted in future work.

The software developers' use of the proposed procedure is supported by the provided interview guiding questions and analysis checklist and was successfully demonstrated in *Case Study B*. However, due to the high effort for the involved companies, it was not targeted in any other case study. Nevertheless, the case study evaluation of requirement R_{Use} is considered fulfilled. Overall, the software developers' use of the procedure highly depends on the scope and amount of training and additional research is required, which leads to R_{Use} being considered partially satisfied. The case studies showed that the company-specific and application sector-specific boundary conditions could be considered during the analysis and quality assessment. The WG evaluation confirmed it as well (R_{Sec}). Nevertheless, future work should focus on application sectors with specific boundary conditions since the workshop participants working in the food and beverage sector were unsure regarding the constraints of their application sector.

The evaluations confirmed that the proposed assessment procedure supports identifying pain points as the basis for deriving the analysis goal (R_{PP}). Furthermore, integrating the procedure into the development workflow is rated as feasible (R_{Work}). However, without proper tool support, the application requires too much additional effort and is, thus, not combinable with the developers' day-to-day tasks. While the case studies confirmed that deliberate design decisions regarding the control software could be considered in the analysis, the WG and the workshop participants raised concerns (R_{DD}). Thus, the requirement is considered partially fulfilled and future work regarding the procedure's limitations is necessary.

Conducting the static code analysis for a quality assessment in the five case studies and utilizing the *advacode* prototype confirmed that the procedure supports targeting different analysis goals

(R_{Goal}) and that it is applicable to industry-sized control software (R_{Scal}). While the evaluation demonstrated that the procedure is suitable to assess the quality of industry-sized control software, it also showed that the effort required to perform the static code analysis in Step 2 is not feasible in industrial use without tool support and, furthermore, that the current tool-support is not sufficient. Nevertheless, the proposed procedure supports conducting static code analysis of industry-sized control software systematically and goal-oriented, regardless of the specific analysis goal. While the procedure points out possible analysis methods and aspects to be targeted, it is not limited to the suggested ones. Moreover, some procedure steps do not have to be repeated if a similar analysis goal is considered. For example, familiarization with the software architecture and available analysis methods, targeted aspects and types of documentation is performed in the first iteration of the procedure. The gained knowledge can be used in further procedure applications, which reduces the required familiarization time (R_{Eff}). However, if control software that follows different guidelines and a new analysis goal are targeted in the second application, the effort reduction is lower. Nevertheless, it is still not as high as during the first application if the software developers from the first application of the procedure are involved, who are already familiar with the procedure's steps and available analysis means. While applying automatic means for static code analysis whenever possible is suggested, the case studies confirm the importance of including the manual assessment of the gained insights for rating the software's quality while considering boundary conditions, constraints and design decisions (R_{Rat}).

The procedure evaluation with case studies confirmed that weaknesses could be identified and, further, the required change effort to overcome these can be qualitatively estimated (R_{Weak}). However, the workshop participants were not completely convinced that the procedure enables deriving recommendations for action. Consequently, R_{Weak} is not considered completely fulfilled. Future research should target how available approaches like the goal-lever-indicator-principle can be enhanced or even automated, e.g., in the context of an assistance system, to integrate it closer into the procedure and derive recommendations for action. Finally, documenting the analysis results and insights gained is rated fully satisfied (R_{Doc}).

Comparing the questionnaire answers shows that the responses from the workshop are less convincing than the answers from the WG. Different factors might cause this. On the one hand, the WG members discussed modularity and code analysis several times with different foci in the previous WG meetings. Additionally, some WG members were previously involved in research projects with the AIS institute targeting modularity, static code analysis and reuse. In contrast, 65% of the workshop participants stated that they do not apply any means of static code analysis at all during the software development. Thus, the WG members have more background knowledge and experience with means for quality assessment and transferring the presented procedure to their

control software is expected to be easier for them. On the other hand, the workshop developers work in the application sector of food processing. Since the boundary conditions, laws and regulations in this sector are quite strict, they might require specific design decisions and, thus, hinder the applicability of the presented procedure. Nevertheless, the workshop participants were more convinced of the interview guiding questions than the WG members. This estimation might result from the fact that the discussion of interview guiding questions in the workshop was more detailed than in the WG meeting. In conclusion, further research is needed on the amount of training, especially when software developers are unfamiliar with the means for quality assessment, and on the adaptability to application sector-specific constraints.

Overall, the assessment of the two requirements R_{Goal} and R_{Eff} is limited to a qualitative evaluation within the detailed case studies. Although the case studies represent different aPS characteristics, some boundary conditions might not have been included, which might require adaptation of the quality assessment procedure. Moreover, the insights gained from the questionnaires conducted in the WG meeting and the workshop are closely linked to the presented procedure excerpts demonstrated with the *Self-X Material Flow Demonstrator* from the AIS institute. Although this demonstrator was originally an industrial testbed, the focus on an intralogistics system might have biased the participant's responses to the questionnaire. Thus, additional case studies should be conducted with selected WG and workshop participants to compare their questionnaire responses with their assessments after applying the procedure to the control software they work with, ideally in application sectors with specific constraints, e.g., the food and beverage sector or MedTech.

9. Summary and Outlook

Control software realizes an increasing proportion of system functionality in aPS. This requires the reuse of high-quality control software to remain competitive in the global market and deliver evolvable, maintainable and reconfigurable systems that meet the requirements of Industry 4.0. While static code analysis is commonly used to ensure software quality in the computer science domain, it is not yet standardly applied in the aPS domain. Moreover, to gain valuable and meaningful results, static code analysis needs to be adapted and tailored to the domain's boundary conditions, including application sector- and company-specific factors.

Accordingly, this thesis proposed a quality assessment procedure for control software in the aPS domain, including the derivation of recommendations for action from the gained analysis results to improve the software quality by addressing identified weaknesses. The goal-oriented procedure and the additional material, e.g., interview guiding questions, exemplary analysis goals, an analysis checklist, and means for visualizing and documenting analysis results, support control software developers to conduct the static code analysis independently and without needing to consult external experts. Moreover, the procedure suggests performing the assessment as an automatic and manual static code analysis. The combination enables coping with the size of industrial control software projects while considering the rationale, deliberate design decisions and domain knowledge in the assessment.

The presented quality assessment procedure was successfully applied in four industrial case studies and a lab-sized demonstrator case study. Additionally, it has been evaluated by two expert groups, i.e., in an industrial working group meeting and a workshop with software developers and managers from a company operating in the food and beverage sector. Overall, the evaluation confirmed that the developed procedure enables a goal-oriented and systematic use of available means for control software quality assessment in an industrial context. It further supports the identification of improvement potential and deriving recommendations for action for overcoming identified weaknesses. Moreover, the industrial experts agreed that the procedure eases the application of static code analysis if the integration into the company workflow succeeds, which requires allocating resources to the quality assessment and tool support for combining the analysis with day-to-day tasks. Thus, the research question stated in the introduction has been successfully answered and the proposed quality assessment procedure addresses the research gap by fulfilling most requirements.

However, the proposed procedure does not yet fully satisfy all derived requirements, which should be addressed in future research. For example, training and workshops are essential to enable software developers to independently perform a quality assessment of their control software (R_{Use}). The amount and scope of the required training highly depend on the background knowledge of the involved software developers and the targeted analysis goal. Accordingly, the definition of training concepts needs to be investigated in future research. One way to address this is to perform additional workshops with different companies to determine the amount of required training. Another potential starting point is the definition of a catalog containing a classification of available reuse approaches for PLC control software linked to company-specific boundary conditions. More precisely, such a catalog would support the identification of reuse strategies suitable for application in the scope of the company's boundary conditions as started in [Neu+20c; Neu+22]. Based on the catalog, the scope and amount of training could be derived, which are required to transform a company's current development workflow and reuse strategy into the selected solution. Moreover, developing means for automated support during the procedure application, e.g., as an assistant system, is expected to reduce the amount of training required since the software developer could be guided during the quality assessment.

The assessment of R_{Scal} demonstrated that the current tool support is insufficient for the successful, efficient application of the proposed procedure in industrial practice. There is a great need for tool support for the automatic static code analysis of industry-sized control software. Ideally, supportive analysis means should be directly implemented into the PLC development environment to integrate them seamlessly into the development workflow and, thus, enable a continuous assessment during the development process [DP12] and, at the same time, avoid interruptions in the developer's workflow [NNB19]. This requires additional research regarding suitable visualizations, analysis means and software metrics capable of dealing with the high complexity and size of industry-sized control software. Integration of analysis means into the PLC development environment would also enable measuring and, ideally, quantifying the quality improvement gained through the procedure application, which has not been targeted in the scope of this thesis. Estimating the expected benefits in terms of time or money would also make it easier to decide which recommendations for action with a high level of effort should nonetheless be implemented.

While the proposed quality assessment enables the identification of disadvantageous design decisions and software structures, it does not support the greenfield development of high-quality control software but is intended for existing legacy software. Nevertheless, the insights gained and lessons learned from the procedure applications could be used to derive quality attributes that control software should meet, including best practices for meeting these characteristics, in future

work. During the expert discussions, the software developers confirmed the need for design patterns and best practices to support the development of high-quality control software in advance, rather than only analyzing existing software to identify improvement potentials. In turn, best practices and design patterns could be used to establish refactoring guidelines. These would enable industry experts to derive recommendations for action from the quality assessment results even if they have little background knowledge on static code analysis (R_{Weak}).

Moreover, the development of best practices and the definition of quality characteristics to be fulfilled by Industry 4.0-enabling control software would make conflicts between different design decisions explicit, which is a prerequisite for strategies to address and, ideally, resolve them. Static code analysis could be combined with a questionnaire-based approach such as [VO18] to enhance the maturity assessment of control software. A questionnaire could serve as a starting point of the maturity assessment to identify boundary conditions such as the development workflow, applied reuse strategies and experience and background of involved stakeholders. Moreover, it can serve as a benchmark to compare a company's development process and control software maturity to the average maturity. Subsequently, identified weaknesses would be the starting point for detailed static code analysis. Combining a company's rating of both questionnaire and detailed analysis could enable assigning the company to a maturity level similar to the approach of Antkiewicz et al. from computer science [Ant⁺14]. Derived recommendations could then illustrate which actions must be performed to lift the control software maturity to the next higher maturity level. Additionally, this benchmark could serve as a certification of software quality, which companies' could show to their customers.

Finally, the concept proposed within this thesis is tailored to and evaluated with classical IEC 61131-3 control software and discrete processes. Generally, it should be applicable to OO IEC with minor changes, but further investigations are required to derive necessary adaptations of the concept. The same holds true for the application to control software of continuous processes, which face different and additional boundary conditions to be considered in the quality assessment. Thus, additional case studies should be performed to confirm the transferability of the approach to OO IEC and different process types.

10. Literature

- [ABB21] ABB, "Safety Code Analysis (SCA)" [Online] Available: https://library.e.abb.com/public/3f33689dcd904e41a27e7b5947ca7166/3ADR010489_SCA_Read_Me,%203,%20en_US.pdf, [Accessed: 10-05-22], 2021.
- [AFV22] Aicher, T., Fottner, J. and Vogel-Heuser, B., "A model-driven engineering design process for the development of control software for Intralogistics Systems," In: *at – Automatisierungstechnik*, vol. 70, no. 2, pp. 164–180, 2022.
- [Alv⁺12] Alvarez, M. L., Burgos, A., Sarachaga, I., Estévez, E. and Marcos, M., "GEMMA based approach for generating PLCopen Automation projects," In: *IFAC Proceedings Volumes*, vol. 45, no. 4, pp. 230–235, 2012.
- [An⁺21] An, Y., Qin, F., Chen, B., Simon, R. and Wu, H., "OntoPLC: Semantic Model of PLC Programs for Code Exchange and Software Reuse," In: *IEEE Transactions on Industrial Informatics (TII)*, vol. 17, no. 3, pp. 1702–1711, 2021.
- [Ang⁺13] Angerer, F., Prähofer, H., Ramler, R. and Grillenberger, F., "Points-to analysis of IEC 61131-3 programs: Implementation and application," In: *IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA)*: IEEE, pp. 1–8, 2013.
- [Ant⁺14] Antkiewicz, M., Ji, W., Berger, T., Czarnecki, K., Schmorleiz, T., Lämmel, R., Stănculescu, Ș., Wąsowski, A. and Schaefer, I., "Flexible product line engineering with a virtual platform," In: Pankaj Jalote, Lionel Briand and André van der Hoek (Eds.): *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE)*. New York, NY, USA: ACM, pp. 532–535, 2014.
- [Arm⁺18] Armentia, A., Estévez, E., Orive, D. and Marcos, M., "A Tool Suite for Automatic Generation of Modular Machine Automation Projects," In: *IEEE 16th International Conference on Industrial Informatics (INDIN)*: IEEE, pp. 553–558, 2018.
- [Bau⁺04] Bauer, N., Huuck, R., Lukoschus, B. and Engell, S., "A Unifying Semantics for Sequential Function Charts," In: David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell et al. (Eds.): *Integration of Software Specification Techniques for Applications in Engineering*, Bd. 3147. Berlin, Heidelberg: Springer Berlin Heidelberg (Lecture Notes in Computer Science), pp. 400–418, 2004.
- [BBF15] Barbieri, G., Battilani, N. and Fantuzzi, C., "A PackML-based Design Pattern for Modular PLC Code," In: *IFAC-PapersOnLine*, vol. 48, no. 10, pp. 178–183, 2015.
- [BBK12] Biallas, S., Brauer, J. and Kowalewski, S., "Arcade.PLC: a verification platform for programmable logic controllers," In: Michael Goedicke, Tim Menzies and Motoshi Saeki (Eds.): *27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. New York, NY, USA: ACM Press, pp. 338, 2012.

