Applications

Jan Wilch*, Juliane Fischer, Nikolai Langer, Markus Felger, Matthias Bengel and Birgit Vogel-Heuser

Towards automatic generation of functionality semantics to improve PLC software modularization

Erste Schritte einer automatischen Funktionalitätssemantik zur Verbesserung von SPS Softwaremodularität

https://doi.org/10.1515/auto-2021-0138
Received September 21, 2021; accepted January 4, 2022

Abstract: Functions of automated Production Systems (aPS) can be realized by control software (SW), whose high quality and short development time are, therefore, vital. To achieve both, SW should be modular and, thereby, reusable. Static code analysis can help improve the modularization of existing software, e. g., by automatically analyzing control and information flow. However, manual code reviews are still typically required because planning a SW's modularization requires a semantic understanding of its functionality. This paper presents an approach to, instead, identify SW functionality automatically and evaluates it with SW from three aPS manufacturers.

Keywords: automated Production Systems, control software, functionality, semantics, static code analysis

Zusammenfassung: Die Funktionen automatisierter Produktionssysteme (aPS) können durch Steuerungssoftware (SW) realisiert werden, deren hohe Qualität und kurze Entwicklungszeit daher entscheidend sind. Deshalb sollte SW modular und somit wiederverwendbar sein. Statische Codeanalyse kann bei der Modularisierung existierender SW helfen, indem z. B. automatisch Kontroll- und Informationsfluss analysiert werden. Dennoch sind meist manuelle Analysen nötig, um ein semantisches Verständnis der SW-Funktionalität zu entwickeln. Dieses Paper stellt stattdessen einen Ansatz vor, um Funktionalität automatisch zu identifizieren und evaluiert ihn mit SW von drei aPS-Herstellern.

Schlagwörter: automatisierte Produktionssysteme, Steuerungssoftware, Funktionalität, Semantik, statische Codeanalyse

1 Challenges in control software quality assessment

The majority of industrial and consumer products are manufactured or processed by automated Production Systems (aPS) whose functions are primarily and increasingly realized by control software (SW) [1] (Note that "function" in this paper describes any goal-oriented behavior, whereas "functionalities" can be differentiated based on their specific purpose). The SW is typically executed on Programmable Logic Controllers (PLC) and implemented according to IEC 61131-3, which defines five (partially graphical) programming languages and so-called Program Organization Units (POU) to structure the SW [2]. Since aPS have lifetimes of up to several decades, companies working in the domain often manage large historically grown codebases, making the maintainability of SW crucial [3]. Combined with Industry 4.0 requirements like the global demand for highly specialized, flexible systems and a short time to market [3], it is essential to compose SW from reusable, well-tested modules (i. e., groups of one or more POUs) fulfilling a specific set of functionalities [4].

***Corresponding author: Jan Wilch,** Institute of Automation and Information Systems, Department of Mechanical Engineering, TUM School of Engineering and Design, Technical University of Munich, Munich, Germany, e-mail: jan.wilch@tum.de
Juliane Fischer, Institute of Automation and Information Systems, Department of Mechanical Engineering, TUM School of Engineering and Design, Technical University of Munich, Munich, Germany, e-mail: juliane.fischer@tum.de
Nikolai Langer, Brückner Maschinenbau GmbH & Co. KG, Königsberger Str. 5-7, 83313 Siegsdorf, Germany, e-mail: nikolai.langer@brueckner.com
Markus Felger, Matthias Bengel, Teamtechnik Maschinen und Anlagen GmbH, Planckstraße 40, 71691 Freiberg, Germany, e-mails: markus.felger@teamtechnik.com, matthias.bengel@teamtechnik.com
Birgit Vogel-Heuser, Institute of Automation and Information Systems, Department of Mechanical Engineering, TUM School of Engineering and Design, Core Member of MDSI and Member of MIRMI, Technical University of Munich, Munich, Germany, e-mail: vogel-heuser@tum.de

Static code analysis is helpful to improve the modularization of existing SW by decreasing the complexity of modules and improving their clear separation. Common techniques include an assessment of the size of POUs and the control and information flow between them [5–7]. However, current static analysis approaches usually focus on syntactic features without a mechanism of detecting semantics, i. e., the purpose of a piece of code in a larger context. An assignment of functionalities to POUs can express this semantic understanding and assist automation engineers in their decision-making, e. g., by categorizing library POUs to assemble new configurations or by helping to improve the modularity of existing SW [4]. Currently, this semantic understanding requires in-depth manual analyses of a POU's description (e. g., names or comments) and expected runtime behavior (based on its implementation). The main contribution of this paper is an automatic classification workflow that can successfully assign eleven selected functionality classes to eight industrial SW projects from three German aPS manufacturers working in different domains. Thereby, a semantic understanding of SW functionality is enabled by automatic means to assist developers in improving SW modularity. Modularity improvements can focus on limiting functionalities to specific levels in the call hierarchy, prescribing which functionalities should be fulfilled by distinct POUs, or restricting permissible relations (e. g., only allowing data exchange between selected functionalities).

The remainder of the paper is structured as follows. Section 2 derives the requirements for functionality analysis from the boundary conditions in aPS SW development, followed by an investigation of their fulfillment in the state of the art in Section 3. The concept to automatically identify control SW functionality is presented in Section 4 and evaluated in Section 5. Finally, Section 6 provides a summary and an outlook on future research.

2 Requirements for generating functionality semantics

IEC 61131-3 SW projects are structured into POUs, intended to enable reusing functionality [2]. Thus, companies frequently maintain libraries of POUs from which SW projects are composed. To assess the modularity of SW as a prerequisite for planned reuse, it is, therefore, necessary to assign functionality at the level of individual POUs (*R1: POU-level classification*).

Industrial SW projects frequently involve hundreds of POUs [6] and are, thus, too large to routinely identify each POU's functionality manually. Instead, an automatic functionality assignment is required (*R2: automation*).

