The Journal of Systems and Software 166 (2020) 110575

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

A classification framework for automated control code generation in industrial automation☆

Heiko Koziolek[a,*], Andreas Burger[a], Marie Platenius-Mohr[a], Raoul Jetley[b]

[a] ABB Corporate Research Center Germany, Wallstadter Str. 59, D-68526 Ladenburg, Germany
[b] ABB Corporate Research, Bhoruka Tech Park, ITPL Main Rd, Mahadevapura, Bengaluru, Karnataka 560048, India

ARTICLE INFO

Article history:
Received 25 July 2019
Revised 10 February 2020
Accepted 14 March 2020
Available online 16 March 2020

Keywords:
Software design and implementation
Industrial automation
Control engineering
Model-driven development
Code generation
UML / SysML

ABSTRACT

Software development for the automation of industrial facilities (e.g., oil platforms, chemical plants, power plants, etc.) involves implementing control logic, often in IEC 61131-3 programming languages. Developing safe and efficient program code is expensive and today still requires substantial manual effort. Researchers have thus proposed numerous approaches for automatic control logic generation in the last two decades, but a systematic, in-depth analysis of their capabilities and assumptions is missing. This paper proposes a novel classification framework for control logic generation approaches defining criteria derived from industry best practices. The framework is applied to compare and analyze 13 different control logic generation approaches. Prominent findings include different categories of control logic generation approaches, the challenge of dealing with iterative engineering processes, and the need for more experimental validations in larger case studies.

1. Introduction

Software development for industrial automation applications requires substantial design and implementation efforts (Gutermuth, 2010). Besides various client-server applications for supervision and monitoring in this domain, time-critical control software runs on real-time controllers and manages complex production processes of chemical plants, power plants, oil platforms, pulp and paper mills, or steel mills (Martin and Hale, 2010). Control engineers often create this control software in programming languages according to IEC 61131-3, namely function block diagrams, structured text, sequential function charts, ladder diagrams, or instruction lists (Tiegelkamp and John, 1995). The software executes cyclically and computes control signals for actuators, such as valves, motors, or robots, based on periodically sampled sensor signals as inputs (e.g., for temperature, flow, pressure, level, etc.). The software ensures efficient and safe execution of the production process. The development of this software is embedded into a larger engineering process that also covers hardware engineering, HMI development, and device configuration (Gutermuth, 2010).

Engineering industrial plants is an expensive process today and may require several person years for a mid-sized plant with several thousand I/O points (Gutermuth, 2010). Requirements and designs for such plants come from automation customers or engineering contractors, who often provide mostly informal specifications. Control engineers of automation providers partially import these specifications into engineering software tools and manually interpret them to implement the required control logic (Urbas et al., 2012). This process exhibits media discontinuities, where data is manually transformed between different formats. This may lead to data inconsistencies requiring time-consuming feedback loops with the customer. Furthermore, control engineers often use low-level programming abstractions (Lukman et al., 2013) and execute repetitive implementation tasks because of missing reusable components (Schmidberger et al., 2006). Manually written code needs to be tested thoroughly to ensure safe plant operation.

Due to the high engineering costs, practitioners and researchers have sought methods and tools for automating control logic development (Vyatkin, 2013). Practitioners introduced bulk engineering to semi-automatically deal with list-based specifications and created engineering libraries to bundle reusable functionality (Gutermuth, 2010). Researchers worked on higher-level modeling languages (e.g., applying UML or SysML for industrial automation (Vogel-Heuser et al., 2005)) and proposed rule-based code generation approaches (Drath et al., 2006). An overview of these approaches is missing, which complicates identifying and selecting an appropriate approach or to identify research opportunities. Existing literature reviews in this area (Vyatkin, 2013; Lukman et al., 2013; Yang et al., 2014; Liebel et al., 2014; Vogel-Heuser et al., 2015) usually do not focus on the aspect of automatic code

☆ Edited by Prof Doo-Hwan Bae.

* Corresponding author.
E-mail address: heiko.koziolek@de.abb.com (H. Koziolek).

https://doi.org/10.1016/j.jss.2020.110575
0164-1212/

generation and rather provide coarse-grained descriptions of several selected approaches. They do not provide detailed classification frameworks nor mirror existing approaches against requirements from practice. In this work, we aim to close this gap.

The contribution of this paper is a classification framework for automated code generation approaches for IEC 61131-3 control logic. The classification framework defines categories and criteria to evaluate such code generation approaches. This can help researchers and practitioners to compare different approaches regarding their capabilities. Using the classification framework, this paper analyzes 13 different code generation approaches from literature in the last 15 years. This analysis identifies patterns and commonalities between the approaches as well as research gaps. Based on findings from the comparison, this paper also discusses implications and required future work.

The 13 approaches map into three different categories: rule-based engineering, higher-level programming, and higher-level programming using a plant structure as input. Approaches within the same category use similar input specifications. Most approaches have been demonstrated on small lab examples so far, some approaches were tested in student experiments. Challenges for all approaches are missing support for standard input formats, iterative engineering processes, and dealing with natural language requirements. Recent standardization initiatives could improve the situation, warranting further research and more extensive validations.

This paper is structured as follows. Section 2 recaps basics of control logic generation using a running example, sketches a generic transformation process, and surveys inputs available in practice. Section 3 describes the research methodology underlying this study, provides a short overview on the identified approaches, and then proposes and explains our classification framework. Section 4 applies the classification framework to the 13 identified approaches and provides a detailed analysis of their inputs, transformations, and outputs. Section 5 discusses selected findings from the comparison in Section 4 and sketches challenges to be tackled in future work. Section 6 analyzes threats to validity of the present study, and Section 7 summarizes related work. Section 8 concludes the paper.

