

# Formalization of Design Patterns and Their Automatic Identification in PLC Software for Architecture Assessment

Eva-Maria Neumann\*, Birgit Vogel-Heuser\*, Juliane Fischer\*, Felix Ocker\*, Sebastian Diehm\*\*, Michael Schwarz\*\*

\**Technical University of Munich, Institute of Automation and Information Systems, Garching, Germany*  
*e-mail: {eva-maria.neumann | vogel-heuser | juliane.fischer | felix.ocker }@tum.de*

\*\**Schneider Electric Automation GmbH, Marktheidenfeld, Germany*  
*e-mail: {sebastian.diehm | michael.schwarz}@se.com*

---

**Abstract:** Due to current trends in automation technology such as mass customization and an increasing variety of products, control software (SW) in automated Production Systems (aPS) is becoming increasingly complex. Thus, the need for suitable modularization strategies as a prerequisite for planned reuse increases. In classical high-level language programming, frequently recurring problems are often solved through reusable design patterns. In the control SW development of aPS, however, this approach is still not widely spread. Hence, this paper investigates how design patterns can be used for evaluating modularity in the context of control SW architecture by proposing criteria for classifying and formalizing patterns in aPS SW structure. On that basis, a prototypical implementation is proposed to evaluate the concept and to enable an automated pattern identification and interpretation in an industrial context.

**Keywords:** Embedded computer control systems and applications, Logical design, physical design, implementation of embedded computer systems, Programmable logic controllers

---

## 1. INTRODUCTION

Automated Production Systems (aPS) are complex mechatronic systems that face increasingly demanding challenges, such as global competition and new technologies. As a result, the development of control software (SW) in aPS is subject to great time and cost pressure, leading to uncontrolled reuse strategies such as *Copy, Paste & Modify*, which hamper modularized SW architectures. However, numerous experts from industry and academia agree that modularity is a key prerequisite for high SW quality and efficient development processes. To enable modularity and systematic reuse of SW, computer science has promising approaches like object-oriented (OO) programming or reuse of design patterns, which, however, have so far hardly made their way into automation technology.

Previous work (see Fuchs *et al.* (2014)) has already identified design patterns in industrial aPS SW, but detailed research on how to formalize these patterns to assess modularity in the scope of a holistic SW architecture analysis is still lacking. Therefore, the main contribution of this paper is an approach for a comprehensive architecture analysis of aPS control SW using automated SW pattern identification. Based on criteria to describe and formalize typical structural patterns in aPS SW, an implementation is introduced to evaluate the concept and to enable automatic pattern identification in industrial SW. To assess SW architecture, the influence on modularity of the patterns' presence and absence is discussed. Besides, additional data not emerging from code analysis, but required for a holistic architecture evaluation, is identified.

The remainder of the paper is structured as follows: Section 2 presents the state of the art in the field of SW architecture and

software patterns. Next, Section 3 describes the method for using design patterns to assess aPS SW modularity. Section 4 introduces a prototypical implementation to evaluate the concept. The evaluation results are discussed in Section 5. The paper closes with a summary and outlook in Section 6.

## 2. STATE OF THE ART: SW ARCHITECTURE IN APS

This Section presents architecture definitions from computer science, architectural constraints of aPS and patterns in aPS.

### 2.1 Software Architecture in Computer Science

In computer science, several definitions of SW architecture are available. Reussner *et al.* (2019) e.g., describe SW architecture by “the general structure of a system, usually expressed in components, interfaces, and their interconnection”. Meyer (1997) identified flexible system architectures as a crucial prerequisite to ensure extendibility and reusability. These architectures are characterized by “autonomous elements connected by a coherent, simple structure”, i.e. modules. Cámara *et al.* (2017) describe SW architecture by means of structural constraints (e.g., predefined connections between components) and a set of concrete architectural element definitions (e.g., instances of components to realize the architecture). These approaches represent only an extract of the work on evaluating and classifying SW architecture in computer science. However, in the field of aPS, SW architecture has to meet different requirements due to fundamentally different boundary conditions. Hence, the considerations above cannot be transferred one-to-one to SW architecture in aPS.

## 2.2 Software Architecture in automated Production Systems

To structure the control SW architecture of aPS into reusable parts, the standard IEC 61131-3 proposes *Program Organization Units (POUs)* to encapsulate functionality and enable reuse. POUs are either Functions (FC) that return the same output value for the same input values, Function Blocks (FB) that have an internal data storage and must be instantiated for use, and Programs (PRG) that describe a control function sequence and usually form the head of the application program. Usually, aPS are controlled by *Programmable Logic Controllers (PLC)*, which are characterized by a cyclic program execution with fixed cycle times to ensure process stability. PLCs are mainly programmed in accordance to the IEC 61131-3 standard that comprises three graphical and two textual languages.

Vogel-Heuser *et al.* (2015) identified five architectural levels in aPS SW each containing SW modules controlling a certain area of a machine or plant, ranging from *plant modules* controlling whole production plants to *basic* and *atomic basic modules* reading individual sensors or controlling actuators that cannot be decomposed any further. In this context and within this paper, the term *module* refers to an individual POU in the PLC control SW (cf. Fig. 1).