- [BD02] Bansiya, J. and Davis, C. G., "A hierarchical model for object-oriented design quality assessment," In: *IEEE Transactions on Software Engineering*, vol. 28, no. 1, pp. 4–17, 2002.
- [Bec21] Beckhoff Automation GmbH & Co. KG, "TE1200 - TwinCAT 3 | PLC Static Analysis (Manual), Version: 2.5" [Online] Available: https://download.beckhoff.com/download/document/automation/twincat3/TE1200_TC3_PLC_Static_Analysis_EN.pdf, [Accessed: 09-05-22], 2021.
- [Bec22] Beckhoff Information System, "Code analysis (Static Analysis)" [Online] Available: https://infosys.beckhoff.com/english.php?content=/.content/1033/tc3_plc_intro/2527107467.html&id=, [Accessed: 09-05-22], 2022.
- [Ber19] Berscheit, A., "Analyse und Dokumentation der Variabilität in historisch gewachsenen Steuerungssoftwareprojekten aus der Intralogistik [Analysis and Documentation of Variability in Historically Grown Control Software Projects from Intralogistics]," Bachelor's Thesis, Technical University of Munich, 2019.
- [BF03] Bonfè, M. and Fantuzzi, C., "Design and verification of industrial logic controllers with UML and statecharts," In: *IEEE Conference on Control Applications (CCA)*: IEEE, pp. 1029–1034, 2003.
- [BFS13] Bonfè, M., Fantuzzi, C. and Secchi, C., "Design patterns for model-based automation software design and implementation," In: *Control Engineering Practice (CEP)*, vol. 21, no. 11, pp. 1608–1619, 2013.
- [BG21] Barbieri, G. and Gutierrez, D. A., "A GEMMA-GRAFCET Methodology to enable Digital Twin based on Real-Time Coupling," In: *Procedia Computer Science*, vol. 180, pp. 13–23, 2021.
- [Bia16] Biallas, S., "Verification of programmable logic controller code using model checking and static analysis," Dissertation, RWTH Aachen University, Aachen, 2016.
- [Bif⁺15] Biffl, S., Maetzler, E., Wimmer, M., Lueder, A. and Schmidt, N., "Linking and versioning support for AutomationML: A model-driven engineering perspective," In: *IEEE 13th International Conference on Industrial Informatics (INDIN)*. Piscataway, NJ: IEEE, pp. 499–506, 2015.
- [Bou⁺17] Bougouffa, S., Diehm, S., Schwarz, M. and Vogel-Heuser, B., "Scalable cloud based semantic code analysis to support continuous integration of industrial PLC code," In: *IEEE 15th International Conference on Industrial Informatics (INDIN)*: IEEE, pp. 621–627, 2017.
- [Bou⁺19] Bougouffa, S., Vogel-Heuser, B., Fischer, J., Schaefer, I. and Li, H., "Visualization of Variability Analysis of Control Software From Industrial Automation Systems," In: *IEEE International Conference on Systems, Man and Cybernetics (SMC)*, pp. 3357–3364, 2019.

- [Can⁺21] Canedo, A., Goyal, P., Di Huang, Pandey, A. and Quiros, G., "ArduCode: Predictive Framework for Automation Engineering," In: *IEEE Transactions on Automation Science and Engineering (TASE)*, vol. 18, no. 3, pp. 1417–1428, 2021.
- [Cas⁺21] Castillo, J. M., Barbieri, G., Mejia, A., Hernandez, J. D. and Garces, K., "A GEMMA-GRAFCET Generator for the Automation Software of Smart Manufacturing Systems," In: *Machines*, vol. 9, no. 10, pp. 232, 2021.
- [CK94] Chidamber, S. R. and Kemerer, C. F., "A metrics suite for object oriented design," In: *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.
- [CLÅ06] Cengic, G., Ljungkrantz, O. and Åkesson, K., "Formal Modeling of Function Block Applications Running in IEC 61499 Execution Runtime," In: *IEEE Conference on Emerging Technologies & Factory Automation (ETFA)*: IEEE, pp. 1269–1276, 2006.
- [COD19] CODESYS GmbH - Online Help, "Static Analysis" [Online] Available: https://product-help.schneider-electric.com/Machine%20Expert/V1.1/en/LibDevSummary/topics/static_analysis.htm, [Accessed: 01-05-22], 2019.
- [COD22] CODESYS GmbH, "CODESYS Static Analysis" [Online] Available: <https://store.codesys.com/en/codesys-static-analysis.html>, [Accessed: 01-05-22], 2022.
- [CQS22] CQSE GmbH, "Teamscale" [Online] Available: <https://www.cqse.eu/en/teamscale/overview/>, [Accessed: 05-06-22], 2022.
- [CV17] Capitán, L. and Vogel-Heuser, B., "Metrics for software quality in automated production systems as an indicator for technical debt," In: *13th IEEE Conference on Automation Science and Engineering (CASE)*: IEEE, pp. 709–716, 2017.
- [CW07] Chess, B. and West, J., "Secure programming with static analysis." Upper Saddle River, NJ: Addison-Wesley (Software security series), 2007.
- [Cza98] Czarnecki, K., "Generative Programming. Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models," Dissertation, Technical University of Ilmenau. Department of Computer Science and Automation, 1998.
- [DFS06] Drath, R., Fay, A. and Schmidberger, T., "Computer-aided design and implementation of interlock control code," In: *IEEE Conference on Computer Aided Control System Design, IEEE International Conference on Control Applications, IEEE International Symposium on Intelligent Control*: IEEE, pp. 2653–2658, 2006.
- [DP12] Dondey, H. and Peron, C., "Software Qualimetry at Schneider Electric: a field background," In: *Proceedings of the 6th European Congress on Embedded Real Time Software and Systems (ERTS)*, 2012.

- [DZ21] Dorninger, B. and Ziebermayr, T., "Software quality assurance in mechanical and plant engineering: automated assessment of the technical quality of PLC code [Software-Qualitätssicherung im Maschinen- und Anlagenbau: automatisierte Bewertung der technischen Qualität von SPS-Code]," In: *e&i (Elektrotechnik und Informationstechnik)*, 2021.
- [EDL07] Engell, S., Dandachi, A. and Lohmann, S., "Impact of Complexity on Logic Controller Design," In: *IFAC Proceedings Volumes*, vol. 40, no. 6, pp. 121–126, 2007.
- [EMO07] Estévez, E., Marcos, M. and Orive, D., "Automatic generation of PLC automation projects from component-based models," In: *The International Journal of Advanced Manufacturing Technology (IJAMT)*, vol. 35, no. 5-6, pp. 527–540, 2007.
- [EN08] Emanuelsson, P. and Nilsson, U., "A Comparative Study of Industrial Static Analysis Tools," In: *Electronic Notes in Theoretical Computer Science*, vol. 217, pp. 5–21, 2008.
- [Fad⁺22] Fadhilillah, H. S., Feichtinger, K., Meixner, K., Sonnleithner, L., Rabiser, R. and Zoitl, A., "Towards Multidisciplinary Delta-Oriented Variability Management in Cyber-Physical Production Systems," In: Paolo Arcaini, Xavier Devroey and Alessandro Fantechi (Eds.): *Proceedings of the 16th International Working Conference on Variability Modelling of Software-Intensive Systems*. New York, NY, USA: ACM, pp. 1–10, 2022.
- [Fag76] Fagan, M. E., "Design and code inspections to reduce errors in program development," In: *IBM Systems Journal*, vol. 15, no. 3, pp. 182–211, 1976.
- [Fah⁺19] Fahimipirehgalin, M., Fischer, J., Bougouffa, S. and Vogel-Heuser, B., "Similarity Analysis of Control Software Using Graph Mining," In: *IEEE 17th International Conference on Industrial Informatics (INDIN)*: IEEE, pp. 508–515, 2019.
- [Fan⁺15] Fang, M., Leyh, G., Doerr, J., Elsner, C. and Zhao, J., "Towards model-based derivation of systems in the industrial automation domain," In: Douglas C. Schmidt (Eds.): *Proceedings of the 19th International Conference on Software Product Line*. New York, NY: ACM, pp. 283–292, 2015.
- [Fel⁺16a] Feldmann, S., Hauer, F., Ulewicz, S. and Vogel-Heuser, B., "Analysis framework for evaluating PLC software: An application of Semantic Web technologies," In: *IEEE 25th International Symposium on Industrial Electronics (ISIE)*: IEEE, pp. 1048–1054, 2016.
- [Fel⁺16b] Feldmann, S., Ulewicz, S., Diehm, S. and Vogel-Heuser, B., "Structural code analysis - Analysis framework using semantic web technologies [Strukturelle Codeanalyse: Analyseframework mittels Semantic-Web-Technologien]," In: *atp magazin*, vol. 58, no. 09, 42-51, 2016.
- [Fer⁺15] Ferreira, R., Blanchard, S., Gomes, P. and Vestergard, H., "Consolidation of the control system of a chemical polishing machine for superconducting RF cavities

- using the UNICOS-CPC framework," In: *IEEE International Conference on Automation Science and Engineering (CASE)*: IEEE, pp. 1471–1476, 2015.
- [Fis⁺14] Fischer, S., Linsbauer, L., Lopez-Herrejon, R. E. and Egyed, A., "Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants," In: *IEEE International Conference on Software Maintenance and Evolution (ICSME)*: IEEE, pp. 391–400, 2014.
- [Fis⁺15] Fischer, S., Linsbauer, L., Lopez-Herrejon, R. E. and Egyed, A., "The ECCO Tool: Extraction and Composition for Clone-and-Own," In: *2nd ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*: IEEE, pp. 665–668, 2015.
- [Fis⁺18] Fischer, J., Bougouffa, S., Schlie, A., Schaefer, I. and Vogel-Heuser, B., "A Qualitative Study of Variability Management of Control Software for Industrial Automation Systems," In: *IEEE International Conference on Software Maintenance and Evolution (ICSME)*: IEEE, pp. 615–624, 2018.
- [Fis⁺20a] Fischer, J., Vogel-Heuser, B., Wilch, J., Loch, F., Land, K. and Schaefer, I., "Variability Visualization of IEC 61131-3 Legacy Software for Planned Reuse," In: *IEEE International Conference on Systems, Man, and Cybernetics (SMC)*: IEEE, pp. 3760–3767, 2020.
- [Fis⁺20b] Fischer, J., Vogel-Heuser, B., Haben, F. and Schaefer, I., "Reengineering Workflow for Planned Reuse of IEC 61131-3 Legacy Software," In: *IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*: IEEE, pp. 1126–1130, 2020.
- [Fis⁺20c] Fischer, J., Lieberoth-Leden, C., Fottner, J. and Vogel-Heuser, B., "Design, Application, and Evaluation of a Multiagent System in the Logistics Domain," In: *IEEE Transactions on Automation Science and Engineering (TASE)*, vol. 17, no. 3, pp. 1283–1296, 2020.
- [Fis⁺21a] Fischer, J., Vogel-Heuser, B., Huber, C., Felger, M. and Bengel, M., "Reuse Assessment of IEC 61131-3 Control Software Modules Using Metrics – An Industrial Case Study," In: *IEEE 19th International Conference on Industrial Informatics (INDIN)*: IEEE, pp. 1–8, 2021.
- [Fis⁺21b] Fischer, J., Vogel-Heuser, B., Schneider, H., Langer, N., Felger, M. and Bengel, M., "Measuring the Overall Complexity of Graphical and Textual IEC 61131-3 Control Software," In: *IEEE Robotics and Automation Letters (RAL)*, vol. 6, no. 3, pp. 5784–5791, 2021.
- [Fis⁺21c] Fischer, J., Vogel-Heuser, B., Berscheit, A. and Parigger, S., "Comparison of Two Concepts for Planned Reuse of Variant-rich IEC 61131-3-based Control Software," In: *IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*, 713–720, 2021.

- [Fis⁺22a] Fischer, J., Neumann, E.-M., Wilch, J., Obermeier, M., Kellhammer, T. and Vogel-Heuser, B., "Preliminary Evaluation Results of Static Code Analysis of Control Software in an Industrial Context" [Online] Available: <https://mediatum.ub.tum.de/doc/1658416/1658416.pdf>, [Accessed: 11-05-22], 2022.
- [Fis⁺22b] Fischer, J., Vogel-Heuser, B., Haben, F., Beuggert, L. and Neumann, E.-M., "Towards Configurable Conformance Checks of PLC Software with Company-specific Guidelines," In: *5th IEEE International Conference on Industrial Cyber-Physical Systems (ICPS)*: IEEE, pp. 1–8, 2022.
- [Fol⁺11] Follmer, M., Hehenberger, P., Punz, S., Rosen, R. and Zeman, K., "Approach for the Creation of Mechatronic System Models," In: *Proceedings of the 18th International Conference on Engineering Design (ICED)*, pp. 258–267, 2011.
- [FSB11] Fantuzzi, C., Secchi, C. and Bonfè, M., "A Design Pattern for translating UML software models into IEC 61131-3 Programming Languages," In: *IFAC Proceedings Volumes*, vol. 44, no. 1, pp. 9158–9163, 2011.
- [Fuc⁺14] Fuchs, J., Feldmann, S., Legat, C. and Vogel-Heuser, B., "Identification of Design Patterns for IEC 61131-3 in Machine and Plant Manufacturing," In: *IFAC Proceedings Volumes*, vol. 47, no. 3, pp. 6092–6097, 2014.
- [FV17] Feldmann, S. and Vogel-Heuser, B., "Interdisciplinary product lines to support the engineering in the machine manufacturing domain," In: *International Journal of Production Research*, vol. 55, no. 13, pp. 3701–3714, 2017.
- [FVF15] Fischer, J., Vogel-Heuser, B. and Friedrich, D., "Configuration of PLC software for automated warehouses based on reusable components- an industrial case study," In: *IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA)*: IEEE, pp. 1–7, 2015.
- [Gam⁺95] Gamma, E., Helm, R., Johnson, R. and Vlissides, J., "Design Patterns. Elements of Reusable Object-Oriented Software." Reading, Mass.: Addison-Wesley (Addison-Wesley professional computing series), 1995.
- [GC15] Gîrba, T. and Chiş, A., "Pervasive software visualizations (keynote)," In: *IEEE 3rd Working Conference on Software Visualization (VISSOFT)*: IEEE, pp. 1–5, 2015.
- [GEP22a] GEPRIS - projects funded by German Research Association (DFG), "Reverse Engineering Design of Software Product Lines for Automation Technology (RED SPLAT)" [Online] Available: <https://gepris.dfg.de/gepris/projekt/335427442?language=en>, [Accessed: 15-05-22], 2022.
- [GEP22b] GEPRIS - projects funded by German Research Association (DFG), "Increased flexibility for heterogeneously structured material flow systems enabled by intelligent software agents controlling self-configuring conveyors" [Online] Available: <https://gepris.dfg.de/gepris/projekt/251665026?language=en>, [Accessed: 15-05-22], 2022.