2. Control logic generation: Basics and requirements

Control engineers design and implement control logic based on customer requirements and design specifications as well as available control libraries and industrial standards (Gutermuth, 2010). This section provides an overview of the available customer *inputs* in practice, requirements, methods, and tools for model *transformation* and code generation, as well as different types of *outputs*, in terms of the different control logic parts and storage formats. Fig. 1 provides a generic process model for control logic generation, whose different elements will be detailed in the following. Section 2.1 describes the available inputs in detail, Section 2.2 explains each of the six transformation steps starting with “Input Validation”, and Section 2.3 provides concepts and examples for outputs of the transformation.

Today, the implementation of control logic in practice is usually only partially making use of code generation, for example by importing information from I/O lists to create signal variables for the control code (Gutermuth, 2010). As a result, the implementation process still requires control engineers a significant amount of manual interpretation to translate the customer specifications into control algorithms, which makes this procedure time-consuming and error-prone. Approaches for control logic generation aim to increase the amount of generated code, both to shorten implementation time and to improve quality. This section uses a running example of a simple production process to illustrate the concepts. It is based on the Festo MPS PA Compact Workstation[1] a minimal didactic plant for educational purposes (Steinegger and Zoitl, 2012).

2.1. Inputs

The inputs used by engineers for control logic implementation vary across different application domains. According to ANSI / ISA 5.06.01-2007[2] (Functional Requirements Documentation for Control Software Applications), a User Requirements Specification (URS) includes piping and instrumentation diagrams (P&IDs), an instrument list, and process flow diagrams. From these artifacts, instrument tag table (or I/O list), interlock matrix, sequence matrix, and Human Machine Interface (HMI) are developed. These artifacts are used as the Functional Requirements Specification (FRS), which also pertains other aspects besides control logic. However, projects often do not strictly follow this guideline and provide other or alternative artifacts. The following paragraphs detail P&IDs, I/O lists, logic diagrams, control narratives, and flow charts, because they are the most important inputs specifically for designing and implementing the control logic.

P&IDs: A P&ID provides a graphical overview of a plant segment. There can be hundreds of P&IDs to describe a larger plant. For control logic engineering, they provide an important overview of a plant and show the connections between the different instruments besides some explicit control loops. Fig. 2 shows an exemplary P&ID for a process where two substances are mixed, heated, and then pumped into another tank. The diagram includes two large tanks (column shapes T101, T102), a pump (large circle with inner spike), a motor (circle labeled ’M’), valves (small triangles), pipes (lines), and various instruments (circles), which are typical P&ID elements. The instruments describe required temperature / flow / pressure / level sensors, controllers, and actuators (e.g., valves, heaters). There are different standards for the shapes (e.g., ISO 10628, ANSI/ISA 5.1) and identifications in P&IDs (e.g., IEC 62424/ISO 3511, ISO 14617-6, ANSI/ISA 5.1) (Sun, 2013).

Nowadays, engineers usually specify P&IDs with dedicated CAD tools, such as AutoCAD P&ID[3] SmartPlant P&ID[4] COMOS P&ID[5] or OpenPlant P&ID[6]. Additionally, generic drawing tools support P&ID visual shapes (e.g., from ISO 10628), such as Visio[7] Edraw[8] or Lucidchart[9]. Common file formats are AutoCAD DWG for generic drawings (binary) and the AutoCAD DXF interchange format (plain text). Several tools also support exports of equipment and instrument lists to Microsoft Excel tables. However, in today’s engineering projects P&IDs are often only available as PDF exports containing bitmaps for control engineers. Typically, this is not a problem to date, as they are usually subject to manual interpretation.

For automatic control logic generation, such an input is however problematic. To overcome the challenge of dealing with bitmaps or vector-graphics, there are proposals to perform optical symbol and character recognition on the PDF files (Tan et al., 2016; Arroyo et al., 2016). Additionally, the DEXPI initiative[10] is working on a common XML P&ID file format based on ISO 15926[11]. The CAEX XML file format (IEC 62424[12]) as part of AutomationML

[1] https://bit.ly/2XWjmfk.
[2] https://www.isa.org/store/ansi/isa-50601-2007-functional-requirements-documentation-for-control-software-applications/116719.
[3] https://www.autodesk.com/.
[4] https://hexagonppm.com/.
[5] https://www.siemens.com/comos.
[6] https://www.bentley.com/.
[7] https://visio.microsoft.com/.
[8] https://www.edrawsoft.com/.
[9] https://www.lucidchart.com/.
[10] https://dexpi.org/.
[11] http://15926.org/.
[12] https://webstore.iec.ch/publication/25442.