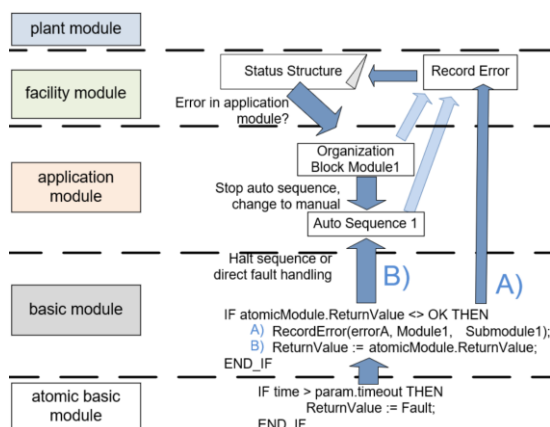


Fig. 1. Fault handling in aPS SW identified in an industrial case study, distributed to the five architectural levels identified by Vogel-Heuser *et al.* (2015)

OO programming has proven as highly beneficial for the management of SW development tasks and for flexible, reusable SW architectures. For selected runtime environments, tools supporting the OO extension of IEC 61131-3 are available, see Werner (2009). But, a survey with 68 companies from machine and plant manufacturing revealed that 42% do not use OO IEC 61131-3, see Vogel-Heuser and Ocker (2018).

## 2.3 Architectural Patterns in automated Production Systems

According to Alexander (1979), a pattern in the context of the architecture of buildings is defined as a “three-part rule, which expresses a relation between a certain context, a problem, and a solution”. The Gang of Four (GoF), see Gamma *et al.* (2011), originally defined 23 design patterns as solution approaches for recurring problems in OO programming and distinguish between creational, structural and behavioural patterns. Contrary to desired design patterns, anti-patterns describe bad

practices causing development issues, but also include methods to transform SW development problems into opportunities, see Brown (1998).

Fantuzzi *et al.* (2011) developed a design pattern to transfer UML models into IEC 61131-3 based control SW especially considering characteristics of the packaging domain. Bonfè *et al.* (2012) investigated the benefits of design patterns as reference examples for solving issues in aPS domain focusing on OO programming. However, neither of the two approaches evaluates the design patterns in regard to the SW architecture.

Fuchs *et al.* (2014) developed a method to visualize SW structures with a graphical representation comprising nodes and edges (cf. line “solution” in Tab. 1). They distinguish between direct data exchange (DDE) between POUs via calls and indirect data exchange (IDE) by writing values into and reading values from global variables. Fuchs *et al.* (2014) introduce two independent views, i.e. a call graph for IDE and one for DDE. They identified the following five patterns in industrial aPS SW: The *Tree* pattern, where each POU in the pattern is called by only one POU at a time via DDE. The *Cornflower* occurs when one central POU calls several adjacent POUs. On the contrary, a *Central Unit* occurs when multiple POUs call one central POU. The *Cuckoo* pattern is usually an unwanted structure that occurs when POUs exchange data in the IDE view, but not in the DDE view. Finally, the *Uniform Complexity* pattern refers to a SW structure in which the complexity is evenly distributed among the involved POUs. Although Fuchs *et al.* (2014) have already manually evaluated these patterns via industrial PLC code analysis, they have neither evaluated the overall SW architecture nor formalized the patterns to enable their automatic identification. Fahimi Pirehgalin *et al.* (2019) investigated how similarities in aPS SW can be found based on the Central Unit pattern by Fuchs *et al.* (2014) to identify parts in PLC SW variants, which seem suitable for planned reuse, by comparing two projects and not considering other architectural aspects.

So far, the patterns defined by Fuchs *et al.* (2014) still lack a clear description using suitable criteria as a basis for formalization. Sanz and Zalewski (2003) transferred the original definition of design patterns to the domain of control engineering and developed schemata to describe patterns but did not focus on SW patterns to enhance modularity. Fay *et al.* (2015) focused on design patterns in the context of Non-Functional Requirements (NFR) of aPS and defined categories for describing function and deployment patterns including the application context, the solution, and advantages/disadvantages of using a specific pattern. The categories of Fay *et al.* (2015) have proven to be useful for accurately describing patterns and are, therefore, adopted in the concept part of this paper to describe and formalize structural SW patterns.

## 3. CONCEPT OF UTILIZING PATTERNS TO ASSESS aPS SOFTWARE ARCHITECTURE

This section presents influencing factors on aPS SW architecture and, subsequently, describes how design patterns can be used as indicators for aPS SW architecture and modularity.

### 3.1 Influencing Factors of aPS Software Architecture

As a preliminary work to understand aPS SW architecture, it was first examined which factors affect architecture and how these influences are related. For this purpose, a literature search was first conducted, which was then adapted and extended based on expert feedback from the field of industrial automation. In this way, the following eight main categories of influencing factors on aPS SW architecture were identified:

#### Company-specific Factors

Based on the results of three comprehensive industrial surveys by Vogel-Heuser *et al.* (2017) and Vogel-Heuser and Ocker (2018), it could be observed that SW architecture is highly dependent on individual company-specific constraints such as the location(s) of the company or the educational background of engineers or technicians.

#### Type of Automated Processes

Different process types require different SW architectures. Within the process domain, e.g., continuous processes are predominant leading to strong dependencies and hampered modularity. Contrary, discrete processes, which occur, e.g., in the logistics domain, are characterized by universal, well-defined interfaces supporting a modular, reusable SW architecture.