To ensure that the concept matches the boundary conditions of aPS development, it must be scalable to the extensive size of industrial SW projects mentioned above. Additionally, industrial SW may be more complex than artificial examples (e. g., additional interlocks to ensure safe operation) and extensive commenting or naming conventions, which may be relevant for static functionality identification. Thus, an evaluation of the concept using industrial SW projects is needed (*R3: industrial SW*).

POUs are connected by calls, direct (DDE), and indirect data exchange (IDE), which are cumulatively referred to as a SW's architecture in this paper. Because architecture and modularization are interdependent (e. g., a POU performing much IDE is challenging to isolate as a module), a visualization of POUs as nodes and relationships as edges can aid the analysis and subsequent improvement of a SW's modularization [7]. As explained above, functionality information is essential to understand existing modularity and should, therefore, also be visualized. Further, the visualization requires means to search and filter the view (*R4: visualization*).

PLC SW often mixes the languages defined in IEC 61131-3, e. g., the graphical language Ladder Diagram to program interlocks and Structured Text for mathematical operations. Thus, the concept should support graphical and textual IEC 61131-3 languages to support the industrial practice of mixed-language projects (*R5: IEC 61131-3*).

3 State of the art in control software analysis

State-of-the-art approaches already fulfill the requirements introduced in Section 2 to a varying degree, as summarized in Table 1. If only some sources per column fulfill a requirement, they are listed in the rows.

Research in *variability management* aims to cope with the variability created by long lifecycles, small lot sizes, and unmanaged reuse approaches. The approaches identify similarities using static metrics computation (i. e., without executing the program) [8] or behavior modeling [9]. There are also attempts to aid companies in transitioning from legacy SW to managed reuse, either based on software product lines (SPL) [10] or the automatic extraction and reuse of variability [11]. For improved variability management, especially if a transition to SPLs is desired [10], SW should be reusable and, thus, modular. As described above, improving modularization requires knowl-

Table 1: Fulfillment of requirements in the state of the art. Fulfilled (✓), not fulfilled (✗), or not considered (–). If not specified otherwise, the row entries refer to all sources in the heading.

Requirement	Variability management (Linsbauer, Thaller et al. [9, 11], Rosiak et al. [8])	Static complexity analysis (Fischer et al. [6], Lucas & Tilbury [12])	Control/data flow analysis (Prähofer et al. [5])	Verification/restart safety (Grochowski et al. [14], Cha et al. [15])	Modularity and architecture (PackML [17], Vogel-Heuser et al. [7, 16, 18])	NLP (Farias et al. [20], Garousi et al. [19])
R1: POU-level classification	–	–	–	–	✓ [18]	–
R2: automation	✓	✓	✓	✓	✓ [7]	✓
R3: industrial SW	✓ [11]	✓	✓	✓ [14]	✓ [7, 16–18]	✓
R4: visualization	–	–	–	–	✗	–
R5: IEC 61131-3	✓ [8]	✓	✓	✓	✓ [7, 16–18]	✗

edge about functionality, which this paper aims to support by providing an automated approach (R1, R2, R4).

POUs must be easy to maintain, test, and understand to be reused. *Static complexity analysis* focusing on different types of complexity commonly serves to evaluate these properties [6]. While complexity metrics are well established in computer science, fewer adoptions exist that specifically consider the boundary conditions of aPS, such as the different languages and program structures [12]. Several commercial tools (e. g., *EcoStruxure Control Engineering – Verification*, *logi.CAD Static Analysis*, *Codesys Static Analysis*, and *TwinCAT 3 PLC Static Analysis*) are available to analyze guidelines regarding, e. g., POU complexity or naming conventions, and *Prähofer* et al. provide additional techniques to target architectural complexity in a *control/data flow analysis* [5]. However, all current automated complexity analyses lack semantic awareness, which is necessary to judge whether a POU is "too complex" because certain functionalities inevitably increase the complexity in different categories [6].

In aPS specifications, functional requirements regarding the processing of inputs (e. g., sensor readings) into outputs (e. g., hardware actuation) can be distinguished from extra-functional ones, describing boundary conditions for aPS operation like reliability, performance, efficiency, and safety [13]. For the *verification* of requirements, specifically regarding *restart safety*, *Kowalewski* et al. propose the Arcade.PLC framework combining static analysis, simulation, and model checking [14]. Similarly, *Cha* et al. attempt a generalized description language for test cases and automatic checking against a model of PLC SW [15]. Because SW functionality depends on the imposed requirements, improved knowledge of the functionality distribution on POU-level may allow more direct traceability from requirements to the implementation. This enables a more targeted verification of selected implementation sections.

Available descriptions of *modularity and architecture* in aPS SW comprise case studies of current industry practices [16] and suggestions in guidelines like OMAC/ISA PackML [17]. *Fuchs et al.* already provide a functionality classification that this paper will extend upon [18], and *Neumann* et al. describe design patterns (recurring kinds of relationships between multiple POUs) to identify them in a visualization (R4) [7]. However, existing approaches do not attempt completeness (i. e., all analyzed POUs can be categorized) and provide partially overlapping definitions. This paper attempts a functionality classification extended from [18] to describe all encountered descriptions and implementations of functionality unambiguously and automatically (R2), including visualization of the results (R4).

Code analysis via natural language processing (*NLP*) uses descriptive metadata like source code comments and SW element names. Thereby, automated SW testing [19] and the identification of knowingly accumulated technical debt [20] are pursued. However, no approaches targeting IEC 61131-3 could be found (R5). Because guidelines frequently prescribe naming conventions and commenting, NLP may be helpful for the automatic identification of a POU's intended functionality (R1, R2, R4) as well as potential deviations (which may be "admitted" in comments as found by *Farias* et al. [20]).

In summary, the automatic analysis of SW is already commonplace, including evaluations with industry-scale projects. Many research areas specifically focus on the languages of IEC 61131-3, excluding NLP applications. Further, previous work describes functionalities and their relation to architecture. So far, however, no approaches exist to identify POU functionality automatically and, thereby,

Table 2: Overview of the identified functionality classes.