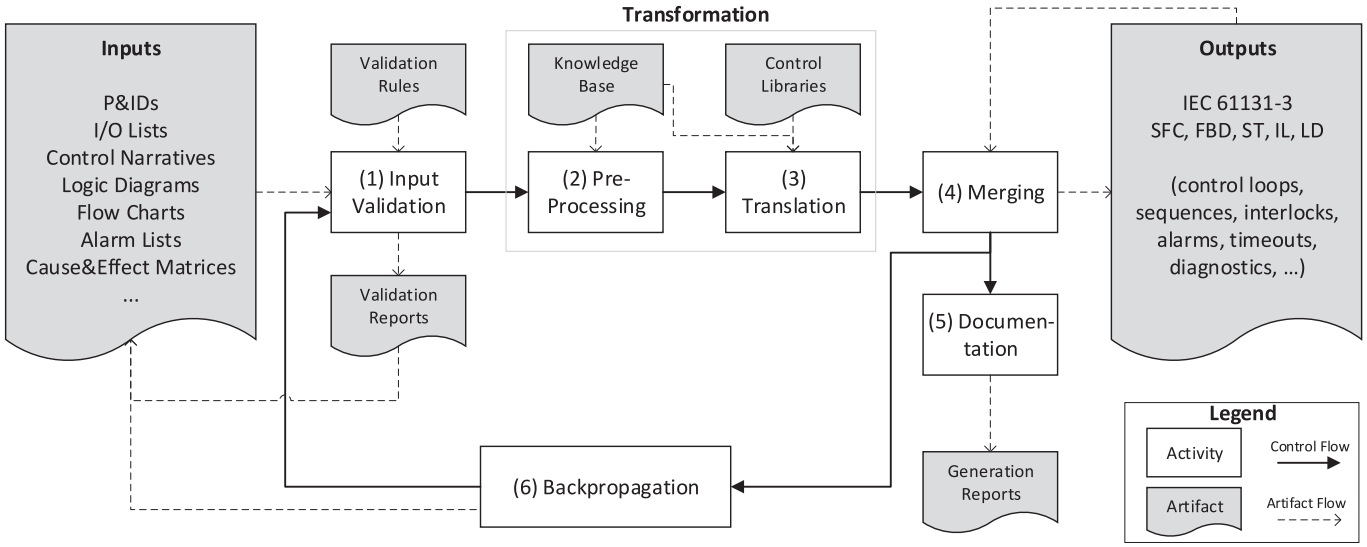

Fig. 1. Generic control logic generation process: six transformation steps with numerous inputs and outputs.

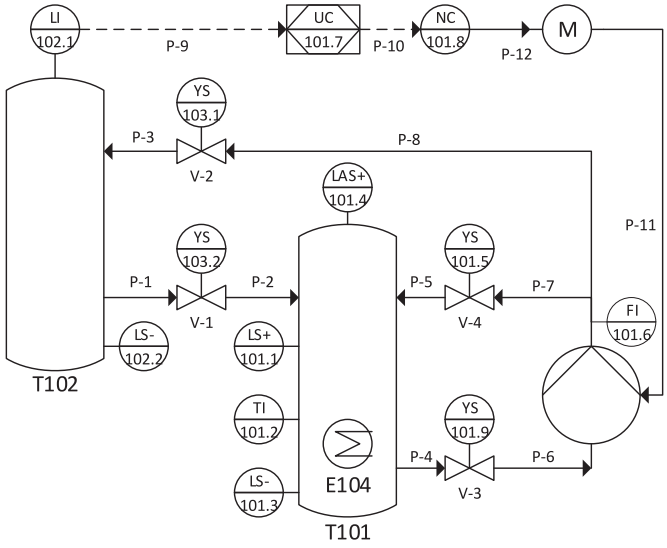

Fig. 2. Running Example: P&ID describing equipment and instruments.

Basic Point Data					I/O Data			HMI Data				Operating Data			
Tag Number	Service	Location	P&ID	...	Point Type	Signal Type	...	Range Min	Range Max	Engin. Unit	...	Alarm Low	Alarm High	Controller Algorithm	...
LS101.1	T101 Level	P-12001	1234		AI	4-20 mA		0	100	%		20	80	PID ideal	
TI101.2	T101 Temp.	P-12001	1234		AI	4-20 mA		15	95	°C		20	85	PID std	
LS101.3	T101 Level	P-12001	1234		AI	4-20 mA		0	100	%		20	80	PID ideal	
LAS101.4	T101 Level	P-12001	1234		AI	4-20 mA		0	100	%		20	80	PID ideal	
YS101.5	Inlet T101 Valve	P-12002	1234		DO	24 VDC		-	-	-		-	-	-	
FI101.6	Flowmeter P-8	P-12001	1234		PI	Pulse		0	100	m^3/h		-	-	-	
UC101.7	Pump Motor	P-12003	1234		DO	24 VDC		-	-	-		-	-	PID std	
NC101.8	Pump Motor	P-12003	1234		DO	24 VDC		-	-	-		-	-	PID std	
YS101.9	Outlet T101 Valve	P-12004	1234		DO	24 VDC		-	-	-		-	-	-	
LI102.1	T102 Level	P-12005	1234		AI	4-20 mA		0	100	%		10	90	PID ideal	
LS102.2	T102 Level	P-12005	1234		AI	4-20 mA		0	100	%		10	90	PID ideal	
YS103.1	Inlet T102 Valve	P-12006	1234		DO	24 VDC		0	100	%		-	-	-	
YS103.2	Inlet T101 Valve	P-12007	1234		DO	24 VDC		0	100	%		-	-	-	
E104	Heater T101	P-12005	1234		DO	24 VDC		0	100	%		-	-	PID std	

Fig. 3. Running Example: IO List describing signal properties.

(IEC 62714[13]), can be coupled with additional P&ID libraries (e.g., PandIX Schüller and Epple (2012)). However, there are hardly any commercial CAD tool P&ID exporters for CAEX (Drath and Ingebrigtsen, 2018) available so far.

I/O lists: Fig. 3 shows a simplified I/O list for the running example. A complete I/O list may include thousands of signals. Each row is dedicated to a single signal. There are many columns specifying properties of signals (tag name, revision, service description, location, value ranges, engineering units, alarm limits, typical assignment, wiring types, references to P&IDs, etc.). There are no industry standards for these properties, but companies usually use guidelines or internal standards. I/O lists are also known as tag lists, instrument or signal indices. Control engineers consider them as one of the most important inputs and they are usually provided in the form of large tables (Gutermuth, 2010).

In the context of control logic generation, I/O lists can be automatically imported into control engineering tools to create tags and I/O objects that then can be connected with function blocks to form the whole control logic. Control engineers then need write glue logic (e.g., recipes, interlocks) among them. In addition to I/O lists, there can be electrical plans according to IEC 60617 (Steinegger and Zoitl, 2012).