#### Software Engineering Process

Dependent on factors like the applied reuse strategies or size of development teams, different types of architectures are beneficial. In some companies, e.g., SW is developed by small teams enabling immediate exchange, whereas larger companies employ more than hundred application engineers in different groups leading to a higher need for a clear structure of the SW architecture, as there may not be a direct exchange between the project team members.

#### Boundary Conditions from other Disciplines

aPS are mechatronic systems and, therefore, SW architecture is also dependent on constraints from electric/electronics and mechanics. In case, e.g., a mechanical or electrical hardware component is exchanged or replaced, this usually causes a change of SW in the form of adapted or new POU's.

#### Characteristics of PLC Software

As aPS are usually controlled by PLCs, the SW architecture has to cope with different boundary conditions compared to, e.g., embedded systems SW from the field of computer science due to restrictions in size (storage), calculation power or limited time for the control tasks to run the program in one cycle to ensure process stability.

#### Data Exchange between Modules

Code analysis of several industrial companies has shown that SW architecture depends to a large extent on the way data is exchanged. If, e.g., a company only uses global variables to exchange data, the possibilities of the architecture are more limited, since complex architectures with many call levels would cause a loss of clarity and control.

### Modularity and Software Hierarchy Levels

In the field of batch control, ISA-88 specifies the hierarchy of a company's physical assets. Based on survey results, Vogel-Heuser *et al.* (2017), corresponding SW hierarchies and various strategies of modularization were observed in different companies from the field of aPS leading to distinct SW architectures. In some companies, SW is modularized according to the physical structure of the machine layout. On the other hand, companies use different approaches regarding the level of standardization and reuse on different architectural levels.

#### Use of Agents, Service-oriented Architectures (SoA), CPPS

Depending on whether or not these types of SW structures are implemented, different architectures occur, see Legat and Vogel-Heuser (2015), e.g., SoA are characterized by high reusability, scalability, and interoperability and thus have positive impact on SW architecture.

To prioritize which of the factors should be addressed first for SW architecture assessment, it is necessary to identify the factors with the most critical influence on the aPS SW architecture. Hence, an influence matrix as proposed by Vester (2007) is applied to determine factors which have a high impact on but are also strongly influenced by other factors as a first starting point to evaluate architecture. Therefore, for each factor the cumulative influence on as well as the influenceability by all other factors were determined. The results were confirmed by industrial experts. Subsequently, an influence matrix was derived to classify the identified influencing factors into *active*, *passive*, *buffering* and *critical* factors (cf. Fig. 2).

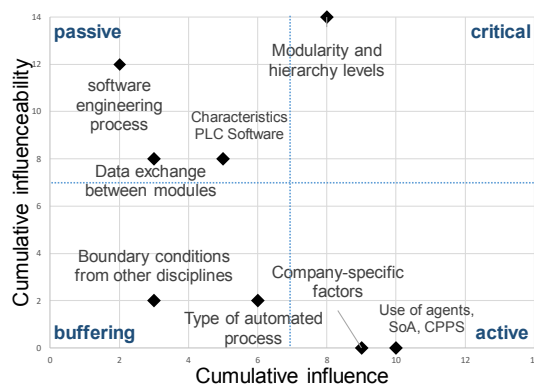


Fig. 2. Resulting Influence Matrix by comparing the eight main influencing factors on SW architecture

Active factors located in the right lower corner are characterized by strong impact on other factors, but can hardly be changed. Contrary, passive factors (left upper corner) have only a small influence, but can easily be influenced. Buffering factors (left lower corner) are characterized by low influence and influenceability. Critical factors (right upper corner) represent the most strongly interconnected influencing factors. Hence, these factors have a decisive impact on the system, whereby they are also subject to strong influenceability. As it is apparent from Fig. 2, *Modularity and hierarchy levels* represent the most critical influencing factor on aPS SW architecture. This result confirms findings from previous work by Vo-

gel-Heuser *et al.* (2017) as well as the expectation of the involved industrial experts who identified modularity as one of the key challenges in the field of industrial automation.

### 3.2. Software Patterns as Indicators for Software Modularity

Structural SW patterns address influencing factors on modularity including data exchange, interfaces between POU's, and POU size. Hence, the starting point for architecture evaluation is the identification and interpretation of typical aPS SW patterns through static code analysis and expert knowledge. The expert evaluations have shown that code analysis results are often ambiguous when it comes to interpret how the use of a particular pattern affects SW architecture, e.g. because the call graph by itself does not show what functionality is implemented with a particular pattern. Thus, additional input such as naming conventions, programming guidelines or information regarding the interaction of the controlled hardware is required to check if the overall SW architecture is consistent.

For an effective use of pattern analyses in the context of a holistic architecture assessment, it is first necessary to classify the patterns to be examined based on suitable criteria. Therefore, the criteria of Fay *et al.* (2015) are adopted (cf. Tab. 1): The *pattern type* indicates whether the given structure is a pattern or an anti-pattern and if it is characterized by data exchange or by other factors, such as the size or complexity of the involved POU's. The *pattern category* indicates in which of the GoF categories the pattern can be classified, whereby the patterns to be examined in this paper are exclusively structural patterns. The *Solution* shows the graphical representation of the patterns and a short description. Moreover, it comprises information regarding which parts of the pattern comprise application-specific or standardized SW parts. The *associate automation functions* describe for which typical functions in the field of automation the respective pattern is suitable. To enable an automated identification of the design patterns, several *parameters* are required to formally describe the patterns. Finally, the expected *advantages* and *disadvantages* regarding SW quality attributes in case a pattern is used are included. Fuchs *et al.* (2014) did not follow a formal pattern description using appropriate categories but mainly focused on their graphical representation. Therefore, each of the patterns identified by Fuchs *et al.* (2014) is enlarged, formalized and classified using the criteria described above based on expert knowledge from the domain of aPS (cf. Tab. 1).