- [Gha06] Gharieb, W., "Software Quality in Ladder Programming," In: *International Conference on Computer Engineering and Systems: IEEE*, pp. 150–154, 2006.
- [GMP] European Commission, "EU Guidelines for Good Manufacturing Practice for Medicinal Products for Human and Veterinary Use: GMP" [Online] Available: https://ec.europa.eu/health/medicinal-products/eudralex/eudralex-volume-4_en, [Accessed: 07-03-22].
- [Gup⁺14] Gupta, S., Singh, H. K., Venkatasubramanyam, R. D. and Uppili, U., "SCQAM: a scalable structured code quality assessment method for industrial software," In: Chanchal K. Roy, Andrew Begel and Leon Moonen (Eds.): *Proceedings of the 22nd International Conference on Program Comprehension - ICPC 2014*. New York, New York, USA: ACM Press, pp. 244–252, 2014.
- [GWF08] Güttel, K., Weber, P. and Fay, A., "Automatic generation of PLC code beyond the nominal sequence," In: *IEEE International Conference on Emerging Technologies & Factory Automation (ETFA)*: IEEE, pp. 1277–1284, 2008.
- [Hal77] Halstead, M. H., "Elements of software science." New York, NY, USA: Elsevier Science Inc. (Operating and programming systems series), 1977.
- [Han15] Hanssen, D. Håkon, "Programmable logic controllers. A practical approach to IEC 61131-3 using CODESYS." Chichester, West Sussex: Wiley, 2015.
- [Hin⁺18] Hinterreiter, D., Prähofer, H., Linsbauer, L., Grünbacher, P., Reisinger, F. and Egyed, A., "Feature-Oriented Evolution of Automation Software Systems in Industrial Software Ecosystems," In: *IEEE 23rd International Conference on Emerging Technologies & Factory Automation (ETFA)*: IEEE, pp. 107–114, 2018.
- [HK81] Henry, S. and Kafura, D., "Software Structure Metrics Based on Information Flow," In: *IEEE Transactions on Software Engineering*, vol. SE-7, no. 5, pp. 510–518, 1981.
- [Hom⁺11] ten Hompel, M., Nettsträter, A., Feldhorst, S. and Schier, A., "Engineering of Modular Material Flow Systems in the Internet of Things [Engineering von modularen Förderanlagen im Internet der Dinge]," In: *at – Automatisierungstechnik*, vol. 59, no. 4, pp. 248–256, 2011.
- [Hub20] Huber, C., "Analysis of the version history of an IEC 61131-3-based control project with special consideration of modularity and criteria for change evaluation," Bachelor's Thesis, Technical University of Munich, 2020.
- [HVA16] Harrison, R., Vera, D. and Ahmad, B., "Engineering Methods and Tools for Cyber-Physical Automation Systems," In: *Proc. IEEE*, vol. 104, no. 5, pp. 973–985, 2016.
- [IEC61131-10] IEC 61131-10, 2019, "Programmable controllers - Part 10: PLC open XML exchange format".
- [IEC61131-3] IEC 61131-3, 2013, "Programmable controllers - Part 3: Programming languages".

- [IEC61131-8] IEC 61131-8, 2017, "Industrial-process measurement and control - Programmable controllers - Part 8: Guidelines for the application and implementation of programming languages".
- [IEC61499] IEC 61499-1, 2013, "Function blocks - Part 1: Architecture".
- [IEC61512] IEC 61512, 1997, "Batch control - Part 1: Models and terminology".
- [IEC81346] IEC 81346-2, 2019, "Industrial systems, installations and equipment and industrial products - Structuring principles and reference designations - Part 2: Classification of objects and codes for classes".
- [IEEE1028] IEEE SA 1028, 2008, "IEEE Standard for Software Reviews and Audits".
- [IEEE1061] IEEE Std 1061, 1998, "IEEE Standard for a Software Quality Metrics Methodology".
- [IEEE610] IEEE Std 610, 1990, "IEEE Standard Glossary of Software Engineering Terminology".
- [Ins22] Institute of Automation and Information Systems, "Advanced systems engineering for control software as a prerequisite for flexible, adaptive cyberphysical production systems (advacode)" [Online] Available: <https://www.mec.ed.tum.de/ais/forschung/aktuelle-forschungsprojekte/advacode/>, [Accessed: 28-04-22], 2022.
- [ISA88] ANSI/ISA 88.01, 1995, "Batch Control, Part 1: Models and Terminology".
- [ISO13485] ISO 13485, 2016, "Medical devices - Quality management systems - Requirements for regulatory purposes".
- [ISO25010] ISO/IEC 25010, 2011, "Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models".
- [ISO25023] ISO/IEC 25023, 2016, "Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - Measurement of system and software product quality".
- [ISO42010] ISO/IEC/IEEE 42010, 2011, "Systems and Software Engineering – Architecture Description".
- [Jab⁺15] Jabangwe, R., Börstler, J., Šmite, D. and Wohlin, C., "Empirical evidence on the link between object-oriented measures and external quality attributes: a systematic literature review," In: *Empirical Software Engineering*, vol. 20, no. 3, pp. 640–693, 2015.
- [Jet⁺13a] Jetley, R., Rath, A., Aparajithan, V., Kumar, D., Prasad, V. and Ramaswamy, S., "An approach for comparison of IEC 61131-3 graphical programs," In: *IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA)*: IEEE, pp. 1–8, 2013.

- [Jet⁺13b] Jetley, R., Nair, A., Chandrasekaran, P. and Dubey, A., "Applying software engineering practices for development of industrial automation applications," In: *11th IEEE International Conference on Industrial Informatics (INDIN)*: IEEE, pp. 558–563, 2013.
- [Jet22] JetBrains s.r.o., "ReSharper - The Visual Studio Extension for.NET Developers" [Online] Available: <https://www.jetbrains.com/resharper/>, [Accessed: 01-05-22], 2022.
- [Jna⁺20] Jnanamurthy, H. K., Jetley, R., Henskens, F., Paul, D., Wallis, M. and Sudarsan, S. D., "Multi-level analysis of IEC 61131-3 languages to detect clones," In: *International Journal of Computer Applications in Technology (IJCAT)*, vol. 63, no. 4, pp. 286, 2020.
- [Joh77] Johnson, S. C., "Lint, a C Program Checker. techn. report 65," Published by: Bell Laboratories, 1977.
- [Jul⁺17] Julius, R., Schürenberg, M., Schumacher, F. and Fay, A., "Transformation of GRAFCET to PLC code including hierarchical structures," In: *Control Engineering Practice*, vol. 64, pp. 173–194, 2017.
- [JYL17] Jung, S., Yoo, J. and Lee, Y.-J., "A PLC platform-independent structural analysis on FBD programs for digital reactor protection systems," In: *Annals of Nuclear Energy*, vol. 103, pp. 454–469, 2017.
- [KfV04] Katzke, U., Fischer, K. and Vogel-Heuser, B., "Development and Evaluation of a Model for Modular Automation in Plant Manufacturing," In: *10th International Conference on Information Systems Analysis and Synthesis (CITSA)*, pp. 15–20, 2004.
- [Kir⁺16] Kirchmayr, W., Moser, M., Nocke, L., Pichler, J. and Tober, R., "Integration of Static and Dynamic Code Analysis for Understanding Legacy Source Code," In: *IEEE International Conference on Software Maintenance and Evolution (ICSME)*: IEEE, pp. 543–552, 2016.
- [KJS16] Kumar, L., Jetley, R. and Sureka, A., "Source code metrics for programmable logic controller (PLC) ladder diagram (LD) visual programming language," In: *IEEE/ACM 7th International Workshop on Emerging Trends in Software Metrics (WETSoM)*, pp. 15–21, 2016.
- [KKV18] Koltun, G., Kolter, M. and Vogel-Heuser, B., "Automated Generation of Modular PLC Control Software from P&ID Diagrams in Process Industry," In: *IEEE International Systems Engineering Symposium (ISSE)*: IEEE, pp. 1–8, 2018.
- [Koz⁺20] Koziolok, H., Burger, A., Platenius-Mohr, M. and Jetley, R., "A classification framework for automated control code generation in industrial automation," In: *Journal of Systems and Software (JSS)*, vol. 166, pp. 1–23, 2020.

- [KP14] Klammer, C. and Pichler, J., "Towards tool support for analyzing legacy systems in technical domains," In: *IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*: IEEE, pp. 371–374, 2014.
- [KS13] Kaur, N. and Singh, A., "A Complexity Metric for Black Box Components," In: *International Journal of Soft Computing and Engineering (IJSCE)*, vol. 3, no. 2, 179-184, 2013.
- [KS17] Kumar, L. and Sureka, A., "Using Structured Text Source Code Metrics and Artificial Neural Networks to Predict Change Proneness at Code Tab and Program Organization Level," In: *10th Innovations in Software Engineering Conference (ISEC)*. New York, NY, USA: ACM Press, pp. 172–180, 2017.
- [KS19] Kazala, R. and Straczynski, P., "The Most Important Open Technologies for Design of Cost Efficient Automation Systems," In: *IFAC-PapersOnLine*, vol. 52, no. 25, pp. 391–396, 2019.
- [LÅ07] Ljungkrantz, O. and Åkesson, K., "A Study of Industrial Logic Control Programming using Library Components," In: *IEEE International Conference on Automation Science and Engineering (CASE)*: IEEE, pp. 117–122, 2007.
- [Lad⁺13] Ladiges, J., Fay, A., Haubeck, C. and Lamersdorf, W., "Operationalized definitions of non-functional requirements on automated production facilities to measure evolution effects with an automation system," In: *IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA)*: IEEE, pp. 1–6, 2013.
- [Lad⁺18] Ladiges, J., Fay, A., Holm, T., Hempen, U., Urbas, L., Obst, M. and Albers, T., "Integration of Modular Process Units Into Process Control Systems," In: *IEEE Transactions on Industry Applications*, vol. 54, no. 2, pp. 1870–1880, 2018.
- [LC94] Lake, A. and Cook, C. R., "Use of Factor Analysis to Develop OOP Software Complexity Metrics," Published by: Oregon State University, 1994.
- [LG99] Lauber, R. and Göhner, P., "Process Automation 1 [Prozessautomatisierung 1]." Berlin, Heidelberg: Springer Berlin Heidelberg, 1999.
- [LH01] Lee, J.-S. and Hsu, P.-L., "A new approach to evaluate ladder logic diagrams and Petri nets via the IF-THEN transformation," In: *IEEE International Conference on Systems, Man & Cybernetics*: IEEE, pp. 2711–2716, 2001.
- [Lie22] Lieberoth-Leden, C., "Steuerungskonzept für die Berücksichtigung von gegenseitigen Abhängigkeiten zwischen Transporten in adaptiven automatisierten Materialflusssystemen," Dissertation, Technical University of Munich. Materials Handling, Material Flow, Logistics (fml), 2022.
- [log22a] logi.cals GmbH, "Validating an application" [Online] Available: <https://help.logi-cals.com/lco3docu/latest/user-documentation/en/referenzdokumentation/anwendung-validieren>, [Accessed: 09-05-22], 2022.

- [log22b] logi.cals GmbH, "Rules for the validation of an application" [Online] Available: <https://help.logicals.com/lco3docu/latest/user-documentation/en/referenzdokumentation/anwendung-validieren/regeln-fuer-das-validieren-einer-anwendung>, [Accessed: 09-05-22], 2022.
- [Lop⁺21] Lopez-Miguel, I. D., Adiego, B. F., Tournier, J.-C., Viñuela, E. B. and Rodriguez-Aguilar, J. A., "Simplification of numeric variables for PLC model checking," In: S. Arun-Kumar, Dominique Mery, Indranil Saha and Lijun Zhang (Eds.): *Proceedings of the 19th ACM-IEEE International Conference on Formal Methods and Models for System Design*. New York, NY, USA: ACM, pp. 10–20, 2021.
- [Lou06] Louridas, P., "Static code analysis," In: *IEEE Softw.*, vol. 23, no. 4, pp. 58–61, 2006.
- [LT03] Lucas, M. R. and Tilbury, D. M., "A study of current logic design practices in the automotive manufacturing industry," In: *International Journal of Human-Computer Studies*, vol. 59, no. 5, pp. 725–753, 2003.
- [LT05] Lucas, M. R. and Tilbury, D. M., "Methods of measuring the size and complexity of PLC programs in different logic control design methodologies," In: *The International Journal of Advanced Manufacturing Technology (JAMT)*, vol. 26, no. 5-6, pp. 436–447, 2005.
- [LVD06] LaToza, T. D., Venolia, G. and DeLine, R., "Maintaining mental models," In: Leon J. Osterweil, Dieter Rombach and Mary Lou Soffa (Eds.): *Proceeding of the 28th International Conference on Software Engineering - ICSE '06*. New York, New York, USA: ACM Press, pp. 492–501, 2006.
- [Lyt⁺20] Lytra, I., Carrillo, C., Capilla, R. and Zdun, U., "Quality attributes use in architecture design decision methods: research and practice," In: *Computing*, vol. 102, no. 2, pp. 551–572, 2020.
- [Mah14] Mahler, C., "Automatisierungsmodule für ein funktionsorientiertes Automatisierungsengineering," Dissertation, Helmut-Schmidt-Universität, Hamburg. Institut für Automatisierungstechnik, 2014.
- [Man⁺18] Mandal, A., Mohan, D., Jetley, R., Nair, S. and D'Souza, M., "A Generic Static Analysis Framework for Domain-specific Languages," In: *IEEE 23rd International Conference on Emerging Technologies & Factory Automation (ETFA)*: IEEE, pp. 27–34, 2018.
- [Math+22] MathWorks, Inc., "Simulink PLC Coder" [Online] Available: <https://www.mathworks.com/products/simulink-plc-coder.html>, [Accessed: 04-04-22].
- [McC76] McCabe, T. J., "A Complexity Measure," In: *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.
- [ME20] Muslija, A. and Enoiu, E. P., "On the Measurement of Software Complexity for PLC Industrial Control Systems using TIQVA," In: *35th ACM/SIGAPP Symposium on Applied Computing*, pp. 1556–1565, 2020.

- [Mey97] Meyer, B., "Object-Oriented Software Construction. Second Edition." New York: Prentice hall, 1997.
- [MGB22] Mejia, A., Guarnizo, A. F. and Barbieri, G., "Assessment of the PLC Code generated with the GEMMA-GRAF CET Methodology," In: *Procedia Computer Science*, vol. 200, pp. 699–709, 2022.
- [MJG11a] Maga, C. R., Jazdi, N. and Göhner, P., "Requirements on engineering tools for increasing reuse in industrial automation," In: *IEEE Conference on Emerging Technologies & Factory Automation (ETFA)*: IEEE, pp. 1–7, 2011.
- [MJG11b] Maga, C. R., Jazdi, N. and Göhner, P., "Reusable Models in Industrial Automation: Experiences in Defining Appropriate Levels of Granularity," In: *IFAC Proceedings Volumes*, vol. 44, no. 1, pp. 9145–9150, 2011.
- [MML15] Minelli, R., Mocci, A. and Lanza, M., "I Know What You Did Last Summer - An Investigation of How Developers Spend Their Time," In: *IEEE 23rd International Conference on Program Comprehension*: IEEE, pp. 25–35, 2015.
- [MT00] Medvidovic, N. and Taylor, R. N., "A classification and comparison framework for software architecture description languages," In: *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pp. 70–93, 2000.
- [Nai⁺15] Nair, S., Jetley, R., Nair, A. and Hauck-Stattelmann, S., "A static code analysis tool for control system software," In: *IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*: IEEE, pp. 459–463, 2015.
- [Nai12] Nair, A., "Product metrics for IEC 61131-3 languages," In: *IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA)*: IEEE, pp. 1–8, 2012.
- [NDG05] Nierstrasz, O., Ducasse, S. and Girba, T., "The Story of Moose: an Agilke Reengineering Environment," In: *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 1–10, 2005.
- [Neu⁺20a] Neumann, E.-M., Vogel-Heuser, B., Fischer, J., Keller, J., Weis, I., Diehm, S., Schwarz, M., Englert, T., Stoll, M. and Zell, U., "Identifying Runtime Issues in Object-Oriented IEC 61131-3-Compliant Control Software using Metrics," In: *The 46th Annual Conference of the IEEE Industrial Electronics Society (IECON)*: IEEE, pp. 259–266, 2020.
- [Neu⁺20b] Neumann, E.-M., Fischer, J., Schneider, H., Vogel-Heuser, B. and Bengel, M., "Metric-based determination of maturity of IEC 61131-3-compliant control software (Metrikbasierte Reifebestimmung von IEC 61131-3 konformer Steuerungssoftware)," In: *VDI Berichte AUTOMATION 2020*, pp. 417–428, 2020.
- [Neu⁺20c] Neumann, E.-M., Vogel-Heuser, B., Fischer, J., Ocker, F., Diehm, S. and Schwarz, M., "Formalization of Design Patterns and Their Automatic Identification in PLC Software for Architecture Assessment," In: *IFAC World Congress*, 7819-7826, 2020.