Control Narratives (Gutermuth, 2010): These prose writings describe the intended control algorithms in an informal way, usually in a Microsoft Word document or PDF file. They may refer to tag names from the I/O list and provide steps for the startup/shutdown of a plant or setpoints for regulatory control as well as alarm procedures. This notation is easy and fast to create. It also allows the engineering contractor to abstract from technical specifics of the target automation system (e.g., available typical libraries), and thus enables control engineers to optimize the logic (e.g., by mapping the intended logic to vendor-specific typicals).

Logic Diagrams: ISA5.2–1976 (Binary Control Logic Diagrams for Process Operations)[14] provides guidelines to create diagrams for binary interlock and sequencing systems for startup, operation, alarm, shutdown of equipment and processes. Input and output signals can be connected via AND/OR/NOT gates, as well as symbols to express timing relations. This notation is already similar to the actual control logic and therefore comparably easy to translate to it. Engineering contractors usually create logic diagrams with CAD tools or generic drawing tools. The structure and format of the

[13] https://webstore.iec.ch/publication/32339.

[14] https://goo.gl/VpfpD4.

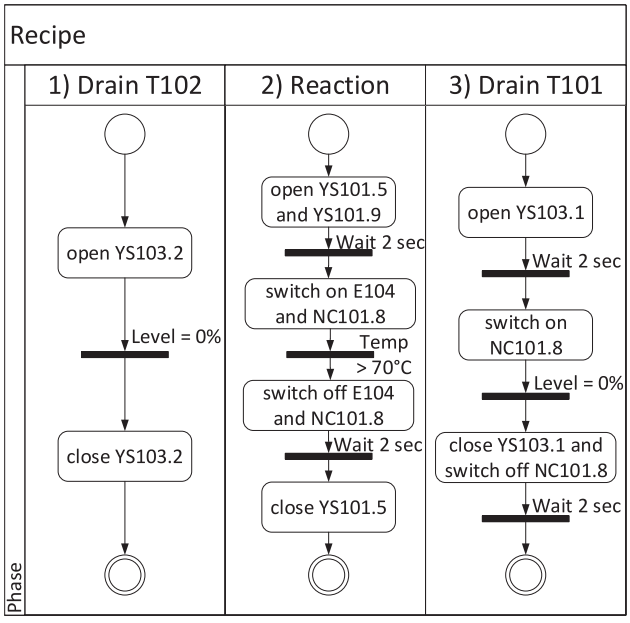

Fig. 4. Running Example: Recipe as flow charts.

used logic blocks may vary between different organizations, which can complicate the control logic generation.

Flow Charts: Sequential control flow or recipes can also be expressed with ordinary flow charts[15]. Fig. 4 shows a flow chart for the running example plant in a notation similar to UML activity diagrams. It consists of three phases. First, by opening valve 103.2, tank T102 is drained to transfer both substances in T101. Subsequently, the reaction starts in phase 2. The valves 101.5 and 101.9 open. Afterwards, the heater E104 and the pump driven by NC101.8 start. Once the temperature reaches 70 ° Celsius, pump and heater switch off and valve 101.5 closes. Finally, tank T101 is drained using the pump in phase 3.

A specific flow chart notation is GRAFCET ("GRAphe Fonctionnel de Commande Etapes/Transitions", IEC 60484[16]). It is a standardized graphical modeling language for designing controller behavior (Schumacher and Fay, 2014). GRAFCET was developed in France in the late 1970s, where it gained some industry adoption, and was also integrated into the curriculum of control practitioners in Germany. IEC 61131-3 Sequential Function Charts (SFC) are based on GRAFCET and are considered as one of its possible implementations.

UML or SysML[17]-based flow chart notations are rather uncommon for engineering contractors and control engineers (Thramboulidis, 2004; Friedrich and Vogel-Heuser, 2007; Schumacher and Fay, 2014). Researchers have previously developed several UML profiles for process automation (e.g., UML-PA (Katzke and Vogel-Heuser, 2005)), but they never became formal OMG specifications[18]. CODESYS UML[19] and TwinCAT3 UML[20] allow the specification of class and state diagrams, but are meant for control engineering programmers, not automation customers. ANSI/ISA-88[21] provides a standard notation for recipes in batch applications. While a mapping from UML state diagrams to IEC 61131-3 may be possible, an additional modeling tool adds complexity to the tool chain (Schumacher and Fay, 2014). Some engineering contractors therefore directly use SFCs to specify sequences.

Other Inputs: Additional inputs may assist control engineers in designing and implementing control logic.

- Alarm Lists (Steinegger and Zoitl, 2012): specify alarm limits and support writing control logic to handle alarms.
- Cause & Effect Matrices (C&Es) (Breyfogle III et al., 2000; Drath et al., 2006): provide a streamlined table to interconnect signals, which enables generating interlocking control code. They can be partially or fully provided from engineering contractors due to safety regulations (ISO 10418).
- Control Logic Libraries (Ljungkrantz and Akesson, 2007): containing pre-specified function blocks that capture higher-level domain knowledge.
- Legacy Control Code (Fay et al., 2001; Fay, 2003a): often available in migration (brownfield) projects. They can sometimes be translated into the control code of the target automation system.
- Domain Specific Inputs: these include for example System Control Diagrams (SCD) for NORSOK-compliant oil and gas facilities (Drath and Ingebrigtsen, 2018), or Scientific Apparatus Makers Association (SAMA) logic drawings[22] for power generation plants.

2.2. Transformations

Transforming the available inputs into valid control logic output requires a number of steps (Fig. 1) as detailed in the following.