The implementation of an automated pattern identification (cf. Step (1) in Fig. 3) is a crucial prerequisite to enable pattern interpretation in aPS software and represents an important enlargement of previous work (see Fuchs *et al.* (2014)). Details regarding Step (1) are described in Section 4. In the following, the conceptual part of the approach (cf. Step (2) and (3) in Fig. 3) is derived, i.e. the results from previous work (see Fuchs *et al.* (2014)) are enlarged by formalizing and annotating information to the design patterns to enrich the analysis of aPS SW architecture (Step (2) in Fig. 3) and it is derived, which conclusions can be drawn regarding aPS SW architecture in case the respective patterns are present or absent in the SW (Step (3), Fig. 3). The results were developed by analysing the SW

architecture of five use cases of machine and plant manufacturing applications in combination with industrial and academic expert discussions.

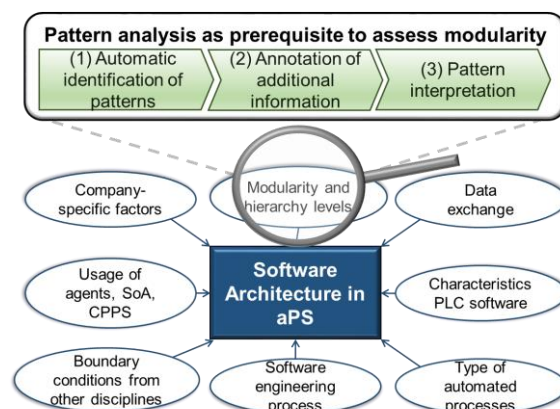


Fig. 3. Positioning pattern analysis in the context of influencing factors on SW architecture in aPS

#### Central Unit

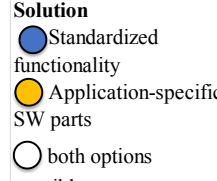
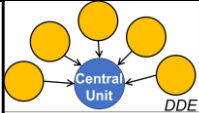

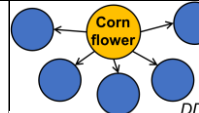
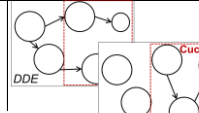
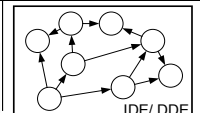
The presence of Central Units indicates high reusability and standardized interfaces of the called POU. Hence, the existence of the pattern is usually desirable and implies mature structures in the SW architecture. Moreover, Central Units often indicate that so-called infrastructural tasks are organized centrally. Infrastructural tasks represent implementation parts, which fulfil basic tasks that are not part of the application SW of the machine, e.g., error handling or change of operation modes. However, the existence of the pattern is not sufficient to indicate what kind of infrastructural task the respective SW part executes. Thus, to specify the role of the respective Central Unit, an annotation of the task type implemented by the called centre POU and by the calling ones is needed. In one of the analysed use cases, e.g., several drive components use the same alarm function. If data is mainly exchanged via global variables, the Central Unit could be a reference to a central database in which, e.g., a certain recipe is stored. Overall, a large proportion of Central Units is expected to indicate well modularized SW architecture.

Based on the experts' experience, it is exceptionally rare that no Central Unit appears in the SW. A lack of the pattern may indicate that infrastructure tasks are not implemented with/through/via/in reusable library modules, but rather decentrally in the respective application modules or outsourced to separate, non-reused modules. This prevents recurring, similar functionalities from being made reusable and hence, with regard to the SW architecture, it can be concluded that the SW is not modularized in a function-oriented way. Moreover, the absence indicates a lack of planned reuse with library modules.

#### Tree

The Tree Pattern is usually an indicator for a hierarchical modularization approach oriented towards the hardware structure. Commonly, there are modules controlling machines or plant parts, which then call modules controlling standard drives. Underneath the standard drives, usually auxiliary components are called. Preceding interviews in industry, see Vogel-Heuser *et*

Tab. 1. Classification and formalization of SW patterns according to pattern criteria (abbreviations: **HW** = hardware, **Compa** = Compatibility, **Compr** = Comprehensibility, **Main** = Maintainability, **Mat** = Maturity of SW architecture, **Mod** = Modularity, **Per** = Performance, **Re** = Reusability, **Rel** = Reliability). All listed patterns belong to category “Structural”.