- [Neu⁺22] Neumann, E.-M., Vogel-Heuser, B., Fischer, J., Diehm, S., Schwarz, M. and Englert, T., "Automation software architectures in automated production systems: an industrial case study in the packaging machine industry," In: *Production Engineering (PERE)*, vol. 16, pp. 847–856, 2022.
- [Nie12] Nierstrasz, O., "Agile software assessment with Moose," In: *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 3, pp. 1–5, 2012.
- [NJ16] Nair, S. and Jetley, R., "Solving circular dependencies in industrial automation programs," In: *IEEE International Conference on Industrial Informatics (INDIN)*: IEEE, pp. 397–404, 2016.
- [NNB19] Nachtigall, M., Nguyen Quang Do, L. and Bodden, E., "Explaining Static Analysis - A Perspective," In: *34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*: IEEE, pp. 29–32, 2019.
- [NVO15] Nedved, M., Vrba, P. and Obitko, M., "Tool for visual difference display of programs in IEC 61131-3 ladder diagrams," In: *IEEE International Conference on Industrial Technology (ICIT)*: IEEE, pp. 2994–2999, 2015.
- [Obe⁺21] Oberlehner, M., Sonnleithner, L., Wiesmayr, B. and Zoitl, A., "Catalog of Refactoring Operations for IEC 61499," In: *IEEE 26th International Conference on Emerging Technologies & Factory Automation (ETF A)*: IEEE, pp. 1–4, 2021.
- [Obs21] Obster, M., "Unterstützung der SPS-Programmierung durch statische Analyse während der Programmeingabe," Dissertation, RWTH Aachen University, 2021.
- [OK17] Obster, M. and Kowalewski, S., "A live static code analysis architecture for PLC software," In: *22nd IEEE International Conference on Emerging Technologies & Factory Automation (ETF A)*: IEEE, pp. 1–4, 2017.
- [OSCAT] Tobias Mühlbauer, "Open Source Community for Automation Technology (OS-CAT)" [Online] Available: <http://www.oscat.de/>, [Accessed: 10-03-22].
- [Ove20] OverOps, "2020 Report: The State of Software Quality," Published by: OverOps, 2020.
- [PackML] Organization for Machine Automation and Control (OMAC), 2016, "PackML Unit / Machine Implementation Guide Part 1: PackML Interface State Manager".
- [PBv05] Pohl, K., Böckle, G. and van der Linden, F., "Software Product Line Engineering." Berlin, Heidelberg: Springer Berlin Heidelberg, 2005.
- [PKK12] Preschern, C., Kajtazovic, N. and Kreiner, C., "Applying patterns to model-driven development of automation systems," In: Andreas Fiesser and Christian Kohls (Eds.): *Proceedings of the 17th European Conference on Pattern Languages of Programs - EuroPLoP '12*. New York, New York, USA: ACM Press, pp. 1–10, 2012.

- [PKS18] Pavlovskiy, Y., Kennel, M. and Schmucker, U., "Template-Based Generation of PLC Software from Plant Models Using Graph Representation," In: *25th International Conference on Mechatronics and Machine Vision in Practice (M2VIP)*: IEEE, pp. 1–8, 2018.
- [PLC16] PLCopen Software Construction Guidelines - Task Force Coding Guidelines, "Coding Guidelines V.1.0," Published by: PLCopen, 2016.
- [PLC21] PLCopen Promotional Committee 2 – Training, "Guidelines for usage of Object-Oriented Programming. Version 1.0," Published by: PLCopen, 2021.
- [PLC22] PLCopen, "Downloads - Guidelines Category" [Online] Available: https://plcopen.org/downloads?field_category_target_id=164, [Accessed: 05-06-22], 2022.
- [Plö+08] Plösch, R., Gruber, H., Hentschel, A., Körner, C., Pomberger, G., Schiffer, S., Saft, M. and Storck, S., "The EMISQ method and its tool support-expert-based evaluation of internal software quality," In: *Innovations in Systems and Software Engineering*, vol. 4, no. 1, pp. 3–15, 2008.
- [Plö+10] Plösch, R., Gruber, H., Körner, C. and Saft, M., "A Method for Continuous Code Quality Management Using Static Analysis," In: *7th IEEE International Conference on the Quality of Information and Communications Technology*: IEEE, pp. 370–375, 2010.
- [Prä+12] Prähofer, H., Angerer, F., Ramler, R., Lacheiner, H. and Grillenberger, F., "Opportunities and challenges of static code analysis of IEC 61131-3 programs," In: *IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA)*: IEEE, pp. 1–8, 2012.
- [Prä+17] Prähofer, H., Angerer, F., Ramler, R. and Grillenberger, F., "Static Code Analysis of IEC 61131-3 Programs: Comprehensive Tool Support and Experiences from Large-Scale Industrial Application," In: *IEEE Transactions on Industrial Informatics (TII)*, vol. 13, no. 1, pp. 37–47, 2017.
- [Pri+16] Priego, R., Armentia, A., Estévez, E. and Marcos, M., "Modeling techniques as applied to generating tool-independent automation projects," In: *at – Automatisierungstechnik*, vol. 64, no. 4, 2016.
- [PS13] Papakonstantinou, N. and Sierla, S., "Generating an Object Oriented IEC 61131-3 software product line architecture from SysML," In: *IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA)*: IEEE, pp. 1–8, 2013.
- [Pun+22] Puntel Schmidt, P., Fischer, J., Neumann, E.-M., Vogel-Heuser, B., Witte, M., Bengel, M. and Felger, M., "Kriterien für die Strukturanalyse von Steuerungscode: SPS-Software-Architekturen verstehen und interpretieren," In: *atp*, vol. 64, no. 3, pp. 60–69, 2022.
- [Rab+18] Rabiser, D., Prähofer, H., Grünbacher, P., Petruzella, M., Eder, K., Angerer, F., Kromoser, M. and Grimmer, A., "Multi-purpose, multi-level feature modeling of

- large-scale industrial software systems," In: *Software & Systems Modeling*, vol. 17, no. 3, pp. 913–938, 2018.
- [Ram⁺19] Ramler, R., Buchgeher, G., Klammer, C., Pfeiffer, M., Salomon, C., Thaller, H. and Linsbauer, L., "Benefits and Drawbacks of Representing and Analyzing Source Code and Software Engineering Artifacts with Graph Databases," In: Dietmar Winkler, Stefan Biffl and Johannes Bergsmann (Eds.): *Software Quality: The Complexity and Challenges of Software Engineering and Software Quality in the Cloud*, Bd. 338. Cham: Springer International Publishing (Lecture Notes in Business Information Processing), pp. 125–148, 2019.
- [Ram⁺85] Ramamoorthy, C. V., Tsai, W. T., Yamaura, T. and Bhide, A., "Metrics Guided Methodology," In: *IEEE International Computer Software and Applications Conference (COMPSAC)*: IEEE, pp. 111–120, 1985.
- [RBS13] Rattan, D., Bhatia, R. and Singh, M., "Software clone detection: A systematic review," In: *Information and Software Technology*, vol. 55, no. 7, pp. 1165–1199, 2013.
- [Reu⁺19] Reussner, R., Goedicke, M., Hasselbring, W., Vogel-Heuser, B., Keim, J. and Martin, L., "Managed Software Evolution." Cham: Springer International Publishing, 2019.
- [Ros⁺21a] Rosiak, K., Schlie, A., Linsbauer, L., Vogel-Heuser, B. and Schaefer, I., "Custom-tailored clone detection for IEC 61131-3 programming languages," In: *Journal of Systems and Software (JSS)*, vol. 182, pp. 1–18, 2021.
- [Ros21b] Rosiak, K., "TUBS-ISF / IEC_61131_3_Clone_Detection" [Online] Available: https://github.com/TUBS-ISF/IEC_61131_3_Clone_Detection, [Accessed: 05-06-22], 2021.
- [Ros97] Rosenberg, J., "Some misconceptions about lines of code," In: *Fourth International Software Metrics Symposium*: IEEE Computer Society, pp. 137–142, 1997.
- [Ryd79] Ryder, B. G., "Constructing the Call Graph of a Program," In: *IEEE Transactions on Software Engineering*, vol. SE-5, no. 3, pp. 216–226, 1979.
- [Sam⁺13] Samarthyam, G., Suryanarayana, G., Sharma, T. and Gupta, S., "MIDAS: A design quality assessment method for industrial software," In: *35th International Conference on Software Engineering (ICSE)*: IEEE, pp. 911–920, 2013.
- [SC21] Sarkar, S. and Chandrika, K. R., "Automatic Control Code Generation from SAMA Specification," In: *IEEE 26th International Conference on Emerging Technologies & Factory Automation (ETFA)*: IEEE, pp. 1–4, 2021.
- [Sch⁺19] Schlie, A., Rosiak, K., Urbaniak, O., Schaefer, I. and Vogel-Heuser, B., "Analyzing variability in automation software with the variability analysis toolkit," In: Camille Salinesi and Tewfik Ziadi (Eds.): *Proceedings of the 23rd International Systems and Software Product Line Conference volume B - SPLC '19*. New York, New York, USA: ACM Press, pp. 1–8, 2019.

- [Scrum22] ScrumGuides.org, "Scrum Guides" [Online] Available: <https://scrumguides.org/>, [Accessed: 20-05-22].
- [SE21] Schneider Electric, "EcoStruxure Control Engineering - Documentation (User Manual)" [Online] Available: https://www.se.com/ww/en/download/document/EIO0000004426.01_Documentation/, [Accessed: 10-05-22], 2021.
- [SE22a] Schneider Electric, "EcoStruxure Machine Expert - Machine Code Analysis (User Guide)" [Online] Available: <https://www.se.com/us/en/download/document/EIO0000002710/>, [Accessed: 10-05-22], 2022.
- [SE22b] Schneider Electric, "EcoStruxure Control Engineering" [Online] Available: <https://www.se.com/ww/en/product-range/39982702-ecostruxure-control-engineering/>, [Accessed: 10-05-22], 2022.
- [Seh⁺21] Sehr, M. A., Lohstroh, M., Weber, M., Ugalde, I., Witte, M., Neidig, J., Hoeme, S., Niknami, M. and Lee, E. A., "Programmable Logic Controllers in the Context of Industry 4.0," In: *IEEE Transactions on Industrial Informatics (TII)*, vol. 17, no. 5, pp. 3523–3533, 2021.
- [SF14] Schumacher, F. and Fay, A., "Formal representation of GRAFCET to automatically generate control code," In: *Control Engineering Practice*, vol. 33, pp. 84–93, 2014.
- [SFJ15] Schröck, S., Fay, A. and Jäger, T., "Systematic interdisciplinary reuse within the engineering of automated plants," In: *9th Annual IEEE International Systems Conference (SysCon)*. Piscataway, NJ: IEEE, pp. 508–515, 2015.
- [Sie06] Siemens AG, "SIMATIC Programming with STEP 7. Manual," Published by: Siemens AG, 2006.
- [Sie15] Siemens AG, "Standards compliance according to IEC 61131-3 (3rd Edition)" [Online] Available: https://cache.industry.siemens.com/dl/files/748/109476748/att_845621/v1/IEC_61131_compliance_en_US.pdf, [Accessed: 11-06-22], 2015.
- [Sie18] Siemens AG, "Programming Guideline for S7-1200/1500" [Online] Available: https://cache.industry.siemens.com/dl/files/040/90885040/att_970576/v1/81318674_Programming_guideline_DOC_v16_en.pdf, [Accessed: 18-04-22], 2018.
- [Sie20a] Siemens AG, "Project Check for TIA Portal" [Online] Available: <https://support.industry.siemens.com/cs/document/109741418/project-check-for-tia-portal-check-against-programming-style-guides?dti=0&lc=en-WW>, [Accessed: 18-05-22], 2020.
- [Sie20b] Siemens AG, "TIA Portal Test Suite (Function Manual)" [Online] Available: https://cache.industry.siemens.com/dl/files/356/109779356/att_1019655/v1/TestSuiteOLH_enUS_en-US.pdf, [Accessed: 10-05-22], 2020.

- [Sie22] Siemens AG, "TIA Portal Openness: Introduction and Demo Application" [Online] Available: <https://support.industry.siemens.com/cs/document/108716692/tia-portal-openness-introduction-and-demo-application?dti=0&lc=en-DE>, [Accessed: 22-04-22].
- [SK16] Simon, H. and Kowalewski, S., "Static analysis of Sequential Function Charts using abstract interpretation," In: *IEEE 21st International Conference on Emerging Technologies & Factory Automation (ETFA)*: IEEE, pp. 1–4, 2016.
- [Son+21a] Sonnleithner, L., Oberlehner, M., Kutsia, E., Zoitl, A. and Bacsı, S., "Do you smell it too? Towards Bad Smells in IEC 61499 Applications," In: *IEEE 26th International Conference on Emerging Technologies & Factory Automation (ETFA)*: IEEE, pp. 1–4, 2021.
- [Son+21b] Sonnleithner, L., Wiesmayr, B., Ashiwal, V. and Zoitl, A., "IEC 61499 Distributed Design Patterns," In: *IEEE 26th International Conference on Emerging Technologies & Factory Automation (ETFA)*: IEEE, pp. 1–8, 2021.
- [Son22] SonarSource S.A, "SonarQube" [Online] Available: <https://www.sonarqube.org/>, [Accessed: 05-06-22], 2022.
- [Spi+17] Spindler, M., Aicher, T., Vogel-Heuser, B. and Fottner, J., "Engineering the Control Software of Automated Material Handling Systems via Drag & Drop [Erstellung von Steuerungssoftware für automatisierte Materialflusssysteme per Drag & Drop]," In: *Logistics Journal*, pp. 1–8, 2017.
- [SS16] Silva, B. G. and Sousa, M. de, "Internal inconsistencies in the third edition of the IEC 61131-3 international standard," In: *IEEE 21st International Conference on Emerging Technologies & Factory Automation (ETFA)*: IEEE, pp. 1–4, 2016.
- [Sta+14] Stattelmann, S., Biallas, S., Schlich, B. and Kowalewski, S., "Applying static code analysis on industrial controller code," In: *IEEE Emerging Technology & Factory Automation (ETFA)*: IEEE, pp. 1–4, 2014.
- [Ste00] Stewart, M., "An experiment in scientific program understanding," In: *Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering*: IEEE, pp. 281–284, 2000.
- [SZ12] Steinegger, M. and Zoitl, A., "Automated code generation for programmable logic controllers based on knowledge acquisition from engineering artifacts: Concept and case study," In: *IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA)*: IEEE, pp. 1–8, 2012.
- [SZ20] Sonnleithner, L. and Zoitl, A., "A Software Measure for IEC 61499 Basic Function Blocks," In: *25th IEEE International Conference on Emerging Technologies & Factory Automation (ETFA)*: IEEE, pp. 997–1000, 2020.
- [TB10] Thramboulidis, K. and Buda, A., "3+1 SysML view model for IEC61499 Function Block control systems," In: *8th IEEE International Conference on Industrial Informatics (INDIN)*: IEEE, pp. 175–180, 2010.