	Functionality class	Mentioned in
	Program Main (F_{Main}): Entry point of an application or module. From here, calls extend to subordinate units.	[16]
	Sequence Control (F_{Seq}): Implementation of the nominal, sequential behavior of the machine [21]. POUs of this functionality correlate with the technical process.	[16–18, 21]
	Sensor/Actuator Control ($F_{S/A}$): Control of connected components. POUs of this functionality correlate with the machine's mechanical composition.	[2, 13, 17, 21]
	Changing Operation Modes (F_{Modes}): Changing a machine's modes of operation. An example is OMAC/ISA PackML [17], but similar approaches are also valid.	[13, 16, 17, 21]
	Diagnostics (F_{Diag}): Identification of errors. Either *technical* (about single components) or *process-related*.	[13, 16, 18, 21]
	Fault/Message Propagation (F_{Msg}): Distribution of errors identified via *Diagnostics*.	[13, 16, 21]
	Communication – HMI ($F_{HMI\text{-}C}$): Data exchange with human operators via an HMI.	[2, 13, 16–18, 21]
	Communication – Other Modules ($F_{Mod\text{-}C}$): Data exchange with conceptually separate parts of the software.	[13, 18, 21]
	Communication – External Sources ($F_{Ext\text{-}C}$): Data exchange with computer programs outside of the analyzed SW.	[2, 13, 18, 21]
	Utility (F_{Util}): Units created only for better separation of concerns and not to fulfill specific requirements of the automation problem.	[18]
⚠	**Safety Control (F_{Saf}):** Control of specialized safety hardware.	[13, 18, 21]

help improve the modularity in large-scale industrial SW projects.

4 Automatic identification of selected functionality classes

To classify functionalities, first, a set of functionality classes to describe the reviewed literature and industrial SW is introduced in Section 4.1. Using these classes, Section 4.2 introduces the concept to identify functionality automatically.

4.1 Selection of functionality classes

Typically, hundreds of POUs (the largest investigated project contains 588 POUs) are needed to fulfill the requirements of aPS by implementing various functionalities, making a manual semantic understanding difficult and time-consuming. Because, in the absence of industry-wide standards, companies develop varying approaches to functionality implementation, it is also challenging for a single manufacturer to determine a suitable granularity of functionality classes. Therefore, an analysis of descriptions and implementations of functionality from multiple heterogeneous sources serves to identify an appropriate abstraction level. The resulting classification of eleven functionality classes is depicted in Table 2 and introduced in the following.

An aPS SW uses tasks that cyclically execute the POUs from which the call structure originates (cf. the main-function in C-like languages). These POUs at the highest call-hierarchy level are classified as *Program Main* (F_{Main}) and typically perform little logic besides calling other POUs. Often, multiple levels of F_{Main} increasingly refine the modularization, e. g., matching the architectural levels of *Vogel-Heuser* et al. [16]. Analysis of industrial SW reveals that the sequential process logic [21] and hardware control are often implemented distinct from each other, such that, e. g., many processes can reuse one type of actuator's control. Therefore, the corresponding classes *Sequence Control* (F_{Seq}) and *Sensor/Actuator Control* ($F_{S/A}$) describe POUs for functional requirements. In addition, there are extensive extra-functional implementation parts [22], which are explained below, subdivided into eight functionalities.

Depending on influences like operator input or fault states, a process controlled by F_{Seq} must adapt its behavior. One possibility to realize this is ISA/OMAC PackML [17], but other approaches are also present in industrial practice. Common to all of them is the need to globally change the current mode (e. g., *Automatic* or *Manual*), represented by a functionality *Changing Operation Modes* (F_{Modes}). The related functionality *Diagnostics* (F_{Diag}) describes POUs that continuously monitor the state of the hardware (e. g., an axis is stuck) and technical process (e. g., shortage of material) to trigger a global reaction. To achieve this, additional code propagates fault data from

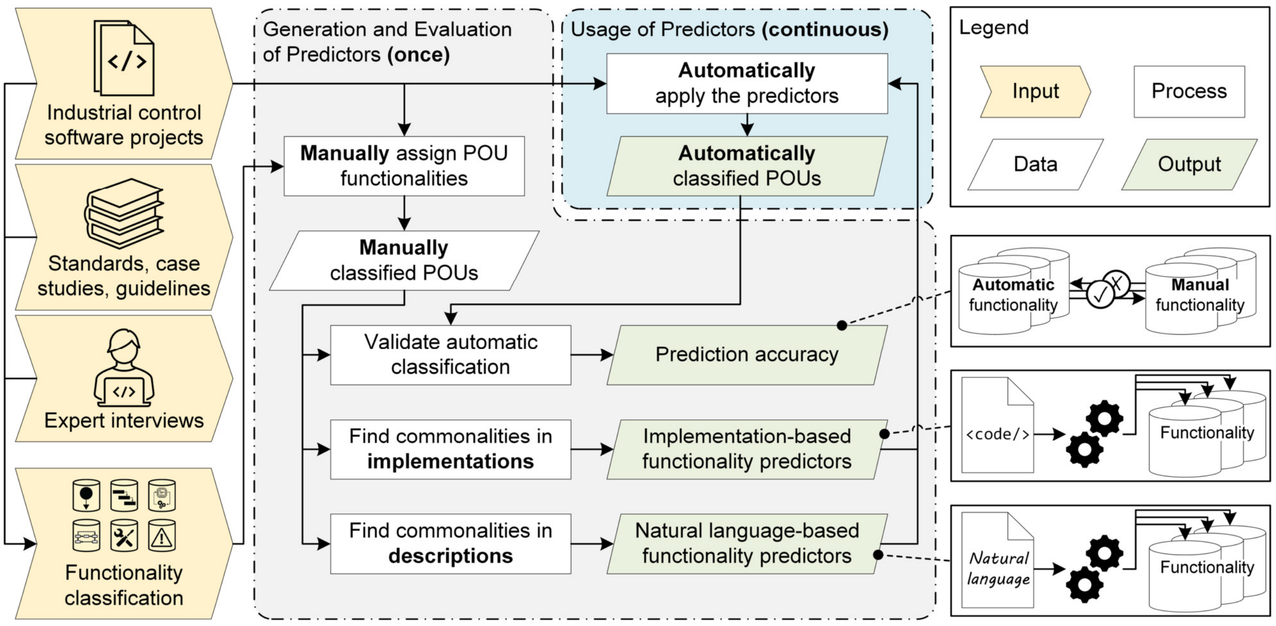
Industrial control software projects
Standards, case studies, guidelines
Expert interviews
Functionality classification
Generation and Evaluation of Predictors (once)
Manually assign POU functionalities
Manually classified POUs
Validate automatic classification
Prediction accuracy
Find commonalities in implementations
Implementation-based functionality predictors
Find commonalities in descriptions
Natural language-based functionality predictors
Usage of Predictors (continuous)
Automatically apply the predictors
Automatically classified POUs
Legend
Input
Process
Data
Output
Automatic functionality
Manual functionality
<code/>
Functionality
Natural language
Functionality