Pattern name	Data exchange patterns				Metric patterns
	Central Unit	Tree	Cornflower	Cuckoo	Uniform Complexity
<b>Pattern type</b>	Pattern based on data exchange	Pattern based on data exchange	Pattern based on data exchange	Anti-Pattern based on data exchange	Pattern based on module size/complexity
<b>Scope</b>	Standardizing of recurring functionalities to avoid double implementation	Separation of control logic and standardized functionality, hierarchical, HW-oriented SW structure	Reusable POU groups to control recurring HW parts	Goal: Software should not contain Cuckoo patterns If present: Hint for hidden dependencies	Divide monolithic SW into modules with appropriate size/complexity
<b>Solution</b> 	 One POU that implements a recurring, standardizable functionality is called by several others	 POU call graph for controlling the machine behaviour resembles Tree structure (branching)	 One POU comprising application specific parts calls many other (library) POUs	 Data exchange between POU's in IDE view but no data exchange in DDE view	 Scope/complexity of all POU's is nearly the same
<b>Associated automation functions</b>	Infrastructural tasks (HMI, Alarm, ...) Library POU's (e.g. control of standard drive)	Control of system behaviour (whole machines on higher levels, sensors and drives on lower levels)	Depending on the granularity: control of entire stations or control of drive groups	No direct correlation with certain automation functions	No direct correlation with certain automation functions
<b>Parameters for formal description</b> (# = number of)	$ID_{node}$ = ID of nodes in call graph $c_{in}$ = #incoming calls $c_{in,min}$ = minimum number of incoming calls	$ID_{node}$ $l_{node}$ = call level of the respective POU $c_{in} = 1$ $b$ = number of branches	$ID_{node}$ $c_{out}$ = # outgoing calls $c_{out,min}$ = minimum number of outgoing calls	$ID_{node}$ $\#IDE_{edge}$ = # indirect data exchange per edge $\#DDE_{edge}$ = # direct data exchange per edge with DIE	$ID_{node}$ $s_{node}$ = metric result for size/complexity per node
<b>Expected Advantage regarding SW quality attributes</b> (if pattern is used / anti pattern is not used)	+ <i>Mod</i> ↑ + <i>Re</i> ↑ + <i>Mat</i> ↑	+ <i>Mod</i> ↑, <i>Compr</i> ↑: HW-oriented modularization + <i>Re</i> ↑: reuse of tree branches	+ <i>Mod</i> ↑, <i>Re</i> ↑: potential for deriving new library modules based on the called POU's + <i>Compa</i> ↑: uncomplicated rearrangement of software parts	+ <i>Mod</i> ↑, <i>Mat</i> ↑: well thought-out data exchange + <i>Re</i> ↑: reuse of SW parts not hindered by hidden dependencies + <i>Compr</i> ↑ + <i>Rel</i> ↑: decrease risk of errors	<i>Mod</i> ↑: well thought-out, modular SW (no monolithic SW), no historically grown SW + <i>Compr</i> ↑
<b>Expected Disadvantage regarding SW quality attributes</b> (if pattern is used / anti pattern is not used)	- <i>Mod</i> ↓: depending on the functionality a lot of information from callers might be required → increasing data exchange - <i>Compa</i> ↓: changes to central unit may lead to many adaptations in calling POU's	- <i>Main</i> ↓: issues regarding functions which do not fit into the HW hierarchy, e.g. error handling functions which must access information from different levels	- <i>Mod</i> ↓: risk of high amount of data exchange between different cornflowers if too HW oriented and neglect of process logic - <i>Re</i> ↓: wrong cut between application specific and standardized levels → high degree of clone & own in cornflower	- <i>Mod</i> ↓: POU interfaces to other POU's might increase if all data exchange is made direct and increased number of calls, if POU needs to access one variable value from another - <i>Per</i> ↓: increasing amount of calls resulting in increased cycle time	- <i>Mod</i> ↓: too fine-grained level of modularity decreases comprehensibility - <i>Overall SW quality</i> ↓: Optimizing SW modularity towards individual metric values can lead to other characteristics of the SW being neglected, e.g. the program runtime

al. (2017), have shown that the control logic on higher architectural levels, which specifies, e.g., when which actuator will be activated, often differs and can, therefore, not be reasonably standardized. Contrary, at the lower levels there are often standard components such as sensors or drives that are reused in various applications and are, therefore, commonly standardized. Hence, Trees often end in Central Units. The strict distribution of functionality to separate branches enables reuse of individual branches, i.e. parts controlling certain sub-parts of

machines – in case the structure is not broken up due to IDE among the POU's. To specify the SW architecture based on the identified Tree patterns, further information is needed regarding the tasks, which are located on the respective call levels (cf. Fig. 1) to derive, which functionalities (application- or infrastructure-related) are implemented in a hierarchical way.

The absence of the Tree pattern either indicates flat SW hierarchies or long chains of single calls. Unless the applications

are particularly small, and thus flat hierarchies are appropriate (which is rather rare due to the scale of industrial applications), this program structure hampers comprehensibility and maintainability. Thus, the tree pattern's absence usually indicates weaknesses regarding SW modularization and architecture.

### *Cornflower*

Since aPS SW is dependent on the controlled hardware, Cornflowers are often the result of certain requirements resulting from the mechatronic nature of the overall system. For safety reasons, e.g., only the area visible to the operator may be started from a console, which is usually implemented by a Cornflower pattern controlling the drive POU's from a higher level. The experts agreed that Cornflowers often indicate module groups of different granularity, e.g. entire stations or individual drives. Commonly, the nodes of the pattern jointly implement a certain sequence which can be considered detached from the rest of the machine, whereby the application-specific parts of the implementation are usually encapsulated on higher level. In this case, the Cornflower represents a reusable unit with certain functionality, whereby the control of sensors and actuators is often implemented on the lower levels using library modules and the logic within the centre node can be adapted according to the application. The centre POU often shows standardized interfaces, whereas the called devices are often suitable for the definition of reusable library modules. However, to enable a detailed analysis of the SW architecture, information regarding the type of functionality implemented by means of the respective Cornflower is required. If, e.g., error management is implemented using a Cornflower, it can be concluded that the error reaction is passed through from top to bottom and then group alarms are formed at the lower level.