- [TBF17] Tsiplaki Spiliopoulou, C., Blanco Viñuela, E. and Fernández Adiego, B., "Experience With Static PLC Code Analysis at CERN," In: *16th International Conference on Accelerator and Large Experimental Control Systems (ICALEPCS)*. JACoW, Geneva, Switzerland, pp. 1787–1791, 2017.
- [Tha⁺17] Thaller, H., Ramler, R., Pichler, J. and Egyed, A., "Exploring code clones in programmable logic controller software," In: *22nd IEEE International Conference on Emerging Technologies & Factory Automation (ETFA)*: IEEE, pp. 1–8, 2017.
- [Thr10] Thramboulidis, K., "The 3+1 SysML View-Model in Model Integrated Mechatronics," In: *Journal of Software Engineering and Applications (JSEA)*, vol. 03, no. 02, pp. 109–118, 2010.
- [Thr13] Thramboulidis, K., "IEC 61499 as an Enabler of Distributed and Intelligent Automation: A State-of-the-Art Review—A Different View," In: *Journal of Engineering*, vol. 2013, no. 4, pp. 1–9, 2013.
- [Thü⁺14] Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G. and Leich, T., "FeatureIDE: An extensible framework for feature-oriented software development," In: *Science of Computer Programming*, vol. 79, pp. 70–85, 2014.
- [VB21] Vogel-Heuser, B. and Bi, F., "Interdisciplinary effects of technical debt in companies with mechatronic products — a qualitative study," In: *Journal of Systems and Software (JSS)*, vol. 171, pp. 1–17, 2021.
- [VDI2206] VDI/VDE 2206, November 2021, "Development of mechatronic and cyber-physical systems".
- [VDI2206-04] VDI/VDE 2206, June 2004, "Design methodology for mechatronic systems".
- [VDI5100] VDI/VDMA 5100, 2016, "System Architecture For Intralogistics (SAIL)".
- [VFB03] Vogel-Heuser, B., Friedrich, D. and Bristol, E. R., "Evaluation of modeling notations for basic software engineering in process control," In: *29th Annual Conference of the IEEE Industrial Electronics Society (IECON'03, IEEE Cat. No.03CH37468)*: IEEE, pp. 2209–2214, 2003.
- [VFN20] Vogel-Heuser, B., Fischer, J. and Neumann, E.-M., "Goal-Lever-Indicator-Principle to Derive Recommendations for Improving IEC 61131-3 Control Software," In: *IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*: IEEE, pp. 1131–1136, 2020.
- [VNF22] Vogel-Heuser, B., Neumann, E.-M. and Fischer, J., "MICOSE4aPS: Industrially Applicable Maturity Metric to Improve Systematic Reuse of Control Software," In: *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 1, pp. 1–24, 2022.
- [VO18] Vogel-Heuser, B. and Ocker, F., "Maintainability and evolvability of control software in machine and plant manufacturing — An industrial survey," In: *Control Engineering Practice*, vol. 80, pp. 157–173, 2018.

- [Vog⁺14] Vogel-Heuser, B., Legat, C., Folmer, J. and Feldmann, S., "Researching Evolution in Industrial Plant Automation: Scenarios and Documentation of the Pick and Place Unit," AIS Institute, Technical University of Munich, 2014.
- [Vog⁺15a] Vogel-Heuser, B., Fischer, J., Rösch, S., Feldmann, S. and Ulewicz, S., "Challenges for maintenance of PLC-software and its related hardware for automated production systems: Selected industrial Case Studies," In: *IEEE International Conference on Software Maintenance and Evolution (ICSME)*: IEEE, pp. 362–371, 2015.
- [Vog⁺15b] Vogel-Heuser, B., Fay, A., Schaefer, I. and Tichy, M., "Evolution of software in automated production systems: Challenges and research directions," In: *Journal of Systems and Software (JSS)*, vol. 110, pp. 54–84, 2015.
- [Vog⁺16] Vogel-Heuser, B., Rösch, S., Fischer, J., Simon, T., Ulewicz, S. and Folmer, J., "Fault Handling in PLC-Based Industry 4.0 Automated Production Systems as a Basis for Restart and Self-Configuration and Its Evaluation," In: *Journal of Software Engineering and Applications (JSEA)*, vol. 09, no. 01, pp. 1–43, 2016.
- [Vog⁺17] Vogel-Heuser, B., Fischer, J., Feldmann, S., Ulewicz, S. and Rösch, S., "Modularity and architecture of PLC-based software for automated production Systems: An analysis in industrial companies," In: *Journal of Systems and Software (JSS)*, vol. 131, pp. 35–62, 2017.
- [Vog⁺18] Vogel-Heuser, B., Fischer, J., Neumann, E.-M. and Diehm, S., "Key maturity indicators for module libraries for PLC-based control software in the domain of automated Production Systems," In: *IFAC-PapersOnLine*, vol. 51, no. 11, pp. 1610–1617, 2018.
- [Vog⁺21a] Vogel-Heuser, B., Fischer, J., Neumann, E.-M. and Kreiner, M., "Success Factors for the Design of Field-level Control Code in Machine and Plant Manufacturing - an Industrial Survey," In: *Research Square - preprint*, 2021.
- [Vog⁺21b] Vogel-Heuser, B., Huber, C., Cha, S. and Beckert, B., "Integration of a formal specification approach into CPPS engineering workflow for machinery validation," In: *IEEE International Conference on Industrial Informatics (INDIN)*: IEEE, 2021.
- [Vog⁺21c] Vogel-Heuser, B., Neumann, E.-M., Zoitl, A., Fernandez, A. M. G., Rabiser, R. and Fadhlillah, H. S., "An International Case Study on Control Software Development in Large-Scale Plant Manufacturing Companies of One Industrial Sector at Different Locations," In: *47th Annual Conference of the IEEE Industrial Electronics Society (IECON)*: IEEE, pp. 1–8, 2021.
- [Vog⁺22a] Vogel-Heuser, B., Fischer, J., Hess, D., Neumann, E.-M. and Würri, M., "Boosting Extra-Functional Code Reusability in Cyber-Physical Production Systems: The Error Handling Case Study," In: *IEEE Transactions on Emerging Topics in Computing (TETC)*, vol. 10, no. 1, pp. 60–73, 2022.
- [Vog⁺22b] Vogel-Heuser, B., Neumann, E.-M., Fischer, J., Marcos, M., Estévez Estévez, E., Barbieri, G., Sonnleithner, L. and Rabiser, R., "Automation Software Architecture

- in CPPS - Definition, Challenges and Research Potentials," In: *5th IEEE International Conference on Industrial Cyber-Physical Systems (ICPS)*: IEEE, pp. 1–8, 2022.
- [VON18] Vogel-Heuser, B., Ocker, F. and Neumann, E.-M., "Maturity variations of PLC-based control software within a company and among companies from the same industrial sector," In: *IEEE Industrial Cyber-Physical Systems (ICPS)*: IEEE, pp. 283–290, 2018.
- [VSF16] Vogel-Heuser, B., Simon, T. and Fischer, J., "Variability management for automated production systems using product lines and feature models," In: *IEEE International Conference on Industrial Informatics (INDIN)*: IEEE, pp. 1231–1237, 2016.
- [Vya11] Vyatkin, V., "IEC 61499 as Enabler of Distributed and Intelligent Automation: State-of-the-Art Review," In: *IEEE Transactions on Industrial Informatics (TII)*, vol. 7, no. 4, pp. 768–781, 2011.
- [Vya13] Vyatkin, V., "Software Engineering in Industrial Automation: State-of-the-Art Review," In: *IEEE Transactions on Industrial Informatics (TII)*, vol. 9, no. 3, pp. 1234–1249, 2013.
- [Wer09] Werner, B., "Object-oriented extensions for IEC 61131-3," In: *IEEE Industrial Electronics Magazine*, vol. 3, no. 4, pp. 36–39, 2009.
- [Wey88] Weyuker, E. J., "Evaluating software complexity measures," In: *IEEE Transactions on Software Engineering*, vol. 14, no. 9, pp. 1357–1365, 1988.
- [Wil⁺19] Wilch, J., Fischer, J., Neumann, E.-M., Diehm, S., Schwarz, M., Lah, E., Wander, M. and Vogel-Heuser, B., "Introduction and Evaluation of Complexity Metrics for Network-based, Graphical IEC 61131-3 Programming Languages," In: *45th Annual Conference of the IEEE Industrial Electronics Society (IECON)*: IEEE, 2019.
- [Wil⁺22] Wilch, J., Fischer, J., Langer, N., Felger, M., Bengel, M. and Vogel-Heuser, B., "Towards automatic generation of functionality semantics to improve PLC software modularization," In: *at-Automatisierungstechnik*, vol. 70, no. 2, pp. 181–191, 2022.
- [Wim⁺17] Wimmer, M., Novak, P., Sindelar, R., Berardinelli, L., Mayerhofer, T. and Mazak, A., "Cardinality-based variability modeling with AutomationML," In: *22nd IEEE International Conference on Emerging Technologies & Factory Automation*. Piscataway, NJ: IEEE, 2017.
- [WSZ20] Wiesmayr, B., Sonnleithner, L. and Zoitl, A., "Structuring Distributed Control Applications for Adaptability," In: *IEEE Conference on Industrial Cyberphysical Systems (ICPS)*: IEEE, pp. 236–241, 2020.
- [Wu⁺20] Wu, Q., Gouyon, D., Levrat, E. and Boudau, S., "Use of Patterns for Know-How Reuse in a Model-Based Systems Engineering Framework," In: *IEEE Systems Journal*, vol. 14, no. 4, pp. 4765–4776, 2020.

-
- [YF07] Younis, M. B. and Frey, G., "Software quality measures to determine the diagnosability of PLC applications," In: *IEEE Conference on Emerging Technologies & Factory Automation (ETFA)*: IEEE, pp. 368–375, 2007.
- [ZP08] Zaeh, M. F. and Poernbacher, C., "Model-driven development of PLC software for machine tools," In: *Production Engineering*, vol. 2, no. 1, pp. 39–46, 2008.
- [ZV15] Zhabelova, G. and Vyatkin, V., "Towards software metrics for evaluating quality of IEC 61499 automation software," In: *IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA)*: IEEE, pp. 1–8, 2015.

11. List of Figures

Figure 1:	Schematic structure of process automation systems (translated from [LG99], p. 7)..	6
Figure 2:	Overview of the research areas regarding means and approaches for PLC software quality assessment, including selected approaches from computer science. The identified research gap is highlighted in dark grey.....	43
Figure 3:	Overall complexity of POU's from an industrial PLC project, annotated according to their functionality (implemented in LD, PLC project taken from Case Study C (cf. Table 2, p. 17); graphically adapted from [Fis ⁺ 21b]).	46
Figure 4:	Software assessment procedure for classical IEC 61131-3 control software.....	50
Figure 5:	Details of Step 1 (Preparation and Familiarization) of the quality assessment procedure.	51
Figure 6:	Exemplary layout of a warehouse (graphically adapted from [FVF15]).	52
Figure 7:	Details of Step 2 (Static code analysis of a single PLC project) of the quality assessment procedure.	60
Figure 8:	Exemplary software development workflow of an aPS manufacturer modeled using BPMN.....	62
Figure 9:	Annotated call graph of the analyzed software project of Case Study A, with entry point OB1.	63
Figure 10:	Call hierarchy (left, adapted from [Vog ⁺ 17]) and communication of errors (right, adapted from [Vog ⁺ 15a]) in regard to the identified architectural hierarchy levels in Case Study A.	69
Figure 11:	Details of Step 3 (Comparison of additional PLC projects or PLC project parts) of the quality assessment procedure.	71
Figure 12:	Call graphs of two warehouse software variants from Case Study A differing in the mechanical variation point “amount of satellite cars” (124 POU's each).....	75
Figure 13:	Different means of visualization and documentation of analysis results for different stakeholders such as managers (👤), application developers (👩‍💻), and module developers (👨‍💻); summarized from [Bou ⁺ 19; Fis ⁺ 20a; Fis ⁺ 21b; Neu ⁺ 20b; VNF22; Vog ⁺ 16; Wil ⁺ 19; Wil ⁺ 22].....	79

Figure 14: Details of Step 4 (Quality assessment and recommendations) of the assessment procedure.	80
Figure 15: Goal-lever-indicator-principle at the example of planned reuse (adopted from [VFN20]).	84
Figure 16: Screenshot of the data exchange view in the advacode prototype, including available, pre-defined rules (top left) with criticality level, rule editor (bottom left) and the table-based results view (right); content published in [Fis+22b].....	92
Figure 17: Screenshot of a call graph, including color-coding for the implemented main functionality of each POU in the advacode prototype.	94
Figure 18: Screenshot of combined view with call graph (left) and data exchange via data blocks (right); dependencies to selected POU “W4_Wrapper_Scan” are highlighted in both views (content published in [Fis+22b]).	95
Figure 19: Comparison of Case Study B and C linked to the four quality assessment steps.....	99
Figure 20: Considered intralogistics plant part of Case Study B, including interfaces to adjacent plant parts and organization in lanes consisting of conveying elements (CEs); adapted and modified from [Ber19].	103
Figure 21: Software and product development workflow applied in Case Study B at the company site using BPMN.	104
Figure 22: Generated, manually annotated call graph (left) and indirect data exchange graph via flag variables (right) of Case Study B’s control software in Step 2; adopted from [Ber19].	106
Figure 23: Performed steps during the automatic and manual control software analysis of project variants concerning influences of hardware variations (left), including gained insights (right).	108
Figure 24: Comparison of two call graphs regarding prominent features and commonalities in their structure as a pre-step for their detailed manual comparison on the sub-POU level.....	108
Figure 25: Goal-lever-indicator principle applied to the identified weakness WB-1 to derive a recommendation for action (following the style of [VFN20]).	110
Figure 26: Development workflow of Case Study C, divided into module and application development, using BPMN (adapted and extended from [Fis+21a]).	114