Figure 1: Concept for the automatic detection of selected functionality classes.

the point of identification to the POUs responsible for a reaction (often on higher architectural levels [16]), a functionality classified as *Fault/Message Propagation* (F_{Msg}). Besides a change of operation mode, typical reactions include an alarm to operating personnel [16], requiring operators to, in turn, impact the machine's behavior, both of which is realized via a human-machine interface (HMI) and the corresponding POUs, classified as *Communication – HMI* ($F_{HMI\text{-}C}$). Other types of frequently encountered communication concern either data exchange between logically distinct modules in the same software (*Communication – Other Modules*; $F_{Mod\text{-}C}$) or data exchange with other computer programs like databases or manufacturing execution systems via *Communication – External Sources* ($F_{Ext\text{-}C}$). Furthermore, SW often utilizes auxiliary POUs not tied to one specific aPS function or extra-functional requirement (e. g., data conversions, copy operations, or validity checks) represented by the functionality *Utility* (F_{Util}). Finally, due to the potential hazards of aPS for their environment, specialized hardware like safety doors or light barriers are necessary, whose control is performed by POUs classified as *Safety Control* (F_{Saf}).

4.2 Concept to statically predict the functionality classes

This paper aims to develop a methodology to statically derive a POU's functionality and, thereby, assist the modularization of existing legacy software. The concept is visualized in Figure 1 and depends on three types of external input data: SW projects from the industrial practice to ensure practically relevant results, existing descriptions of functionality in case studies and guidelines, and regular interviews with experts from the involved companies to refine the concept. On that basis, the functionality classes defined in Section 4.1 serve as a fourth input. The concept relies on a generation of so-called functionality predictors ("predicting" a POU's functionality based on its characteristics) that is necessary only once when adopting the procedure (grey background in Figure 1). The predictors derive a functionality classification either from the implementation, i. e., the expected runtime behavior, or from natural language descriptions in names and comments. They are created by manually classifying POUs according to the definitions in Table 2 and then identifying correlations between a POU's functionality and its implementation or descriptive texts. Additionally, the generated predictors' accuracy (defined as the ratio of POUs whose automatic functionality classification matches the manual classification) allows judging their validity. After generation, the predictors can automatically analyze any SW (blue background in Figure 1). However, it is expected that the SW to classify must adhere to similar design philosophies to produce high accuracy results, e. g., by following the same guidelines as the training data.

To *find commonalities in implementations/descriptions* (cf. Figure 1), each POU is characterized by 23 properties of its implementation and descriptive texts from six sources, both selected based on the state of the art introduced in Section 3. *Neumann* et al. describe a set of design patterns and observe multiple correlations with functionality, including tree-like patterns that emerge at POUs in higher architectural levels and end at so-called (atomic) basic modules [7], which *Vogel-Heuser* et al. previously observed to correlate with reusable functionalities like controlling connected hardware ($F_{S/A}$). Similarly, frequently called POUs often perform functionalities for extra-functional requirements, whereas POUs that call many others are found at the F_{Main} level [7]. To analyze such functionality correlations, the numbers of incoming and outgoing edges of every structural type (calls, DDE, IDE) are calculated for every POU, as well as their depth in call and data exchange hierarchies. Further, industrial case studies show that IDE is often used to communicate information from lower to higher levels of the call hierarchy [16], making them potentially beneficial to detect the data exchange functionalities F_{Msg}, $F_{HMI\text{-}C}$, $F_{Mod\text{-}C}$, and $F_{Ext\text{-}C}$. Similarly, $F_{S/A}$ POUs must likely access variables connected to a PLC's physical I/O to control the connected hardware. In addition to its structural relationships, a POU's functionality manifests in different types of complexity (e. g., size, control flow, or data structure), assessed using nine different metrics adapted from *Fischer* et al. that correlate with POU functionality [6]. Lastly, meta-information like the selected programming language is analyzed, which the IEC 61131-3 standard recommends choosing based on the fulfilled functionality [2].

A POU's functionality may correlate with its natural language description, because industry standards and company-internal guidelines frequently prescribe naming conventions and extensive commenting. Therefore, every POU's name, the names of its variables, all containing folders and the application, and all labels and comments included in the implementation are collected. Different case styles are split into separate words ("FB_SafCtrl" into "FB Saf Ctrl"), and abbreviations are resolved based on company-specific dictionaries ("Function Block Safety Control").

Because the functionality prediction, i. e., the generation of a mapping from characteristics to functionality, is a typical classification problem, machine learning (ML) classifiers are trained to solve it. Thereby, input features comprise the POU characteristics (separated into implementation- and description-based ones), and the output is one of eleven functionalities (Table 2). The classifiers are trained using the manually classified POUs, i. e., labeled training data.