In many cases, the entry point into the SW of a machine is implemented in the form of a Cornflower, see Vogel-Heuser *et al.* (2015), i.e. a PRG, which then calls other POU's. Hence, the absence of Cornflowers is very unusual, but could be an indication that the process does not allow for encapsulating functionality by distributing it among multiple POU's.

### *Cuckoo*

According to the definition of Brown (1998), the Cuckoo pattern represents an anti-pattern. The expert evaluation confirmed that the presence of Cuckoos often refers to weaknesses in the SW architecture: IDE is critical, as it creates connections between POU's that may not have been intended in the original structure of the programmer, especially in case these connections are not visible in the DDE view. Therefore, the pattern often hints at risky parts regarding modularization.

The absence of the Cuckoo pattern indicates well thought-out and structured SW, which is a key prerequisite for modularization. However, this case is barely found in reality. If the pattern is present, it has to be differentiated which amount of data is exchanged via global variables and how many Cuckoos can be found in the overall SW. It also has to be considered whether the exchange of information via global variables is beneficial compared to a solution with DDE. Overall, it can be concluded that the Cuckoo pattern usually indicates a defective SW architecture, but individual cases must be differentiated

regarding the number of Cuckoos, the amount of data exchanged and the intention behind using global variables. Generally, Cuckoo patterns increase the risk of errors as the SW does not follow the rule of explicit interfaces, see (Meyer, 1997), which is a prerequisite for modularity. However, for some applications, IDE is required to avoid an "inflation" of interfaces leading to increased cycle times. Thus, even if a SW architecture without Cuckoos seems to be ideally modularized, this may not be suitable for efficient debugging and maintenance or to fulfil hard real-time requirements, if, e.g., the computing load is too high. The presence of the pattern should therefore not be classified in advance as a weakness in the architecture. For definite conclusions, it is required to annotate if an identified Cuckoo pattern is intended (and why) or not.

### *Uniform Complexity*

Monolithic SW is difficult to maintain and to comprehend, thus it is reasonable to distribute functionality between different POU's that exchange data with each other. The choice of a suitable POU granularity is a compromise: small POU's are often easier to standardize, resulting in better reusability but also in POU's exchanging a lot of data thus the rule of small interfaces, see Meyer (1997), is violated. In contrast, larger POU's often have smaller interfaces (less data exchange), but tend to be less reusable, see Maga *et al.* (2011). Generally, for reasons of clarity and maintainability uniform POU sizes are desirable. In case complexity is distributed in a uniform way, it can be assumed that the architecture is well thought-out and hence, the SW is not affected by typical aPS problems such as historically grown SW due to using Copy, Paste and Modify. However, it has to be considered whether the chosen granularity is appropriate for the application, and, naturally, the complexity is HW-dependent meaning that differences cannot be avoided completely, but should be kept to a minimum.

An absence of the pattern indicates that the complexity is not evenly distributed among the POU's and, hence, a reconsideration of the functionality distribution among the modules might be beneficial. Moreover, the absence of the Uniform Complexity pattern can be a hint that the SW is not modularized oriented towards the complexity of POU's or towards the scope of encapsulated functionality. Additional data from the user concerning the applied modularization approach, e.g. whether the SW is organized towards the hardware, is needed.

## 4. PROTOTYPICAL IMPLEMENTATION

An automated pattern identification is a key prerequisite to use pattern analysis in an industrial workflow. Hence, a prototypical implementation (cf. Step (1) in Fig. 2) using *SPARQL Protocol and RDF Query Language (SPARQL)*, i.e. a graph-based query language, see W3C (2013) is proposed. A SPARQL query comprises three elements, i.e. the namespace definitions that are used for the query, the identifier of the query type, and the pattern to be matched. The SPARQL queries are applied to graph-based code representations based on the Dependency Model of Schneider Electric, see Feldmann *et al.* (2016), which allows the representation of characteristics of an IEC 61131-3 project with nodes and edges. The prototypical implementation comprises the identification of Cornflowers and

Central Units in the DDE view. To automatically identify both patterns, the following algorithms were implemented:

```
#Cornflowers
for IDnode in project
    count outgoing calls as cout
    return if (cout > coutmin - 1)

#Central Units
for IDnode in project
    count incoming calls as cin
    return if (cin > cinmin - 1)
```

Currently, the implementation does not include a possibility for the programmer to enter additional information, but some options were considered in joint brainstorming sessions with industry experts, such as asking the user to annotate details regarding the intention behind the identified patterns by means of a Graphical User Interface or by providing information such as programming guidelines or naming conventions by means of additional files that can be read by the implementation.

### 5. EVALUATION: INDUSTRIAL EXAMPLE PROJECT

The proposed approach (cf. Fig. 2) is evaluated in two steps: *First*, the automated pattern identification by means of SPARQL queries (see Section 4) is applied to an IEC 61131-3 project controlling a showcase demonstrator. Then, it is validated by means of manual analysis whether the implemented functionality of the identified patterns meets the expectations formulated in Tab. 1. *Second*, the occurrence of Central Units and their percentage share in real machine and plant control software is manually analysed for six industrial PLC projects.