Figure 27: Module integration into the template project in the style of [Vog ⁺ 15a], adopted from [Hub20].....	118
Figure 28: Comparison of project versions in Case Study C, focusing on interface changes.	119
Figure 29: Procedure steps for defining customized similarity metrics and the subsequent semi-automatic identification of reusable software parts; details published in [Fis ⁺ 20a; Fis ⁺ 20b; Ros ⁺ 21a].	126
Figure 30: Background of participants regarding application sector (question WG#11, multiple-choice, answers: 20 participants).....	133
Figure 31: Background of participants regarding used programming platforms within their company (question WG#13, multiple-choice, answers: 20 participants).	134
Figure 32: Estimation of the integratability of the quality assessment procedure into the company workflow (question WG#3, single choice, answers: 20 participants).	135
Figure 33: Challenges hindering the procedure application in an industrial context (question WG#4, free-text question, answers: 19 participants).....	135
Figure 34: Answers to W#3 concerning the applicability of the analysis procedure under consideration of company-specific boundary conditions (total answers n = 40)...	140
Figure 35: Answers to W#7 regarding the independent application of the analysis procedure by the participants themselves (total answers n = 37); pseudo accuracy to avoid rounding error.	141
Figure 36: Qualitative assessment of the applicability and usefulness of the procedure for integrating quality-assuring means into the software development process (Group 1 = black (12 replies), Group 2 = grey (6 replies), Group 3 = white (12 replies)), adapted from [Fis ⁺ 22a].....	142
Figure 37: Answers to W#9 regarding the application of static code analysis with and without the proposed procedure (total answers n = 34); pseudo accuracy to avoid rounding error.	144
Figure 38: First page of the industry-WG questionnaire in German.	193
Figure 39: Second page of the industry-WG questionnaire in German.....	194
Figure 40: Third page of the industry-WG questionnaire in German.....	195

12. List of Tables

Table 1:	Selected reuse approaches for control software in the aPS domain.	12
Table 2:	Overview of conducted industrial and lab-sized demonstrator case studies.	17
Table 3:	Details of rating scheme for the evaluation of existing, related approaches with + (completely satisfied), o (partially satisfied), – (not satisfied), ? (unknown) and n.a. (not applicable).	25
Table 4:	Evaluation of related approaches in the field of static code analysis.	32
Table 5:	Tool-based static code analysis of PLC software evaluated with respect to the requirements.	38
Table 6:	Related quality assessment approaches evaluated with respect to the derived requirements.	42
Table 7:	Exemplary analysis goals for static code analysis of legacy control software (not independent from each other).	57
Table 8:	Aspects to be targeted during the static code analysis (no claim to completeness).	65
Table 9:	Summary of the software comparisons during manual static code analysis by means of selected software parts, published in [FVF15].	76
Table 10:	Selected, exemplary means for visualization and documentation of analysis results (not intended to be exhaustive).	78
Table 11:	Criteria of a recommendation for action derived from static code analysis results during procedure Step 4 (*representative list without claim for completeness); enlarged from [VFN20].	85
Table 12:	Evaluation methods per requirement with reference to the relevant Sections.	98
Table 13:	Overview of conducted industrial Case Studies A, B, C and D and lab-sized demonstrator Case Study E.	100
Table 14:	Overview of targeted aspects during the first project analysis in Case Study B.	105
Table 15:	Summary of identified weaknesses, derived recommendations for action and the estimated change effort in Case Study B.	111
Table 16:	Summary lessons learned from procedure application in Case Study B.	112
Table 17:	Overview of targeted aspects during the first project analysis in Case Study C.	117

Table 18:	Excerpt of the findings from the manual analysis concerning implemented interface tasks with their dependency types and unplanned changes in Case Study C.....	120
Table 19:	Summary of identified weaknesses, derived recommendations for action and the estimated change effort in Case Study C.	122
Table 20:	Summary lessons learned from procedure application in Case Study C.....	123
Table 21:	Summarized examples from the case study evaluations targeting different requirements.....	129
Table 22:	Comparing the company experts' answers to the questions asked during the workshop.....	145
Table 23:	Summary of the evaluation of the presented procedure with respect to the requirements.....	147
Table 24:	Checklist regarding the aspects to be targeted during the static code analysis (no claim to completeness).	191
Table 25:	Answers to industry-WG question WG#1: Does the topic of efficient quality assurance of control software currently represent a challenge in your company? (single choice, mandatory).....	195
Table 26:	Answers to industry-WG question WG#2: How high do you estimate the additional effort due to subsequent corrections, bug fixes or refactoring in the control software relative to the initial development effort? (single choice, mandatory)	196
Table 27:	Answers to industry-WG question WG#3: In principle, could you imagine integrating the described procedure for static code analysis of control software into your company workflow? (single choice, mandatory).....	196
Table 28:	Answers to industry-WG question WG#5: In your opinion, is the demonstrated procedure helpful for integrating means and methods for quality assurance with static code analysis into the development process? (single choice, mandatory).....	197
Table 29:	Answers to industry-WG question WG#6: Do you find the interview guiding questions suitable for identifying the analysis goal and usable additional information for the analysis? (single choice, mandatory).....	198
Table 30:	Answers to industry-WG question WG#7: Do you consider the analysis procedure to be successfully applicable in your company with regard to the boundary conditions (such as unchangeable design decisions)? (single choice, mandatory).....	198

Table 31:	Answers to industry-WG question WG#8: After an introduction to the procedure, would you be able to independently transfer and apply the code analysis procedure to your control software? (single choice, mandatory)	198
Table 32:	Answers to industry-WG question WG#9: Are static code analysis methods or tools currently already used in the development process in your company? (single choice, mandatory).....	199
Table 33:	Answers to industry-WG question WG#11: In which industrial application sectors does your company operate? (multiple-choice, mandatory).....	199
Table 34:	Answers to industry-WG question WG#12: Do you think the procedure would be applicable in your application area? (single choice, mandatory).....	200
Table 35:	Answers to industry-WG question WG#13: Which control platforms are used in your company? (multiple-choice, mandatory).....	200
Table 36:	Single-choice questions asked during the workshop via a polling application in German.	201
Table 37:	Answers to workshop question W#1: Have you ever used static code analysis methods or tools in the development process in your daily work?	202
Table 38:	Answers to workshop question W#2: Do you find the interview guiding questions helpful for preparing the analysis and identifying the analysis goal?.....	203
Table 39:	Answers to workshop question W#3: Do you consider the analysis procedure to be successfully applicable in your company with regard to the boundary conditions (such as unchangeable design decisions)?.....	203
Table 40:	Answers to workshop question W#4: Do you think the procedure can sufficiently address the constraints of your application sector and would therefore be applicable?	203
Table 41:	Answers to workshop question W#5: From your point of view, is the documentation of the analysis results on different levels in the context of your own software helpful to identify anomalies as well as disadvantageous software elements?.....	204
Table 42:	Answers to workshop question W#6: Does the documentation enable the derivation of recommendations for action and a rough estimate of effort?	204
Table 43:	Answers to workshop question W#7: After an introduction to the procedure, would you be able to independently transfer and apply the code analysis procedure to your control software?	204

- Table 44: Answers to workshop question W#8: In principle, could you imagine integrating the described procedure for static code analysis of control software into your company workflow?..... 205
- Table 45: Answers to workshop question W#9: From your point of view, is the application of static code analysis easier with the shown procedure than without the procedure? 205

13. List of Abbreviations

Abbreviation	Description
aPS	Automated Production Systems
AST	Abstract Syntax Tree
BPMN	Business Process Model and Notation
CFG	Control Flow Graph
CPPS	Cyber-Physical Production Systems
CPU	Central Processing Unit
DB	Data Block
ECC	Execution Control Charts
FB	Function Block
FBD	Function Block Diagram
FC	Function
GPL	General-Purpose Programming Language
GVL	Global Variable List
HMI	Human Machine Interface
IDE	Integrated Development Environment
IL	Instruction List
I/O ports	Input/Output Ports
IPC	Industrial PC
KPI	Key Performance Indicator
LD	Ladder Diagram
LOC	Lines of Code
MTP	Module-Type Package
MVL-list	Motor-Valve-Limit switch list
OB	Organization Block
OO	Object Orientation
OOP	Object-oriented Programming
P&ID	Piping and Instrumentation Diagram
PLC	Programmable Logic Controller
POU	Program Organization Unit
PRG	Program
SFC	Sequential Function Chart
SLOC	Source Lines of Code
SPL(E)	Software Product Line (Engineering)
ST	Structured Text
TIA Portal	Totally Integrated Automation Portal
UDT	User Defined Type
UML	Unified Modeling Language
WMS	Warehouse Management System
XML	Extensible Markup Language

Appendix A. Interview Guidelines and Checklist

This appendix includes the list of interview guiding questions to be used in step 1 during the expert interviews (Appendix A.1) and the checklist regarding points for static analysis, including available means (no claim to completeness, Appendix A.2).

Appendix A.1 Interview Guiding Questions and Project Selection (in Procedure Step 1)

The interview guiding questions can be divided into three main groups, namely questions regarding the controlled automation system and its functionality, questions targeting the control software and questions concerning organizational aspects of the software development process.

Questions targeting the controlled automation system

These guiding questions aim to understand the controlled automation hardware and the desired process. Identifying the mechanical hardware parts, their relations and the process logic within the control software can ease understanding, especially if no documentation of the software and the process is available.

1. What function does the machine perform, what is the general process?
2. Which (“large”) variants are known from the customer's point of view?
3. Which (“large”) variants are known due to hardware alternatives?
4. Which special, machine- or process-specific boundary conditions must be fulfilled?
5. Apart from the control software itself, what documents, materials or information can be consulted in the software analysis to better understand the software, its structure and its variants?
 - *Aim 1*: Understand the functionality of the controlled machine
 - i. Which variants of the functionality are known?
 - ii. Is there a central process linking other sub-processes?
 - iii. How is the product flow through the aPS organized?
 - *Aim 2*: Include decisions from other disciplines with influence on software in the analysis
 - *Helpful material*: hardware design plan, functional description, known variation points (hardware and/or customer perspective)

Questions targeting the development workflow and the organization of the engineering process

The second block of questions is related to organizational aspects of the company and, especially, the software development workflow and involved stakeholders, including their tasks. The organizational aspects are usually closely coupled to the applied reuse strategy. Thus, the background knowledge and tasks of all involved stakeholders should be gathered.

1. Which PLC platforms are used for the control? Are there machines, which are controlled by PLCs from different vendor's (e.g., due to customer requests from different countries, when operating globally)
2. How or in which steps is the software development workflow organized? Which stakeholders are involved?
 - Which departments are involved? Is, e.g., the software development divided into module and application development?
 - How many software developers work in the company / section of the company?
 - How many software developers cooperate to program one machine? (e.g., one software developer per machine or different software developers responsible for programming a machine; separation of control engineering and automation engineering (hardware control, process logic), dedicated stakeholder for motion or safety, etc.)
 - Are parts of the software developed from a different company and need to be integrated with the software developed in house?
3. Which tools are used for software development and maintenance and to handle software evolution?
 - Which commercial or in-house developed tools are used for software development and maintenance? For example, tools for variant and/or version management, tools for configuring the software, tools for code generation or similar.
 - How are changes within the software documented? In software intended for planned reuse, e.g., library modules, and in in case of changes in the software?
4. Are there software parts that are (planned to be) reused?
 - What strategies are used to link reused, machine-independent and machine-dependent software parts?
 - What functionalities in the software are (planned to be) reused?
 - Variability of machine-specific software parts? (How much do these parts differ?)
 - According to which principle is the software modularized (e.g., hardware-oriented or function-oriented)? Is the modularization strategy used only within the control software or across disciplines (e.g., mechatronic modules)?

-
5. Are there any company-specific programming guidelines? If yes:
 - Do these apply company-wide or to specific machine types only?
 - Who created the guidelines (individual software developer or separate department) and for what purpose?
 - When were the programming guidelines established? Have the guidelines been modified since then?
 - Are there any suppliers that must adhere to the guidelines?
 - Is compliance with the rules monitored? By whom and how?
 - What is the scope of the guidelines? Examples are:
 - i. Naming conventions
 - ii. Software architecture (division of functionality, use of standardized POU's)
 - iii. Templates on different granularity levels, e.g., on project- or POU-level
 - iv. Basic structure of a module or a POU (textual/example POU, no template)
 - v. Rules for change management, e.g., in the module header
 - vi. Use of comments (How many are desired? Which abbreviations should be used (company-specific dictionary if multiple languages need to be supported))
 - vii. Use of certain programming languages for certain functionalities (e.g., LD for interlocking and SFC for process flows)? Prohibition of the use of selected programming languages (e.g., new modules must not be implemented in IL) ?
 6. What are the biggest challenges in the current development processes?
 - *Example 1:* high amount of unplanned reuse via *copy, paste and modify* → identification of common software parts suitable for planned reuse is required
 - *Example 2:* high variability of aPS and variants appear unique (no two identical machines are sold) with no possibilities for reuse (especially special purpose machines) → analysis of variant drivers and their effects on the software to find non-variable, reusable parts and possibly parameterizable parts, including a reuse concept to link them
 7. Are there any plans to cope with known pain points? If yes, which? Have any strategies been tried out lately? If yes, what and what was its outcome?

Questions targeting the control software

A block of questions focusing on the control software itself, e.g., programming guidelines, naming conventions or unique equipment identification numbers.

-
1. How many architectural hierarchy levels does the control software have and which functionality is implemented on which of these?
 2. Which programming languages are used on which hierarchy level or for which functionalities? Are, e.g., specific programming languages used for standardized and non-standardized control software parts?
 3. Which POU's control the machine behavior (step chains/sequences), which are used for hardware control?
 4. Which operating modes and/or machine states are distinguished in the control software?
 5. How does the diagnosis of fault or error states work and which error states are differentiated? (considering both, hardware and control logic)
 6. How is the communication between POU's (potentially across different PLCs) and between POU's and the HMI software implemented? How do safety circuits and HMI panel areas affect the structure of the PLC software (e.g., interlocking conditions)?
 - How is the link to HMI software realized? How many operating panels are there?
 - *Note:* Link to manual operation mode, recovery in case of an error, others
 7. Is there any information that is required and used in the control program but is brought into the PLC from outside?
 - Example: routing from the material flow controller or warehouse management system, which influences the order of and the actuators to be controlled.
 - Which interfaces to which other systems does the PLC need?
 - How is the communication with external sources (e.g., databases) implemented?