Input Validation: Inputs for control logic generation require consistency and completeness checks (Barth et al., 2012). Human engineers specify these artifacts often in an incremental manner with multiple data exchanges between engineering contractor and automation provider. Therefore, the data may be incomplete and the different artifacts may be inconsistent. For example, the I/O list in Fig. 3 may be missing specified ranges or specific tag names might be inconsistent with the P&ID. The P&ID may contain unconnected pipes or missing properties of certain instruments. Sometimes the utilized CAD tools already provide consistency and completeness checks, but in other cases input validation by the automation provider is required. This is crucial for control logic generation, since the applied algorithms cannot reliably compensate for invalid inputs in contrast to a human engineer.

Pre-processing: Some input artifacts are subject to preprocessing, usually to simplify the later translation or to support different output formats. This may be implemented using in-place model transformations (i.e., endogenous transformations) or mapping to intermediate models (i.e., exogenous transformations) (Mens and Van Gorp, 2006). For example, the P&ID in Fig. 2 could be enhanced with flow paths to simplify code generation (Drath et al., 2006). A code generation approach can also map heterogeneous input data artifacts to a common intermediate model so that the code generation becomes less complicated (Steinegger and Zoitl, 2012; Arroyo et al., 2016). Unstructured data, such as Control Narratives, may require human labeling or annotations, so that text mining approaches can perform appropriate information retrieval. For example, a setpoint for the motor controller of the running example could be obtained from a narrative via text mining and then later be used in code generation.

Translation: Additional exogenous transformations translate input artifacts conforming to specific meta-models to output artifacts conforming to other meta-models (Mens and Van Gorp, 2006). In

[15] https://www.iso.org/standard/11955.html.
[16] https://webstore.iec.ch/publication/36840.
[17] https://sysml.org/.
[18] https://www.omg.org/spec/category/uml-profile.
[19] https://store.codesys.com/codesys-uml.html.
[20] https://www.beckhoff.com/english.asp?twincat/tf1910.htm.
[21] https://www.isa.org/isa88/.
[22] https://automationforum.co/what-is-sama-diagram/

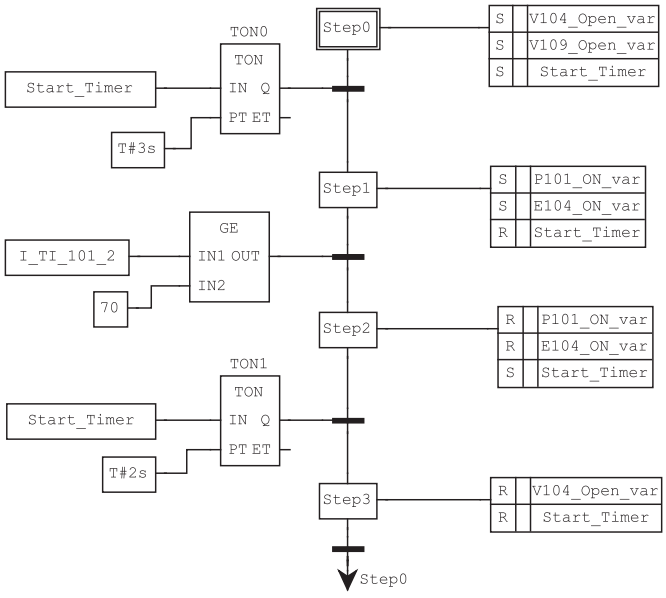

Fig. 5. Running Example: Sequential Function Chart (SFC).

case of logic diagrams or flow charts (cf. Fig. 4) being used, this involves an almost straightforward mapping to function block diagrams or sequential function charts (Fig. 5). In these cases, the underlying meta-models have a high similarity.

Other input artifacts, such as P&IDs and control narratives, require specially prepared code generation rules stored in a knowledge base (Steinegger et al., 2016). Such rules can encode domain knowledge, e.g. referring to the running example "if the filling level of a tank (e.g., T101) is too high, then close a valve (e.g., via YS103.2) connected to an inlet to the tank" (Drath et al., 2006). Rules may pertain different aspects of control logic, such as safety interlocks, sequences, diagnostic routines, or optimizations. Rules may be generic, domain-specific, or project-specific. Ideally, they are formulated in a domain-specific language familiar with the control engineers (Krausser et al., 2011; Grüner et al., 2014b).

Merging: The incremental nature of the engineering process, where partial input artifacts are exchanged multiple times between customers, engineering contractors, and automation providers in successively higher refinement and completion states, makes a one-shot translation impractical (Barth et al., 2012; Steinegger et al., 2016). In the running example, an engineering contractor may add new instrumentation to the P&ID and I/O list, e.g., a new pressure sensor for one of the pipes, or change specific tagnames or alarm limits. Blindly overwriting code from former generation iterations may be problematic in case control engineers have already enhanced or optimized the code manually. Therefore, code generation has to merge new additions from these refinements into IEC 61131-3 code already generated in previous translations. The model transformation first needs to analyze the formerly generated code and properly merge the new additions. It also needs to protect manually written code and report any inconsistencies or incompleteness back to the control engineer.

Documentation: A control engineer needs to understand the model transformation process to be able to detect systematic errors and to speed up root cause analysis in case of code generation errors. To support this requirement, a model transformation can, for example, create a report of the code generation process, that provides a detailed list of the applied code generation rules as well as any warnings and errors that may have occurred in the process. Ideally, such a report allows directly accessing the erroneous input artifacts, affected rules or the code generation output to check. Besides creating more confidence of the control engineer, such a documentation could also support testing and qualification processes.