A visualization fulfilling R4 serves to make the findings understandable for SW developers. This includes a graph-like display of individual POUs as nodes and their relationships (calls, DDE, IDE) as connecting edges [7]. A node's fill color denotes the functionality of individual POUs (R1). Additional functions include dragging nodes and panning and zooming the view to focus on subgraphs of interest. Further, because of the large size of aPS SW projects (cf. R2), the graph can be limited to a "slice" (i. e., only POUs that directly impact or are impacted by a selected POU), and POUs can be searched by name after which the view zooms to the corresponding node.

5 Evaluation in machine and plant manufacturing

For evaluation purposes, the presented concept is prototypically implemented as part of the ZD.B-funded project advacode, focusing on PLC SW developed in Siemens Totally Integrated Automation (TIA) V15.1 and V16. Projects are exported as TIA Openness XML and transformed into a SW model containing all relevant data for functionality classification. After generating the characteristics of implementation and description, a previously trained and imported ML classifier (if available) predicts the functionalities of all POUs. Otherwise, the POU data (manual functionality assignment and characteristics) are exported to use in open-source ML frameworks. For implementation-based classification, decision trees are trained in Scikit-learn [23], and the description-based classification uses the Stanford Classifier with Stanford CoreNLP [24]. Since decision trees are human-readable, it is possible to validate the generated prediction rules based on expert knowledge. The visualization as described in Section 4.2 is also included in the prototype and depicted in Figure 2.

Using this prototype, eight industrial example projects are analyzed as described in Figure 1, i. e., functionality predictors are generated and then evaluated regarding their prediction accuracy, thereby assessing the concept's validity for automatic functionality prediction. The concept is further evaluated based on expert feedback from the involved companies (Section 5.1), regarding the fulfillment of the requirements introduced in Section 2 (Section 5.2), and by discussing its limitations and threats to its validity (Section 5.3).

Three German aPS manufacturers participate in the evaluation (companies A, B, and C) and contribute eight SW projects from previously manufactured plants (four from company A, two each from companies B and C). In total, 2,544 POUs representing all functionalities in Table 2 are available for analysis (cf. Figure 3). All companies work in manufacturing engineering, controlling discrete and

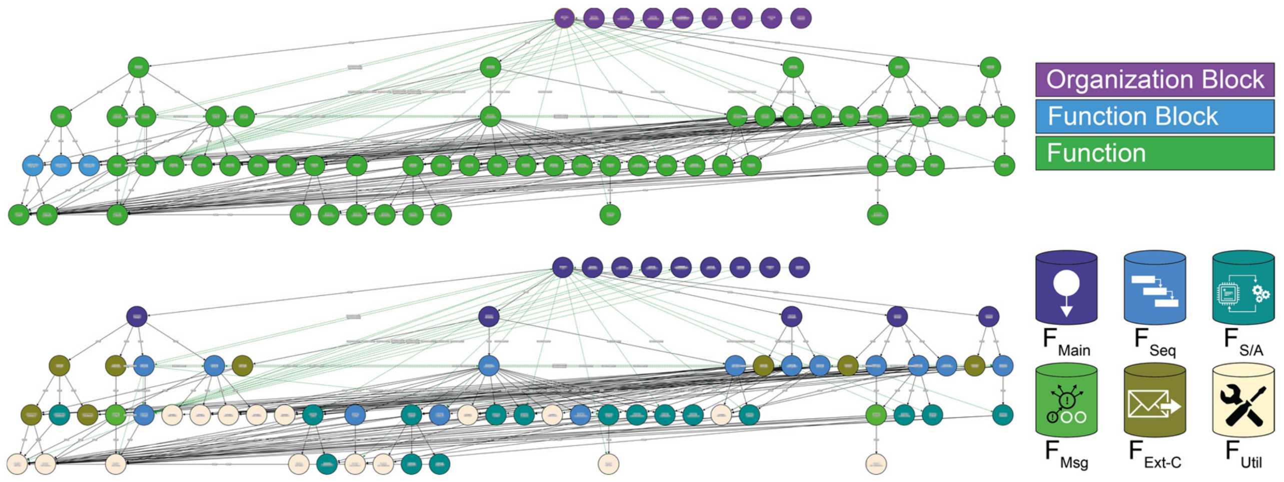

Figure 2: Graph visualizations generated in the prototype. POUs as nodes, call edges in solid black, IDE edges dashed and green. The bottom image includes functionality information generated based on NLP analysis (cf. Table 2).

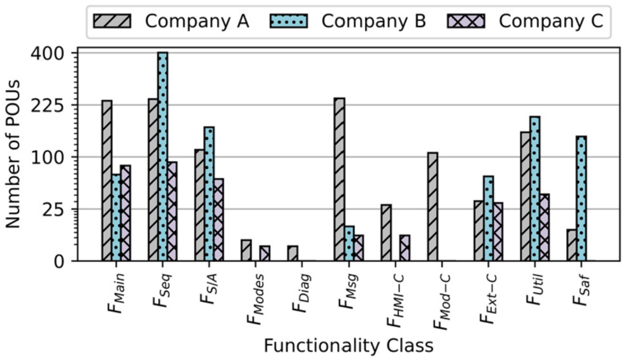

Figure 3: Available sample data per company (vertical axis scaled by square root).

continuous processes in the Automotive, MedTech, and film stretching industries. They exhibit mature development practices producing SW that allows a mostly straightforward manual assignment of one functionality per POU. The companies' SW developers follow written guidelines and partially work based on templates, automatically generated or assembled from library POUs. The guidelines include specifications about architecture, e. g., regarding the call hierarchy and permissible data exchange. While SW architectures are, thus, similar between projects from the same company, they differ notably between companies. The fact that all companies exhibit an awareness of functionality, reflected in their projects' modularity, is beneficial to the purpose of this paper, as it offers insight into the industrial practice of functionality implementation. The representation of varying modularity and architecture designs and requirements from multiple industry sectors further increases the validity of the findings.