Selection of a project (depends on the selected analysis goal)

- 1) The project should be representative of the targeted machine type and the challenge and pain points being investigated - both positive and negative examples are helpful.
- 2) The functionality of the machine of the selected software project and the associated programming guidelines should be clear. The functional description, mechanical layout plan, special customer requirements, etc. should be known.
- 3) Applied reuse strategies and associated templates/libraries/code generation tools/etc. should be known and available for the analysis, if possible.
- 4) Applied/followed development process should be clear.
- 5) Knowledge of all software developers involved and the project parts for which they are responsible is beneficial if questions arise during the analysis.
- 6) Any programming guidelines or software specifications followed during the development of the selected project should be known and available.

Appendix A.2 Analysis Checklist (Used in Procedure Step 2)

The subsequent Table 24 contains a checklist for the analysis aspects introduced in Sec. 5.2.2.

Table 24: Checklist regarding the aspects to be targeted during the static code analysis (no claim to completeness).

Number	Analysis Aspect	Checklist
Aspect 1	Number and type of elements	<input type="checkbox"/> How many POU's does the software contain and what type are they (OB/PRG, FB, FC)? <input type="checkbox"/> How many global structures are contained (GVL, DBs)? <input type="checkbox"/> Is the concept "module" use in the control software structuring? <input type="checkbox"/> If yes, how is a software module defined? (single POU or group of POU's)
Aspect 2	Call graph and architectural hierarchy levels	<input type="checkbox"/> How many hierarchy levels does the control software have? <input type="checkbox"/> Which elements are connected (dependent) via calls? <input type="checkbox"/> How are their interfaces defined? (Amount of data exchanged, input and return parameters?) <input type="checkbox"/> How many data are exchanged via calls? (close coupling?) <input type="checkbox"/> Are there particularly many/few direct dependencies between POU's via calls? <input type="checkbox"/> Across how many hierarchy levels do the calls span? <input type="checkbox"/> What does the call structure look like in terms of hierarchy levels? <input type="checkbox"/> Are there any unused POU's (dead code)? <input type="checkbox"/> Which programming languages are used on which hierarchy level? <input type="checkbox"/> Which functionalities are implemented on which hierarchy levels? <input type="checkbox"/> Which POU's are linked to the controlled automation hardware? On which levels? <input type="checkbox"/> Is a modularization strategy recognizable?
Aspect 3	Structural patterns	<input type="checkbox"/> Are there any POU's, which are called frequently (indicated potential library elements)? <input type="checkbox"/> Are there any POU's with many outgoing calls? <input type="checkbox"/> Can recurring call structures be identified, e.g., as indication for reusable software parts? [Fah ⁺ 19] <input type="checkbox"/> For details about known structural patterns and the interpretation of their presence/absence cf. [Neu ⁺ 20c]
Aspect 4	Included libraries (closely linked to Aspect 6)	<input type="checkbox"/> Are there any libraries included? <input type="checkbox"/> If yes, how many libraries are contained in the project? <input type="checkbox"/> Are POU's from these libraries used? <input type="checkbox"/> Are the libraries platform supplier or company-specific libraries? <input type="checkbox"/> Which functionalities are implemented by the used library modules? <input type="checkbox"/> Are many standardized functionalities from the PLC supplier/PLC development IDE used? <input type="checkbox"/> On which levels are library modules called? <input type="checkbox"/> How is the interface between library POU's and application-specific software parts?
Aspect 5	Organization of software in development environment	<input type="checkbox"/> Are folders used to structure the elements contained in the project? <input type="checkbox"/> If yes, does the structure follow a modularization principle? For example, hardware- or function oriented? <input type="checkbox"/> Can groups of POU's / elements be recognized (indicator for encapsulation principle)?
Aspect 6	Standardized and application-specific parts (closely linked to Aspect 4 and Aspect 7)	<input type="checkbox"/> What is the degree of standardization? <input type="checkbox"/> Where are standardized software parts, where are application-specific software parts? What is the interface between the two? <input type="checkbox"/> What is the ratio of library modules to application-specific modules?

Number	Analysis Aspect	Checklist
		<input type="checkbox"/> How does the scope of the modules implementing different parts compare? <input type="checkbox"/> Which standard functions are used? <input type="checkbox"/> How much software is generated (e.g., from models), configured, part of a template or in any other form reused in a planned way?
Aspect 7	Extra-functional software parts/functionality distribution	<input type="checkbox"/> Which extra-functional tasks are contained in the PLC software? <ul style="list-style-type: none"> <input type="checkbox"/> Error handling (including diagnosis, communication of an error and resolving an error, cf. [Vog⁺22a] for error handling steps) <input type="checkbox"/> Change of operation mode <input type="checkbox"/> Operating data collection <input type="checkbox"/> How are transversal or extra-functional tasks implemented? <ul style="list-style-type: none"> <input type="checkbox"/> Centrally and in dedicated POU's <input type="checkbox"/> Decentrally (contained in every module/actuator) <input type="checkbox"/> Which functionalities are implemented in dedicated POU's / which are combined? (cf. [at Wilch] for overview on functionalities) <input type="checkbox"/> Annotation of call graph/data exchange graph with functionalities (course-grained like, e.g. [Vog ⁺ 16], or detailed such as [Wil ⁺ 22])
Aspect 8	Indirect data exchange graph	<input type="checkbox"/> What does the data exchange via global variables (or flags and data blocks) look like? <input type="checkbox"/> Is a regularity or structure recognizable? <ul style="list-style-type: none"> <input type="checkbox"/> Global structures for gathering data, e.g., alarm data or the current status from all actuators? <input type="checkbox"/> Global structures for distributing the current operation mode? <input type="checkbox"/> Organization of data in correspondence to module structure? <input type="checkbox"/> Which type of elements exchange data indirectly via global structures? Which functionality do they implement? <input type="checkbox"/> How much data is exchanged indirectly? Are standardized structures used for similar data? <input type="checkbox"/> Information flow (semantics behind exchanged data/reason for dependency)
Aspect 9	Properties of individual POU's/groups of POU's	<input type="checkbox"/> Can groups of elements be assigned to certain functionalities? <input type="checkbox"/> Can groups of elements be assigned to safety-circuits? <input type="checkbox"/> Use of software metrics with different focus, e.g., <ul style="list-style-type: none"> <input type="checkbox"/> Size, complexity, or other properties of elements <input type="checkbox"/> Size and amount of interfaces between elements (coupling, cohesion) <input type="checkbox"/> Identification of copied code parts (code clones) <input type="checkbox"/> Overall complexity distribution in the project <input type="checkbox"/> Data exchange within and between modules? <input type="checkbox"/> How frequent is the use of comments?
Aspect 10	Communication with external systems	<input type="checkbox"/> Which external systems exist that influence the control software? <input type="checkbox"/> Where are the interfaces to external systems implemented? <input type="checkbox"/> Which POU's offer the required communication functionalities? <input type="checkbox"/> Which information is communicated to external systems? <input type="checkbox"/> How and where is information from external systems used in the control logic? <input type="checkbox"/> Different types of connection to the HMI [Vog ⁺ 16]

Appendix B. Industry-WG: Questionnaire and Results

The German online expert questionnaire and detailed results for Section 7.1.5 are presented below.

Appendix B.1 Industry-WG Questions in German

The originally asked questions, including their answer options, are provided.

Seiten / Lehrstuhl für Automatisierung und Informationssysteme

Industrie-AK-Modularität: Statische Codeanalyse

Liebe Teilnehmer des Industrie-AK-Modularität,

bitte nehmen Sie sich einen kurzen Moment Zeit, um die nachfolgenden Fragen zum Impulsvortrag und der gezeigten Vorgehensweise zu beantworten. Die Erfassung des Feedbacks erfolgt anonym und selbstverständlich werden Ihre Daten vertraulich behandelt.

Vielen Dank für Ihre Unterstützung!

Umfrage

1.) Stellt das Thema der effizienten Qualitätssicherung von Steuerungssoftware in Ihrem Unternehmen aktuell eine Herausforderung dar? : *

- Sehr große Herausforderung
- Große Herausforderung
- Mittelmäßige Herausforderung
- Geringe Herausforderung
- Kaum eine Herausforderung
- Weiß ich nicht

2.) Wie hoch schätzen Sie den Mehraufwand durch nachträgliches Verbessern, Fehlerbeheben oder Refaktorisieren in der Steuerungssoftware relativ zum initialen Entwicklungsaufwand ein?: *

- Weniger als 10%
- Zwischen 10 -19%
- Zwischen 20 - 29%
- 30 % oder mehr
- Weiß ich nicht

3.) Könnten Sie sich prinzipiell vorstellen, die beschriebene Vorgehensweise zur statischen Codeanalyse von Steuerungssoftware in Ihren Unternehmensablauf zu integrieren? : *

- Ja, die gesamte Vorgehensweise ist prinzipiell integrierbar
- Ja, Teile der Vorgehensweise sind integrierbar
- Ich bin unsicher, ob die Vorgehensweise integrierbar wäre
- Nein, die Vorgehensweise lässt sich gar nicht integrieren
- keine Angabe

4.) Wo liegen Ihrer Meinung nach die größten Herausforderungen bei der Anwendung der gezeigten Vorgehensweise im industriellen Umfeld? : *

5.) Ist die gezeigte Vorgehensweise aus Ihrer Sicht hilfreich, um qualitätssichernde Mittel und Methoden der statischen Codeanalyse in den Entwicklungsprozess zu integrieren?: *

- Ja, die Vorgehensweise ist aus meiner Sicht hilfreich
- Die Vorgehensweise ist teilweise hilfreich
- Nein, die Vorgehensweise ist aus meiner Sicht nicht hilfreich
- Weiß ich nicht

Figure 38: First page of the industry-WG questionnaire in German.

6.) Halten Sie die Interviewleitfragen für geeignet zur Identifikation des Analyseziels und verwendbarer Zusatzinformationen für die Analyse?: *

Ja, die Interviewleitfragen sind geeignet
 Teilweise halte ich die Interviewleitfragen für geeignet
 Nein, die Interviewleitfragen sind nicht geeignet
 Weiß ich nicht

7.) Halten Sie das Analyseverfahren im Hinblick auf Randbedingungen (wie z. B. unveränderliche Designentscheidungen) in Ihrem Unternehmen für erfolgreich einsetzbar?: *

Ja, auf jeden Fall
 Teilweise
 Nein, das Verfahren ist nicht einsetzbar
 Weiß ich nicht

8.) Wären Sie nach einer Einführung in die Vorgehensweise in der Lage, das Code-Analyseverfahren selbstständig auf Ihre Steuerungssoftware zu übertragen und anzuwenden?: *

Ja, nach einer Einführung könnte ich die Vorgehensweise anwenden
 Teilweise könnte ich die Vorgehensweise nach einer Einführung anwenden
 Ich bin unsicher, da dies stark vom Schulungsumfang abhängt
 Nein, auch nach einer Einführung könnte ich die Vorgehensweise eher nicht anwenden
 Weiß ich nicht

9.) Werden in Ihrem Unternehmen aktuell bereits Methoden oder Werkzeuge der statischen Codeanalyse im Entwicklungsprozess eingesetzt?: *

Ja, Mittel der statischen Codeanalyse sind fester Bestandteil im Entwicklungsprozess
 Mittel der statischen Codeanalyse werden optional oder unregelmäßig im Entwicklungsprozess verwendet
 Nein, es werden keine Mittel der statischen Codeanalyse angewendet
 Weiß ich nicht

10.) Falls ja, welche Werkzeuge nutzen Sie und wie zufrieden sind Sie mit den eingesetzten Methoden und Werkzeugen?:

11.) In welchen Anwendungsbereichen ist Ihr Unternehmen tätig?: *

Automotive
 Lebensmitteltechnik
 Bau- und Baustoffmaschinen
 Fördertechnik und Intralogistik
 Pharma / Medizintechnik
 Metallindustrie
 Holzbearbeitungsmaschinen
 Verpackungstechnik
 Sonstige/Weitere
 keine Angabe

12.) Glauben Sie, dass die Vorgehensweise in Ihrem Anwendungsbereich anwendbar wäre?: *

Ja, auf jeden Fall
 Teilweise wäre die Vorgehensweise anwendbar
 Nein, die Vorgehensweise wäre nicht anwendbar
 Weiß ich nicht

Figure 39: Second page of the industry-WG questionnaire in German.

13.) Welche Steuerungsplattformen werden in Ihrem Haus genutzt?: *

- Beckhoff
- B&R
- CODESYS
- Rockwell
- Schneider Electric
- Siemens
- SIGMATEK
- STW
- Sonstige/Weitere
- keine Angabe

14.) Optional: Kontaktdaten (falls wir Sie bei Rückfragen kontaktieren dürfen):

Keine Stichwörter

[Impressum](#) | [Datenschutzerklärung](#) | [Hilfe](#)

Figure 40: Third page of the industry-WG questionnaire in German.

Throughout the text, the industry-WG questions are referred to as *WG#[question number]*, e.g., question 1 is referred to as *WG#1*.

Appendix B.2 Answers to the Online Questionnaire (translated to English)

The detailed answers to the online questionnaire conducted during the working group meeting are provided below, including the number of responses and the percentage.