Backpropagation: Manual additions and changes to the generated code may need to get propagated back to the input artifacts. This supports a shared understanding for the engineering contractor and control engineer and also serves documentation purposes (i.e., to always have an up-to-date design and specification). For the running example, changed tag names or alarm limits may need to get translated back from the control code to the P&IDs or I/O lists. This requirement suggests that the model transformation should support some form of bidirectionality. The level of bidirectionality is naturally higher for meta-models with higher similarity (e.g., flow charts and sequential function charts). In practice, such a return path from the control code back to the engineering contractor is rarely seen today because of the expensive manual overhead. This practice often leads to outdated and inconsistent planning documents over the course of the plant life-cycle.

To implement the described transformation steps, there is a plethora of method and tools for model-to-model (M2M) and model-to-text transformations (M2T). Transformations can be implemented with generic programming languages, such as Java, C++, C# in case the input artifacts are available as object-oriented models. XML files may directly be mapped to other XML files using XSLT/XQuery. The OMG's QVT (Query/View/Transformation) is a standardized set of languages for model transformations, including imperative (QVT-O) and declarative (QVT-R) transformation languages. The Eclipse ecosystem includes a number of M2M-transformations (Epsilon, Eclipse MMT) and M2T-transformations (Acceleo, JET, Xpand, Xtext), besides a common metamodel (Ecore) and tools to create graphical editors. There are extensible IDEs for control logic code available (e.g., 4DIAC[23] or Hardella[24] based on mbeddr[25]).

2.3. Outputs

Many commercial automation systems are still based on software development platforms mostly compliant to the IEC 61131-3 specification (Otto and Hellmann, 2009) for programmable controllers. This standard was originally published in 1993 and defines five programming languages: ladder diagrams (LD), function block diagrams (FBD), sequential function charts (SFC), structured text (ST) and instruction lists (IL). IEC 61,499 was published in 2005 as a partially backward-compatible enhancement of IEC 61131-3 with event-driven concepts, but has not gained widespread adoption in practice so far (Thramboulidis, 2013), although a number of academic approaches used it.

SFC: Fig. 5 shows an IEC 61131-3 SFC based on the recipe specification for the running example in Fig. 4. It includes four steps, which, in the tables on right-hand side, each trigger the setting (S) and resetting (R) of certain signals. The transitions between the steps (black bars) are triggered by function blocks on the left-hand side and involve simple timers and threshold conditions. This kind of control logic can be generated for example by mapping flow charts to SFCs or interpreting control narratives.

FBD: Fig. 6 shows an IEC 61131-3 FBD for a number of interlocks within the running example. On the left-hand side the diagram lists input signals, which trigger specific output signals on the right-hand side. In this example, inputs and outputs are connected by simple boolean (AND) and arithmetic (LE, less equal, GE, greater equal) function blocks as well as timers (TON/TOF).

ST: Fig. 7 shows an IEC 61131-3 ST code snippet from the running example. The syntax of ST is similar to the Pas-

[23] https://www.eclipse.org/4diac/i.
[24] https://github.com/Hardella/ide61131.
[25] http://mbeddr.com/.

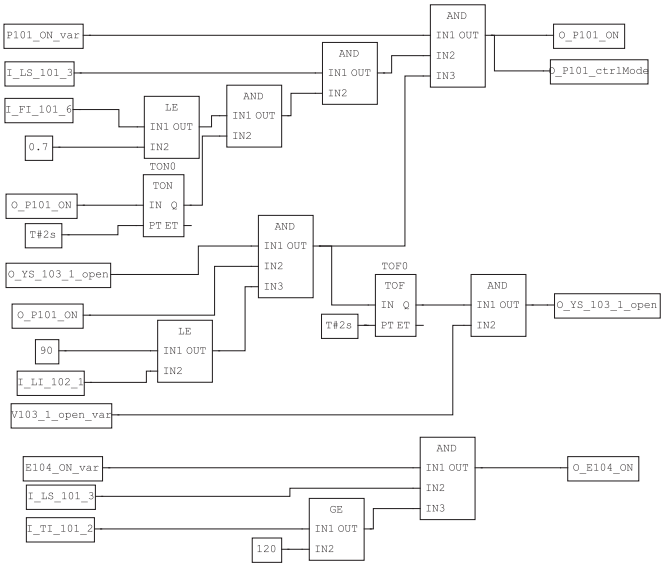

Fig. 6. Running Example: Function Block Diagram (FBD).

```
IF I_TI_101_2 >= 70 THEN (* temperature > 70 degrees *)
  P_101_ON_VAR := FALSE; (* stop pump *)
  E_104_ON_VAR := FALSE; (* stop heat exchanger *)
  CMD_TMR(IN := 2.0);    (* wait for 2 seconds *)
END_IF;
```

Fig. 7. Running Example: Structured Text (ST).

cal programming language. It supports variable assignments, if/case/for/while/repeat statements, as well as pointers and function calls. The language also provides constructs to define function blocks as higher-level programming constructs. ST is often preferred for more complex mathematical computations in control logic.

Other outputs: Ladder diagrams are particularly popular in North America, where many control engineers are not familiar with the IEC 61131-3 standard[26] while FBD, ST, and SFCs are more popular in other parts of the world. ILs have been labeled 'deprecated' in the latest release of the IEC 61131-3 specification. The PLCopen association published the first XML formats for IEC 61131-3 in 2005 Otto and Hellmann (2009), and has by now refined the specification as IEC 61131-10 PLCopen XML exchange format.