Two projects per company (e. g., A1 and A2 from company A) are selected for training and validation and compiled into seven data sets (A – ABC, cf. Table 3). Thereby, e. g., set AB contains POUs from A1, A2, B1, and B2. No set contains the so-called holdout projects A3 and A4. Two classifiers are trained with every set: one using characteristics of a POU's implementation, the other its description. Thereby, 75 % of available POUs serve as training data, 25 % for validation. Every classifier's accuracy is computed regarding the 25 % validation POUs and all aPS SW projects. Thus, a check for overfitting is enabled, if a classifier predicts the train projects much more accurately than the validation set.

The classifiers can learn rules applicable to the entire company and not only to the selected projects, as evidenced by the similar accuracies of the holdout projects A3 and A4 compared to A1 and A2. However, the applicability to projects of a company not involved in training (e. g., A1 and A2 for classifiers BC) is minimal, likely because of the varying implementation practices described above. The average recall rate of classifier ABC-Impl regarding all functionalities is 66 %, with an average precision of 45 %. Classifier ABC-NLP performs with an average recall of 91 % and average precision of 92 %. The lowest values are measured for ABC-Impl regarding $F_{Ext\text{-}C}$ (recall 33 % and precision 18 %).

The accuracy distribution reveals that the NLP-based classification performs significantly better than the implementation-based one. This relation increases as other companies are added to the input, which only negatively affects the implementation-based accuracy. Closer inspection further reveals that projects A4 and B2 are

Table 3: Fourteen classifiers (columns; implementation-based or natural language-based) trained with seven data sets. Prediction accuracies for eight aPS SW projects (rows) in [%].

Evaluation data		**Training sets (combined aPS SW projects)**													
		A*		**B✣**		**C✦**		**AB*✣**		**AC*✦**		**BC✣✦**		**ABC*✣✦**	
		Classifiers trained with the training sets													
		Impl	**NLP**	**Impl**	**NLP**	**Impl**	**NLP**	**Impl**	**NLP**	**Impl**	**NLP**	**Impl**	**NLP**	**Impl**	**NLP**
aPS SW	A1*	85	96	22	20	24	18	77	95	79	96	28	19	72	96
projects	A2*	71	91	29	13	37	15	62	88	56	91	35	13	58	87
	A3	69	95	19	22	21	21	65	94	65	94	27	22	61	96
	A4	68	87	24	23	22	22	58	84	54	88	25	19	42	87
	B1✣	28	31	89	93	8	37	71	92	25	39	74	91	63	91
	B2✣	26	26	81	91	7	34	65	87	25	32	68	86	58	87
	C1✦	29	28	13	19	84	87	30	38	70	93	67	89	60	85
	C2✦	21	20	15	39	87	93	22	43	80	95	75	92	63	91
Validation		69	87	83	84	77	83	69	87	69	88	69	80	58	86

Legend:
Impl = classifier trained with implementation characteristics NLP = classifier trained with natural language characteristics
*training projects from company A ✣training projects from company B ✦training projects from company C
Accuracy ∈ [0, 50] % Accuracy ∈ (50, 75] % Accuracy ∈ (75, 100] %

mostly less predictable than the companies' other projects. The responsible development leads offered different explanations for this effect. A4 was reportedly created by an inexperienced developer, causing deviations from the guidelines like non-regulated IDE and the "reinvention" of available library POUs. B2, on the other hand, adheres to all guidelines but implements more functions than B1, leading to functionalities that are only present in one sample and, thus, not reliably detected. Both examples demonstrate that the functionality predictors inherently express an interpolation of the correlations between POU characteristics and functionality. Hence, insufficient accuracy means either a project deviates from the learned standard and should be revised (A4), or additional data are needed to train a well-performing classifier (B2).

5.1 Expert feedback

The visualization described in Section 4.2 and depicted in Figure 2 was successfully used in repeated workshops to describe the preliminary functionality classification results. Thereby, experts helped to iteratively refine the functionality classification to reach its current state depicted in Table 2, and the selection of POU characteristics (cf. Section 4.2) was adapted and extended. The validity of the classification results in Table 3 was confirmed, as well as the concept's merit for the industrial practice. Potential use cases include the application in quality assurance, either to maintain and improve a functionality-oriented architecture and modularization or to detect violations based on the functionality semantics (e. g., a POU having the wrong name or being too complex *for its functionality*).

Besides its usefulness to SW developers, a visualization including functionality information was also confirmed to aid SW documentation. Potential audiences include new personnel and a company's upper management, as a SW's architectural design and its relation to company guidelines can be easily grasped without in-depth code analyses.

Experts criticized the implicit assumption that every POU has exactly one functionality because it is neither always true (e. g., POUs for $F_{S/A}$ and F_{Seq} often include F_{Diag}) nor always desired. Instead, one company explained that they aim to develop more "intelligent" POUs that subsume multiple functionalities to simplify the overall architecture.

5.2 Assessment of the requirements' fulfillment

Eleven functionality classes are defined and used to fulfill R1 (*POU-level classification*) by classifying the functionalities found in literature and code reviews. In the preparatory phase (grey background in Figure 1), POUs are classified manually, but the entire remaining procedure is automated to fulfill R2 (*automation*) – including the parsing of SW projects, extraction of characteristic properties, and training and application of ML classifiers. Further, R3 (*industrial SW*) is fulfilled by evaluating the concept with eight customer projects from three German aPS manufacturers, thereby ensuring applicability to the industrial practice in different domains. The visualization (cf. Figure 2) comprises structural graphs of calls, DDE, and IDE and information about each POU's functionality via the color of nodes. Additionally, several means of interaction are included that helped to verify the fulfillment of R4 (*visualization*) by explaining and discussing the functionality analysis results with industry experts. The visualization indirectly highlights a lack of modularization, if no POUs are identified for a functionality, and enables the validation of developers' expectations about functionality distribution. Finally, R5 (*IEC 61131-3*) is only partially fulfilled because the investigated SW projects are implemented in TIA, whose distributor Siemens claims compliance with IEC 61131-3 [25] but deviates in some points. Notably, object-oriented programming is highly beneficial for modular program development [26] but unavailable in TIA. Even so, all graphical and textual languages are supported.