Table 25: Answers to industry-WG question *WG#1*: Does the topic of efficient quality assurance of control software currently represent a challenge in your company? (single choice, mandatory)

Answer options	Number of responses (relative)
Very big challenge	1 (5%)
Great challenge	13 (65%)
Moderate challenge	6 (30%)
Low challenge	0 (0%)
Hardly any challenge	0 (0%)
I am uncertain	0 (0%)
<i>Sum (total answers n)</i>	20

Table 26: Answers to industry-WG question WG#2: How high do you estimate the additional effort due to subsequent corrections, bug fixes or refactoring in the control software relative to the initial development effort? (single choice, mandatory)

Answer options	Number of responses (relative)
Less than 10%	1 (5%)
Between 10 -19%	7 (35%)
Between 20 - 29%	3 (15%)
30% or more	8 (40%)
I do not know	1 (5%)
<i>Sum (total answers n)</i>	20

Table 27: Answers to industry-WG question WG#3: In principle, could you imagine integrating the described procedure for static code analysis of control software into your company workflow? (single choice, mandatory)

Answer options	Number of responses (relative)
Yes, the entire procedure can be integrated in principle	4 (20%)
Yes, parts of the procedure can be integrated	11 (55%)
I am unsure if the procedure would be integratable	4 (20%)
No, the procedure cannot be integrated at all	0 (0%)
No answer	1 (5%)
<i>Sum (total answers n)</i>	20

Answers to industry-WG question WG#4: In your opinion, what are the biggest challenges in applying the demonstrated procedure in an industrial environment? (free-text question, optional)

- Complexity of a project requires a very detailed analysis, where I currently cannot yet imagine how this can be fully captured with the interview guiding questions.
- Control platform-spanning tool; import and exports from the IDE; automation of processes
- The effort involved
- The development environments used would have to integrate these possibilities so that they can be executed without much effort and frequently.
- Additional financial effort must justify the benefit; affected employees must be open to change (motto: we have always done it this way)

- The capabilities of the development system to use this approach in a toolchain
- Available time; availability of experts to interpret; added value if templates and detailed programming guidelines already exist
- I think one of the biggest challenges is the change of mindset among colleagues, as these are after all much more abstract concepts that are at least unfamiliar. In this respect, success should also depend significantly on the "leadership" of the process.
- Peculiarities of the programming software, e.g., B&R Automation Studio
- Convincing the employee that it makes sense and is necessary
- To establish object-oriented program structures
- The results of the automatic code analysis must be validated and differentiated on a case-by-case basis. I, therefore, see the analysis primarily as a tool to improve code quality.
- Time pressure
- I see the individuality of the plants as the greatest challenge
- Difficult to say, currently more due to time pressure
- A reasonable relationship between the implementation of the required functions and a good quality of the software. It basically must not "hinder" the actual development.
- Generation change
- Integration into the development process, acceptance by the "affected developers".
- Peculiarities of the different programming software, e.g., B&R

Table 28: *Answers to industry-WG question WG#5: In your opinion, is the demonstrated procedure helpful for integrating means and methods for quality assurance with static code analysis into the development process? (single choice, mandatory)*

Answer options	Number of responses (relative)
Yes, the procedure is helpful from my point of view	10 (50%)
The procedure is partially helpful	10 (50%)
No, the procedure is not helpful from my point of view	0 (0%)
I do not know	0 (0%)
<i>Sum (total answers n)</i>	20

Table 29: Answers to industry-WG question WG#6: Do you find the interview guiding questions suitable for identifying the analysis goal and usable additional information for the analysis? (single choice, mandatory)

Answer options	Number of responses (relative)
Yes, the interview guiding questions are suitable	8 (40%)
In part, I think the interview guiding questions are suitable	11 (55%)
No, the interview guiding questions are not suitable	0 (0%)
I do not know	1 (5%)
<i>Sum (total answers n)</i>	20

Table 30: Answers to industry-WG question WG#7: Do you consider the analysis procedure to be successfully applicable in your company with regard to the boundary conditions (such as unchangeable design decisions)? (single choice, mandatory)

Answer options	Number of responses (relative)
Yes, definitely	4 (20%)
Partially	16 (80%)
No, the procedure is not applicable	0 (0%)
I do not know	0 (0%)
<i>Sum (total answers n)</i>	20

Table 31: Answers to industry-WG question WG#8: After an introduction to the procedure, would you be able to independently transfer and apply the code analysis procedure to your control software? (single choice, mandatory)

Answer options	Number of responses (relative)
Yes, after an introduction, I could use the procedure	6 (30%)
Partly I could use the procedure after an introduction	7 (35%)
I am unsure, as this depends strongly on the scope of training	6 (30%)
No, even after an introduction, I could probably not use the procedure	0 (0%)
I do not know	1 (5%)
<i>Sum (total answers n)</i>	20

Table 32: Answers to industry-WG question WG#9: Are static code analysis methods or tools currently already used in the development process in your company? (single choice, mandatory)

Answer options	Number of responses (relative)
Yes, means of static code analysis are an inherent part of the development process	1 (5%)
Means of static code analysis are used optionally or irregularly in the development process	4 (20%)
No, means of static code analysis are not applied	15 (75%)
I do not know	0 (0%)
<i>Sum (total answers n)</i>	20

Answers to industry-WG question WG#10: *If yes, what tools do you use and how satisfied are you with the methods and tools you use?* (free-text question, optional)

- SVN, Jenkins, Polarion
- In-house development
- We do not use such tools in the control area but certainly in the high-level language area. They are then integrated into the build chain.
- Currently, we are using the code analysis from Schneider [Electric]

Table 33: Answers to industry-WG question WG#11: In which industrial application sectors does your company operate? (multiple-choice, mandatory)

Answer options	Number of responses (relative)
automotive engineering	3 (15%)
food and beverage	2 (10%)
construction (material) equipment	1 (5%)
materials handling and intralogistics	2 (10%)
pharma / medicine	7 (35%)
metals industry	5 (25%)
woodworking machinery	0 (0%)
packaging machinery	2 (10%)
other / further	0 (0%)
No answer	1 (5%)
<i>Sum (total answers n)</i>	23

Table 34: Answers to industry-WG question WG#12: Do you think the procedure would be applicable in your application area? (single choice, mandatory)

Answer options	Number of responses (relative)
Yes, definitely	9 (45%)
Partially the procedure would be applicable	11 (55%)
No, the procedure would not be applicable	0 (0%)
I do not know	0 (0%)
<i>Sum (total answers n)</i>	20

Table 35: Answers to industry-WG question WG#13: Which control platforms are used in your company? (multiple-choice, mandatory)

Answer options	Number of responses (relative)
Beckhoff	6 (30%)
B&R	9 (45%)
CODESYS	0 (0%)
Rockwell	6 (30%)
Schneider Electric	5 (25%)
Siemens	10 (50%)
SIGMATEK	0 (0%)
STW	1 (5%)
other / further	2 (10%)
No answer	0 (0%)
<i>Sum (total answers n)</i>	23

Appendix C. Industrial Expert Workshop – Questions and Results

This appendix contains the detailed results of Section 7.3, namely the nine original single-choice questions asked during the online workshop with an industrial focus group in German (Appendix C.1) and the workshop participants' replies (translated to English) in Appendix C.2. The workshop questions are referred to as *W#[question number]* throughout the text, e.g., question 1 is referred to as *W#1*. The results were originally published in [Fis+22a].

Appendix C.1 Single-choice Questions in German

The originally asked questions, including their answer options, are provided in Table 36 below.

Table 36: *Single-choice questions asked during the workshop via a polling application in German.*

Number	Originally asked Workshop Question (Single-choice)	Answer Options
W#1	Haben Sie in Ihrem Arbeitsalltag schon einmal Methoden oder Werkzeuge der statischen Codeanalyse im Entwicklungsprozess eingesetzt?	<input type="checkbox"/> Ja, Mittel der statischen Codeanalyse sind fester Bestandteil im Entwicklungsprozess <input type="checkbox"/> Mittel der statischen Codeanalyse werden optional oder unregelmäßig im Entwicklungsprozess verwendet <input type="checkbox"/> Nein, es werden keine Mittel der statischen Codeanalyse angewendet <input type="checkbox"/> Weiß ich nicht
W#2	Halten Sie die Interviewleitfragen für hilfreich zur Vorbereitung der Analyse und zur Identifikation des Analyseziels?	<input type="checkbox"/> Ja, die Interviewleitfragen sind hilfreich <input type="checkbox"/> Teilweise halte ich die Interviewleitfragen für hilfreich <input type="checkbox"/> Nein, die Interviewleitfragen sind nicht hilfreich <input type="checkbox"/> Weiß ich nicht
W#3	Halten Sie das Analyseverfahren im Hinblick auf die Randbedingungen (wie z. B. unveränderliche Designentscheidungen) in Ihrem Unternehmen für erfolgreich einsetzbar?	<input type="checkbox"/> Ja, auf jeden Fall <input type="checkbox"/> Teilweise ist das Verfahren einsetzbar <input type="checkbox"/> Nein, das Verfahren ist nicht einsetzbar <input type="checkbox"/> Weiß ich nicht
W#4	Glauben Sie, dass die Vorgehensweise die Randbedingungen Ihres Anwendungsbereichs ausreichend berücksichtigen kann und daher anwendbar wäre?	<input type="checkbox"/> Ja, auf jeden Fall <input type="checkbox"/> Teilweise wäre die Vorgehensweise anwendbar <input type="checkbox"/> Nein, die Vorgehensweise wäre nicht anwendbar <input type="checkbox"/> Weiß ich nicht
W#5	Ist die Dokumentation der Analyseergebnisse auf verschiedenen Ebenen im Kontext Ihrer eigenen Software aus Ihrer Sicht hilfreich, um Auffälligkeiten sowie nachteilige Softwareelemente zu identifizieren?	<input type="checkbox"/> Ja, die Dokumentation ist hilfreich <input type="checkbox"/> Teilweise halte ich die Dokumentation für hilfreich <input type="checkbox"/> Nein, die Dokumentation ist nicht hilfreich <input type="checkbox"/> Ich bin unsicher

Number	Originally asked Workshop Question (Single-choice)	Answer Options
W#6	Ermöglicht die Dokumentation das Ableiten von Handlungsempfehlungen und eine grobe Aufwandsabschätzung?	<input type="checkbox"/> Ja, die Dokumentation der Analyseergebnisse ermöglicht das <input type="checkbox"/> Teilweise ermöglicht das die Dokumentation <input type="checkbox"/> Nein, die Dokumentation ermöglicht das nicht <input type="checkbox"/> Ich bin unsicher
W#7	Wären Sie nach einer Einführung in die Vorgehensweise in der Lage, das Code-Analyseverfahren selbstständig auf Ihre Steuerungssoftware zu übertragen und anzuwenden?	<input type="checkbox"/> Ja, nach einer Einführung könnte ich die Vorgehensweise anwenden <input type="checkbox"/> Teilweise könnte ich die Vorgehensweise nach einer Einführung anwenden <input type="checkbox"/> Ich bin unsicher, da dies stark vom Schulungsumfang abhängt <input type="checkbox"/> Nein, auch nach einer Einführung könnte ich die Vorgehensweise eher nicht anwenden <input type="checkbox"/> Weiß ich nicht
W#8	Könnten Sie sich prinzipiell vorstellen, die beschriebene Vorgehensweise zur statischen Codeanalyse von Steuerungssoftware in Ihren Unternehmensablauf zu integrieren?	<input type="checkbox"/> Ja, die gesamte Vorgehensweise ist prinzipiell integrierbar <input type="checkbox"/> Ja, Teile der Vorgehensweise sind integrierbar <input type="checkbox"/> Ich bin unsicher, ob die Vorgehensweise integrierbar wäre <input type="checkbox"/> Nein, die Vorgehensweise lässt sich gar nicht integrieren <input type="checkbox"/> keine Angabe
W#9	Ist Anwendung statischer Codeanalyse mit der gezeigten Vorgehensweise aus Ihrer Sicht einfacher als ohne die Vorgehensweise?	<input type="checkbox"/> Ja, mit der Vorgehensweise erscheint mir die Anwendung einfacher <input type="checkbox"/> Nein, die Vorgehensweise macht die Anwendung nicht einfacher <input type="checkbox"/> Weiß ich nicht

Appendix C.2 Answers During the Workshop (translated to English)

The detailed answers to the single-choice questions are provided below, including the number of responses and the percentage.

Table 37: Answers to workshop question W#1: Have you ever used static code analysis methods or tools in the development process in your daily work?

Answer options	Number of responses (relative)
Yes, means of static code analysis are an inherent part of the development process	2 (5.4%)
Means of static code analysis are used optionally / irregularly in the development process	8 (21.6%)
No, means of static code analysis are not applied	24 (64.9%)
I do not know	3 (8.1%)
<i>Sum (total answers n)</i>	37

Table 38: Answers to workshop question W#2: Do you find the interview guiding questions helpful for preparing the analysis and identifying the analysis goal?

Answer options	Number of responses (relative)
Yes, the interview guiding questions are helpful	19 (48.7%)
In part, I find the interview guiding questions helpful	17 (43.6%)
No, the interview guiding questions are not helpful	1 (2.6%)
I do not know	2 (5.1%)
<i>Sum (total answers n)</i>	39

Table 39: Answers to workshop question W#3: Do you consider the analysis procedure to be successfully applicable in your company with regard to the boundary conditions (such as unchangeable design decisions)?

Answer options	Number of responses (relative)
Yes, definitely	9 (22.5%)
Partially the procedure can be used	16 (40.0%)
No, the procedure is not applicable	4 (10.0%)
I do not know	11 (27.5%)
<i>Sum (total answers n)</i>	40

Table 40: Answers to workshop question W#4: Do you think the procedure can sufficiently address the constraints of your application sector and would therefore be applicable?

Answer options	Number of responses (relative)
Yes, definitely	4 (10.5%)
Partially the procedure would be applicable	14 (36.8%)
No, the procedure would not be applicable	3 (7.9%)
I do not know	17 (44.8%)
<i>Sum (total answers n)</i>	38

Table 41: Answers to workshop question W#5: From your point of view, is the documentation of the analysis results on different levels in the context of your own software helpful to identify anomalies as well as disadvantageous software elements?

Answer options	Number of responses (relative)
Yes, the documentation is helpful	15 (38.5%)
Partially I think the documentation is helpful	18 (46.1%)
No, the documentation is not helpful	1 (2.6%)
I am uncertain	5 (12.8%)
<i>Sum (total answers n)</i>	39

Table 42: Answers to workshop question W#6: Does the documentation enable the derivation of recommendations for action and a rough estimate of effort?

Answer options	Number of responses (relative)
Yes, the documentation of the analysis results enables this	7 (20.0%)
Partly the documentation enables this	21 (60.0%)
No, the documentation does not allow this	4 (11.4%)
I am uncertain	3 (8.6%)
<i>Sum (total answers n)</i>	35

Table 43: Answers to workshop question W#7: After an introduction to the procedure, would you be able to independently transfer and apply the code analysis procedure to your control software?

Answer options	Number of responses (relative)
Yes, after an introduction, I could use the procedure	7 (18.9%)
Partly I could use the procedure after an introduction	11 (29.7%)
I am unsure, as this depends strongly on the scope of training	13 (35.2%)
No, even after an introduction, I could probably not use the procedure	5 (13.5%)
I do not know	1 (2.7%)
<i>Sum (total answers n)</i>	37

Table 44: Answers to workshop question W#8: In principle, could you imagine integrating the described procedure for static code analysis of control software into your company workflow?

Answer options	Number of responses (relative)
Yes, the entire procedure can be integrated in principle	7 (18.9%)
Yes, parts of the procedure can be integrated	18 (48.7%)
I am unsure if the procedure would be integratable	11 (29.7%)
No, the procedure cannot be integrated at all	1 (2.7%)
No answer	0 (0.0%)
<i>Sum (total answers n)</i>	37

Table 45: Answers to workshop question W#9: From your point of view, is the application of static code analysis easier with the shown procedure than without the procedure?

Answer options	Number of responses (relative)
Yes, the application seems easier to me with the procedure	22 (64.7%)
No, the procedure does not make the application easier	4 (11.8%)
I do not know	8 (23.5%)
<i>Sum (total answers n)</i>	34