Today, control logic implementation is often based on instantiating template function blocks, so-called "typicals", e.g., for valves, motors, from a function block library. A code generator can largely automate this task by exploiting properties in I/O lists which often allow a one-to-one mapping of signals to typicals available in specific libraries, also known as *bulk engineering* (Gutermuth, 2010). Besides the initial variables for signals and the instantiated function blocks, the control logic code consists of a number of different segments for different purposes (Guttel et al., 2008). These include handling the nominal sequence (e.g., PID loops), alarms, startup/shutdown, asset monitoring, operations, diagnosis, interlocks, controller-to-controller communication and timeouts. Code generation should address as many of these segments as possible to lower the manual implementation effort.

Besides the control logic generation itself, a model transformation tool chain for typical customer input artifacts can also be exploited for other purposes. The generation of simulation models can support factory acceptance tests or plant modifications (Barth and Fay, 2013; Arroyo et al., 2016). Connections to HMIs may be generated and also process graphics can be partially generated out of P&IDs (Schmitz and Epple, 2007; Doherr et al., 2013). Intermediate models for code generation may also be used during the plant life cycle to create queries about the plant and to find root causes of plant-wide disturbances (Yim et al., 2006).

3. Classification framework

3.1. Methodology

The objective of the present study is to classify research approaches for IEC 61131-3 code generation from the viewpoint of a practicing control engineer. A detailed breakdown of each method's inputs and outputs helps to better understand the context the researchers assumed and the method's capabilities. The classification supports identifying patterns between the approaches and to assess their maturity levels.

We identified corresponding approaches using Google Scholar, IEEE Digital Library, and ACM Library using the search term "61131 AND control AND code AND (generation OR transformation OR automatic OR automated OR model-driven)" (Kitchenham and Charters, 2007). The search initially identified 5350 works. From the search results, we applied the following inclusion criteria to focus on our research objective and viewpoint:

- **IEC 61131-3 control code**: Included approaches use at least one of the five IEC 61131-3 languages as output to cover a large range of commercial and open source control systems. Excluded are approaches for example involving IEC 61499 (Hussain and Frey, 2006; Panjaitan and Frey, 2006; Thramboulidis and Buda, 2010; Thramboulidis, 2010), C-code, Simulink, or other PLC programming languages, because they are either less widespread in practice or tackle different application domains.
- **Code Generation**: Included approaches provide concepts and possible implementations aiming at actual code generation. Excluded are approaches that apply model transformation on automation requirements for other purposes (e.g., simulation (Barth and Fay, 2013; Arroyo et al., 2016), test case generation (Suresh et al., 2019), asset management (Schmidberger and Fay, 2007), loop performance assessment (Yim et al., 2006), HAZOP analysis (Fay et al., 2009)). While additional transformation targets are helpful to improve engineering, we limit the study to details of code generation as we aim at a deeper understanding of that process.
- **Uses Inputs typically used in Practice**: Included approaches rely on at least one of the inputs listed in Section 2, e.g., I/O lists, P&IDs, flow charts. This excludes approaches requiring for example Petri Nets (Cutts and Rattigan, 1992; Jörns et al., 1995; Frey, 2000; Music et al., 2005) or timed automata (Wang et al., 2009) as input, which are not used in practice today.
- **Journal/Conference publication**: Included are peer-reviewed approaches published in a Journal or conference proceedings. Approaches embedded into a commercial offering may be highly relevant for practitioners (e.g., Simulink PLC Coder, CODESYS UML (Tikhonov et al., 2014), TwinCAT3 UML), since they include commercial tooling and support. However, they are excluded from the following classification to not favor a particular commercial vendor or because of a lack of reliable information sources. Rivops AutoGen[27] generates control logic from an equipment list using Python scripts, and most commercial DCS products offer similar

[26] https://www.controleng.com/articles/iec-61131-3-whats-the-acceptance-rate-of-this-control-programming-standard/.

[27] http://www.rivops.com/autogen

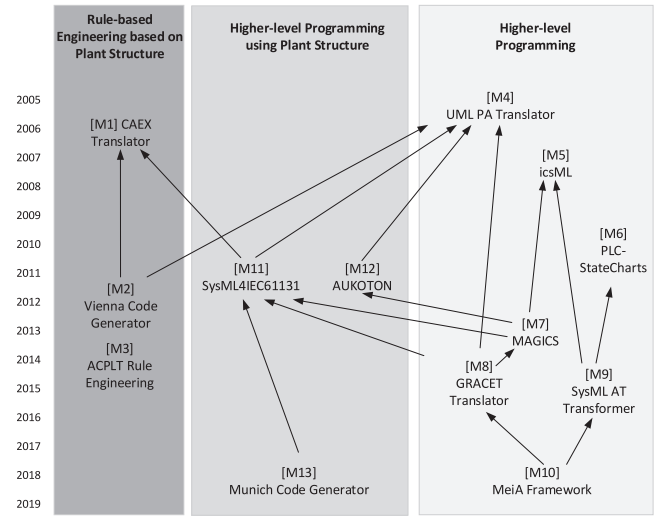

Fig. 8. Map of Approaches (2005–2018): Arrows indicate citations. The approaches can be divided in three classes depending on whether they use a plant structure as input and whether they use domain rules for code generation.

functionality (e.g., Simatic PCS7, Yokogawa Automation Design Suite). DEIF PLC Link[28] provides special modeling means for wind power applications and can produce PLCopen code. Actifsource Workbench[29] and 4DIAC[30] are examples for IDEs that can generate control code, but are also considered out of scope for this paper.

- **Published after 2004**: Included are papers less than 15 years old to keep the classification concise, so that approaches from the 1990ths are neglected. We assume that the conceptual contributions of these early approaches have influenced the design of the more recent approaches classified within this paper.

We applied a forward/backward reference search (Wohlin, 2014) on the identified approaches to increase literature coverage. We combined multiple papers from the same authors if they referred to the same approach (e.g., (Steinegger and Zoitl, 2012; Steinegger et al., 2016; 2017) or (Schumacher et al., 2013a; 2013b; Schumacher and Fay, 2014; Julius et al., 2017; 2019). Some of the excluded works will be discussed in Section 7.