5.3 Threats to validity and limitations

The concept presented above is highly dependent on the large-scale analysis of industrial SW projects and, thus, the outcomes rely on the selected data. As assumed in Section 4.2 and confirmed by the results in Table 3, predictors are generally only effective when applied to projects that resemble the training data, e. g., because they were created based on the same guidelines or by the same developers. Thus, companies can only adopt the approach in principle but may have to modify the selection of functionality classes and POU characteristics.

Since the classifiers benefit from data consistency (e. g., use of naming conventions, templates, library POUs, and code generation), it is likely that companies with less mature development practices than described in Section 5 (specifically, consistent implementation of functionality in individual POUs) cannot achieve the high accuracies in Table 3. Similarly, the training of a single classifier for multiple companies shown in the rightmost columns of Table 3 likely benefits from the fact that all involved companies work in manufacturing engineering and, thus, must fulfill similar requirements. Finally, as criticized by practitioners (Section 5.1), it is unlikely that exactly one functionality from Table 2 can be assigned to every POU if companies develop less modularized SW.

6 Summary and outlook

This paper proposes a concept to automatically classify aPS SW functionality, thus helping the improvement of legacy code modularization. The presented classification scheme can comprehensively describe functionality in the reviewed literature and eight SW projects from three machine and plant manufacturing companies. All available projects are classified manually and automatically based on characteristics of a POU's implementation or description, achieving accuracies of up to 96 % in the holdout test set (cf. Table 3). However, several shortcomings still exist that future work should address.

An iterative training procedure should replace the large amount of up-front manual classification work to ease the concept's adoption for aPS manufacturers. Thereby, means to assign multiple functionalities per POU and visualize them are required. Enlargements of the investigated scope are needed to include the functionalities of other industry sectors (cf. Section 5) and different coding and modularization styles (e. g., using object-orientation or more coarse-grained modules). It is also unclear whether a "one-size-fits-all" approach (like the combined classifiers in Table 3) is feasible in this enlarged scope. As the concept's usefulness to the industrial practice depends strongly on the visualization, additional properties like metric values (size, control flow complexity, data complexity) should be displayed.

Another approach to the concept not explored here is to invert the interpretation of the observed mapping between characteristics and functionality, i. e., to derive generalizations about how functionality is implemented. By analyzing the human-readable decision trees used for implementation classification, trained with a selection of projects that are considered exceptionally well designed, it may be possible to derive best practices about how aPS functionality should be implemented. Thereby, well-modularized POUs can be designed [4] and reused in new SW configurations.

Acknowledgment: The authors want to thank all three involved companies for their support and feedback.

Funding: This work was supported by Bayerisches Staatsministerium für Wirtschaft, Landesentwicklung und Energie and supported by Zentrum Digitalisierung.Bayern under Grant DIK0112/04.

References

1. Thramboulidis, K. 2010. The 3+1 SysML View-Model in Model Integrated Mechatronics. *Journal of Software Engineering and Applications* 3(2): 109–118.
2. Programmable controllers – Part 3: Programming languages. IEC 61131-3, International Electrotechnical Commission, 2013.
3. Broy, M. 2006. The 'Grand Challenge' in Informatics: Engineering Software-Intensive Systems. *Computer* 39(10): 72–80.
4. Maga, C., N. Jazdi and P. Göhner. 2011. Reusable Models in Industrial Automation: Experiences in Defining Appropriate Levels of Granularity. *IFAC Proceedings Volumes* 44(1): 9145–9150.
5. Prähofer, H., F. Angerer, R. Ramler and F. Grillenberger. 2017. Static Code Analysis of IEC 61131-3 Programs: Comprehensive Tool Support and Experiences from Large-Scale Industrial Application. *IEEE Transactions on Industrial Informatics* 13(1): 37–47.
6. Fischer, J., B. Vogel-Heuser, H. Schneider, N. Langer, M. Felger and M. Bengel. 2021. Measuring the Overall Complexity of Graphical and Textual IEC 61131-3 Control Software. *Robotics and Automation Letters* 6(3): 5784–5791.
7. Neumann, E.-M., B. Vogel-Heuser, J. Fischer, F. Ocker, S. Diehm and M. Schwarz. 2020. Formalization of Design Patterns and Their Automatic Identification in PLC Software for Architecture Assessment. *IFAC-PapersOnLine* 53(2): 7819–7826.
8. Rosiak, K., A. Schlie, L. Linsbauer, B. Vogel-Heuser and I. Schaefer. 2021. Custom-Tailored Clone Detection for IEC 61131-3 Programming Languages. *Journal of Systems and Software* 182: 111070.
9. Thaller, H., L. Linsbauer, B. van Bladel and A. Egyed. 2020. Semantic Clone Detection via Probabilistic Software Modeling. arXiv: 2008.04891.
10. Quinton, C., M. Vierhauser, R. Rabiser, L. Baresi, P. Grünbacher and C. Schuhmayer. 2021. Evolution in dynamic software product lines. *Journal of Software: Evolution and Process* 33(2): e2293.
11. Linsbauer, L., R. E. Lopez-Herrejon and A. Egyed. 2017. Variability extraction and modeling for product variants. *Software & Systems Modeling* 16(4): 1179–1199.
12. Lucas, M. R. and D. M. Tilbury. 2005. Methods of measuring the size and complexity of PLC programs in different logic control design methodologies. *The International Journal of Advanced Manufacturing Technology* 26(5): 436–447.
13. Frank, T. et al. 2011. Dealing with non-functional requirements in distributed control systems engineering. In: *International Conference on Emerging Technologies and Factory Automation*. IEEE, pp. 1–4.
14. Grochowski, M. et al. 2020. Formal methods for reconfigurable cyber-physical systems in production. *at – Automatisierungstechnik* 68(1): 3–14.
15. Cha, S., A. Weigl, M. Ulbrich, B. Beckert and B. Vogel-Heuser. 2018. Applicability of Generalized Test Tables: A Case Study Using the Manufacturing System Demonstrator xPPU. *at – Automatisierungstechnik* 10(66): 834–848.
16. Vogel-Heuser, B., J. Fischer, S. Rösch, S. Feldmann and S. Ulewicz. 2015. Challenges for maintenance of PLC-software and its related hardware for automated production systems: Selected industrial Case Studies. In: *International Conference on Software Maintenance and Evolution*. IEEE, pp. 362–371.
17. Machine and Unit States: An implementation example of ISA-88 (*PackML*). ISA-TR88.00.02, International Society of Automation, 2008.
18. Fuchs, J. and B. Vogel-Heuser. 2012. Metrics and Methods to Restructure Modular Control Software in Special-purpose Engineering [Metriken und Methoden zur Umstrukturierung einer modularen Steuerungssoftware im Sondermaschinenbau]. In: *VDI-Kongress Automation*.
19. Garousi, V., S. Bauer and M. Felderer. 2020. NLP-assisted software testing: A systematic mapping of the literature. *Information and Software Technology* 126: 106321.
20. Farias, M. A. d. F., M. G. d. M. Neto, M. Kalinowski and R. O. Spínola. 2020. Identifying self-admitted technical debt through code comment analysis with a contextualized vocabulary. *Information and Software Technology* 121: 106270.
21. Güttel, K., P. Weber and A. Fay. 2008. Automatic generation of PLC code beyond the nominal sequence. In: *International Conference on Emerging Technologies and Factory Automation*. IEEE, pp. 1277–1284.
22. Lucas, M. R. and D. M. Tilbury. 2002. Quantitative and qualitative comparisons of PLC programs for a small testbed with a focus on human issues. In: *Proceedings of the 2002 American Control Conference (IEEE Cat. No. CH37301)*, vol. 5. IEEE, pp. 4165–4171.
23. Pedregosa, F. et al. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12(85): 2825–2830.
24. Manning, C., M. Surdeanu, J. Bauer, J. Finkel, S. Bethard and D. McClosky. 2014. The Stanford CoreNLP Natural Language Processing Toolkit. In: *Annual Meeting of the Association for Computational Linguistics: System Demonstrations*. Association for Computational Linguistics, pp. 55–60.
25. Siemens AG. *Training document for the company-wide automation solution – Appendix II IEC 61131*. Available from: https://www.automation.siemens.com/sce-static/learning-training-documents/classic/appendix/ii-iec61131-en.pdf [19 April 2021].
26. Meyer, B. 1997. *Object-oriented software construction*. Prentice Hall, Englewood Cliffs.