The demonstrator used for the first part of the evaluation represents a robot system serving beverages to users. The analysed project comprises 18 PRGs, 119 FBs, and 75 FCs. The execution of the project is initiated by the *MainMachine* PRG which calls three sub-programs. For identifying Cornflowers and Central Units in the example projects, the parameters  $c_{out,min}$  and  $c_{in,min}$  are set to ten.

By applying the prototypical implementation to the example project, 19 Cornflowers are identified using SPARQL queries and analysed manually. As expected, nine of the ten nodes with most outgoing calls are PRGs. Moreover, all identified Cornflowers belong to the application-specific part of the project, which corresponds to the expectations.

Eight of the identified Cornflowers are also detected when executing the query for Central Units, i.e. these program parts have at least ten outgoing *and* ten ingoing calls. This observation confirms the assumption that Cornflowers often represent a reusable POU implementing a specific functionality, which is then called by other modules. In the case of the analysed example project, the elements that fulfil both the criteria of a Cornflower and a Central Unit are mainly PRGs implementing application-specific tasks. One of the identified PRGs (PRG1), e.g., controls seven axes of the machine for serving trays to the user. In total, 151 outgoing calls were counted for PRG1. On the other hand, the *MainMachine* PRG node and the corresponding sub-nodes call PRG1 47 times, but do not call the

axes controlled by it. Hence, PRG1 represents a central interface to access the axis control of the robot system (cf. Fig. 4). The POUs called by PRG1, i.e. FBs controlling the axes, represent instances of a library FB and, therefore, the assumption is confirmed that the called POUs of a Cornflower are or at least have the potential for reusable library modules. Similar observations were made for the other Cornflowers, which also meet the definition of a Central Unit. Hence, in the scope of the investigated exemplary project, the assumptions regarding the interpretation of Cornflowers can be confirmed.

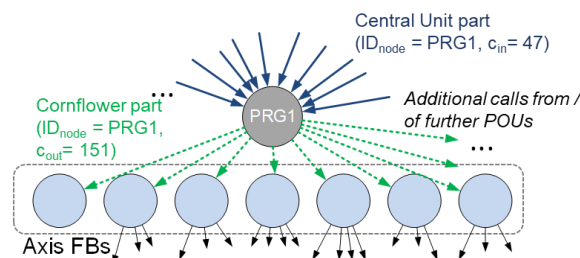


Fig. 4. Call graph extract to illustrate the search algorithm for Central Units (green, dotted lines) and Cornflowers (blue, continuous lines) in showcase demonstrator PLC project

In total, 32 Central Units were identified. Based on the applied naming conventions, it is apparent that 19 of the Central Units fulfil tasks for web visualization applications, three fulfil diagnosis tasks, and one implements exception handling. Hence, half of the identified Central Units implement infrastructural tasks, which is in accordance to the assumptions of the experts. The called POUs are mainly reusable POUs, which are either generated or library modules. This also supports the experts' assumptions. In total, the investigated project shows a well thought-out, mature architecture, which is also indicated by the high number of Central Units in the SW. Hence, the assumptions regarding the presence of Central Units could be confirmed for the example project.

In the second step of the evaluation, the formalized pattern description (cf. Tab. 1) was used to manually identify Central Units in six real industrial PLC SW projects controlling machines or plants by looking at the call graphs. Depending on the number of POUs in the considered projects, different empiric threshold values for  $c_{in,min}$  are proposed (cf. Tab. 2):

Tab. 2. Distribution of Central Units (CU) in the analysed industrial PLC SW projects

Project	#CU	$c_{in,min}$	#POUs	% of CU
1	10	10	258	3,8%
2	2	7	124	1,6%
3	6	10	168	3,5%
4	1	7	133	0,75%
5	3	7	100	3%
6	3	7	102	2,9%

The manual analysis showed that Central Units occur in each of the analysed industrial projects. A high proportion of Central Units indicates that more functionality is encapsulated in reusable POUs, and can thus be an indicator of a superior SW modularization. Hence, the resulting criteria and formalization of the patterns (cf. Tab. 1) in combination with the derived pattern interpretation (cf. Section 3.2) are the basis for a holistic PLC SW architecture assessment.

## 6. CONCLUSION AND OUTLOOK

The paper describes an approach to use pattern analysis for SW modularity and architecture evaluation by providing criteria for classifying and formalizing typical aPS SW patterns. Furthermore, the presence or absence of patterns was interpreted regarding SW architecture. It was discussed with industrial experts up to which limit information can be obtained from pure code analysis and which additional information is required from the user for a holistic architecture evaluation. A prototypical implementation for the automatic identification of Cornflowers and Central Units in aPS SW projects was developed to evaluate industrial example projects and to validate the considerations regarding pattern interpretation. Additionally required information was analysed, e.g. the applied naming conventions, but so far only manually or by discussions with programmers.

Overall, the results of this paper lay the foundation for a comprehensive assessment of modularity in the context of aPS SW architecture. Nevertheless, further research is required to implement pattern-based modularity assessment in large-scale industrial applications. To annotate additional information, e.g., the implementation of a user input mask that is intuitively understandable will be investigated to extract certain knowledge from the programmers' minds to automate not only identification of patterns but also their interpretation. Also, current challenges like OO programming will be addressed in future work. Moreover, the investigation of different ways of data exchange within and between SW parts implementing coherent automation functions, such as, e.g., fault handling, HMI linkage, operation mode handling, or application-specific functionalities will be investigated to enlarge the derived SW patterns. Also, research will be done on how patterns can be described and formalized more precisely using appropriate quality and complexity metrics to measure characteristics of involved POUs.

## REFERENCES

- Alexander, C. (1979), *The timeless way of building*, Center for Environmental Structure series, Vol. 1, Oxford Univ. Press, New York, NY.
- Bonfè, M., Fantuzzi, C. and Secchi, C. (2012), "Design patterns for model-based automation software design and implementation", *CEP*, Vol. 21 No. 11, pp. 1608–1619.
- Brown, W.J. (1998), *AntiPatterns: Refactoring software, architectures, and projects in crisis*, Wiley computer publishing, Wiley, New York.
- Cámara, J., Garlan, D. and Schmerl, B. (2017), "Synthesis and Quantitative Verification of Tradeoff Spaces for Families of Software Systems", *ECSA 2017*, Vol. 10475, pp. 3–21.
- Fahimi Pirehgalin, M., Fischer, J., Bougouffa, S. and Vogel-Heuser, B. (2019), "Similarity Analysis of Control Software Using Graph Mining", *17th IEEE Int. Conf. on Industrial Informatics (INDIN)*, pp. 508–515.
- Fantuzzi, C., Secchi, C. and Bonfè, M. (2011), "A Design Pattern for translating UML software models into IEC 61131-3 Programming Languages", *IFAC Proceedings Volumes*, Vol. 44 No. 1, pp. 9158–9163.
- Fay, A., Vogel-Heuser, B., Frank, T., Eckert, K., Hadlich, T. and Diedrich, C. (2015), "Enhancing a model-based engineering approach for distributed manufacturing automation systems with characteristics and design patterns", *JSS*, Vol. 101, pp. 221–235.
- Feldmann, S., Hauer, F., Ulewicz, S. and Vogel-Heuser, B. (2016), "Analysis Framework for Evaluating PLC Software: An Application of Semantic Web Technologies", *IEEE Int. Symp. on Ind. Electronics (ISIE)*, pp. 1048–1054.
- Fuchs, J., Feldmann, S., Legat, C. and Vogel-Heuser, B. (2014), "Identification of Design Patterns for IEC 61131-3 in Machine and Plant Manufacturing", *IFAC Proceedings Volumes*, Vol. 47 No. 3, pp. 6092–6097.
- Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (2011), *Design patterns: Elements of reusable object-oriented software*, 39th ed., Addison-Wesley, Boston.
- Legat, C. and Vogel-Heuser, B. (2015), "An Orchestration Engine for Services-Oriented Field Level Automation Software", in *Service Orientation in Holonic and Multi-agent Manufacturing, Studies in Computational Intelligence*, Vol. 594, Springer, Cham, pp. 71–80.
- Maga, C., Jazdi, N. and Göhner, P. (2011), "Reusable Models in Industrial Automation: Experiences in Defining Appropriate Levels of Granularity", *IFAC Proceedings Volumes*, Vol. 44 No. 1, pp. 9145–9150.
- Meyer, B. (1997), *Object Oriented Software Construction*, 2nd Edition, Prentice Hall, New Jersey.
- Reussner, R., Goedicke, M., Hasselbring, W., Vogel-Heuser, B., Keim, J. and Martin, L. (2019), *Managed Software Evolution*, Springer International Publishing, Cham.
- Sanz, R. and Zalewski, J. (2003), "Pattern-based control systems engineering", *IEEE Control Systems*, Vol. 23 No. 3, pp. 43–60.
- Vester, F. (2007), *Die Kunst vernetzt zu denken: Ideen und Werkzeuge für einen neuen Umgang mit Komplexität ; ein Bericht an den Club of Rome*, Dtv, Vol. 33077, Dt. Taschenbuch-Verl., Munich.
- Vogel-Heuser, B., Fischer, J., Feldmann, S., Ulewicz, S. and Rösch, S. (2017), "Modularity and architecture of PLC-based software for automated production Systems: An analysis in industrial companies", *JSS*, Vol. 131, pp. 35–62.
- Vogel-Heuser, B., Fischer, J., Rösch, S., Feldmann, S. and Ulewicz, S. (2015), "Challenges for maintenance of PLC-software and its related hardware for automated production systems: Selected industrial Case Studies", *31st Int. Conf. on Software Maintenance and Evolution (ICSME)*, pp. 362–371.
- Vogel-Heuser, B. and Ocker, F. (2018), "Maintainability and evolvability of control software in machine and plant manufacturing — An industrial survey", *CEP*, Vol. 80, pp. 157–173.
- W3C (2013), "W3C Recommendation - SPARQL 1.1 Protocol", available at: <https://www.w3.org/TR/2013/REC-sparql11-protocol-20130321/> (accessed 13 August 2019).
- Werner, B. (2009), "Object-oriented extensions for iec 61131-3", *IEEE Industrial Electronics Magazine*, Vol. 3 No. 4, pp. 36–39.