Bionotes

Jan Wilch
Institute of Automation and Information Systems, Department of Mechanical Engineering, TUM School of Engineering and Design, Technical University of Munich, Munich, Germany
jan.wilch@tum.de

Jan Wilch, M. Sc., graduated in Mechanical Engineering at the Technical University of Munich (TUM) in 2021 and is now pursuing a Ph. D. as a member of the scientific staff at TUM's Institute of Automation and Information Systems. His research interests include the assessment and improvement of software quality and the verification of requirement-conforming runtime behavior.

Juliane Fischer
Institute of Automation and Information Systems, Department of Mechanical Engineering, TUM School of Engineering and Design, Technical University of Munich, Munich, Germany
juliane.fischer@tum.de

Juliane Fischer received an M. Sc. in Mechanical Engineering from the Technical University of Munich (TUM) in 2017. She is currently pursuing a Ph. D. at the Institute of Automation and Information Systems at TUM. Her main research interests are the design of modular, reusable control software and methods from the field of static code analysis to enhance the reuse of variant-rich legacy control software via identification of potentials for software improvement.

Nikolai Langer
Brückner Maschinenbau GmbH & Co. KG, Königsberger Str. 5-7, 83313 Siegsdorf, Germany
nikolai.langer@brueckner.com

Nikolai Langer holds a Dipl.-Inf. (FH) degree from TH Rosenheim in Computer Science and is employed at Brückner Maschinenbau GmbH & Co. KG in Siegsdorf, Germany. His current position is Software Quality Assurance (QA) Manager for Automation and Enterprise Software. Thereby, his work includes the management of a QA team and the selection and utilization of tools and methodologies to ensure that software tests identify defects and comply with quality standards.

Markus Felger
Teamtechnik Maschinen und Anlagen GmbH, Planckstraße 40, 71691 Freiberg, Germany
markus.felger@teamtechnik.com

Markus Felger graduated from Wilhelm Büchner Hochschule in Mechatronics. At teamtechnik Maschinen und Anlagen GmbH, he currently holds the position of the Staff Software Engineer, where his work focuses mainly on standardization of PLC Software for the MedTech Business Division.

Matthias Bengel
Teamtechnik Maschinen und Anlagen GmbH, Planckstraße 40, 71691 Freiberg, Germany
matthias.bengel@teamtechnik.com

Dr. Matthias Bengel received a Dr.-Ing. degree from University of Stuttgart in Control Engineering. He holds the position of the Vice President R&D Software at teamtechnik Maschinen und Anlagen GmbH in Freiberg, Germany, where his work focuses mainly on providing automation software for automated test stands.

Birgit Vogel-Heuser
Institute of Automation and Information Systems, Department of Mechanical Engineering, TUM School of Engineering and Design, Core Member of MDSI and Member of MIRMI, Technical University of Munich, Munich, Germany
vogel-heuser@tum.de

Prof. Dr.-Ing. Birgit Vogel-Heuser received a Dr.-Ing. degree in Electrical Engineering and a Ph. D. degree in Mechanical Engineering from RWTH Aachen. Since 2009, she is a full professor and director of the Insititute of Automation and Information Systems at the Technical University of Munich (TUM). Her current research focuses on systems and software engineering. She is member of the acatech (German National Academy of Science and Engineering), editor of IEEE T-ASE and member of the science board of MIRMI at TUM.