3.2. Approaches Overview

Our survey contains 13 approaches from 2005 to 2018 (Fig. 8) within the inclusion criteria. In total they have been referenced by other works 679 times. All approaches originate from European researchers (only Lukman et al. (2013) has a co-author from the US). This corresponds to the observation from Section 2 that IEC 61131-3 is more popular in Europe.

The approaches can be coarsely divided into three classes:

1. **Rule-based Generation based on Plant Structure**: These approaches [M1,M2,M3] require control engineers to create a knowledge base of rules that are later automatically applied to formalized requirements and design documents to generate IEC 61131-3 code. The main benefits are the high degree of automation and the seamless integration into existing engineering workflows and tools. Drawbacks are the difficulty to construct robust, generic rule bases (i.e., applicable for many projects) and the need for formalized input artifacts so that the rules can be automatically applied. These approaches correlate with the application domain process automation.
2. **Higher-level Programming**: These approaches [M4,M5,M6,M7,M8,M9,M10] require control engineers themselves to formalize requirements and design in a higher-level modeling language (UML/SysML/GRAFCET) and then establish a mapping between the elements of the higher-level language and IEC 61131-3 constructs. Early Petri Net-based approaches would also be in this class. The main benefit is that the control engineers may be more productive using a high-level notation than directly IEC 61131-3. Drawbacks are the limited amount of automation and the challenge to learn and maintain two different notations. These approaches mainly originate from discrete manufacturing.
3. **Higher-level Programming using Plant Structure**: These hybrid approaches [M11,M12,M13] both take the plant structure into account *and* require control engineers to formalize additional requirements in higher-level languages, such as UML or SysML. They do not use rule-based generation. While these approaches may provide a richer starting point by utilizing the plant structure, they inherit the same benefits and drawbacks from the other higher-level programming approaches.

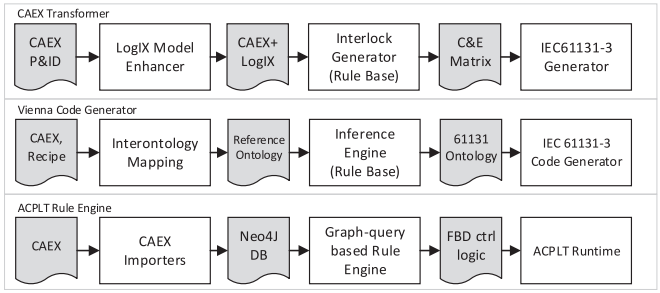

Fig. 9. Rule-based generation: high-level overview. The approaches extract information from plant topology models and apply rule engines to generate control logic.

Fig. 9 provides a high-level overview of the rule-based generation approaches:

[M1] **CAEX Transformer:** This approach takes a P&ID encoded as a CAEX XML as input and first enhances the file with the LogIX Model to group related equipment into lines. Afterwards it applies a set of interlocking rules on the resulting model and generates a C&E matrix, which can be reviewed and enhanced by control engineers. From the resulting C&E matrix, a generator directly creates IEC 61131-3 ST or FBD code.

[M2] **Vienna Code Generator:** The authors suggest to map various requirements and design documents into a reference ontology based on CAEX. Using pre-defined rules from a knowledge base, an inference engine generates instances of an IEC 61,131 ontology containing for example interlocking or diagnostic logic. A code generator processes this ontology and produces 61131-3 code.

[M3] **ACPLT Rule Engine:** An importer converts a CAEX plant structure XML file into a graph structure for the Neo4J[31] graph database. Using Cypher[32] queries on the graph database allows to identify certain patterns in the plant structure and directly generating FBD control logic or discovering flow paths. In this approach,

[28] https://www.deif.de/wind-power/technology/plc-link-code-generation
[29] http://www.actifsource.com/
[30] https://www.eclipse.org/4diac/
[31] https://neo4j.com/.
[32] https://neo4j.com/developer/cypher-query-language/.

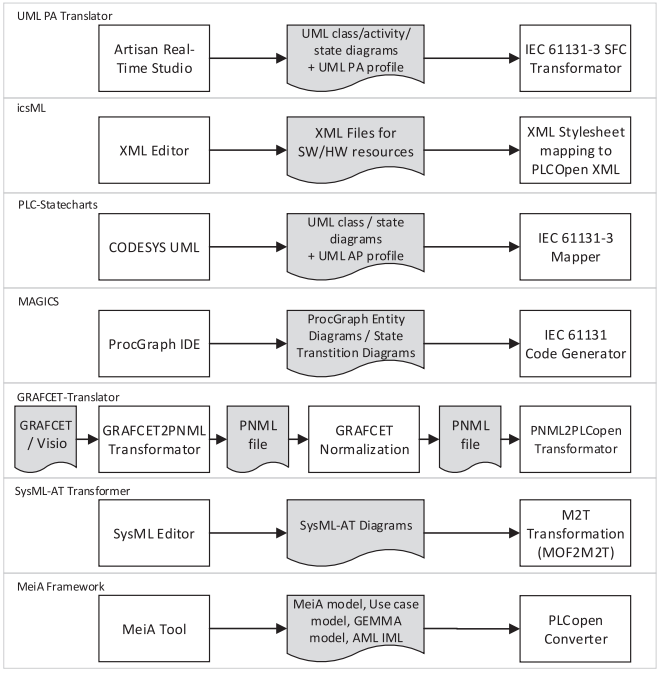

Fig. 10. Higher-level programming: high-level overview. The approaches require an upfront modeling in UML/SysML or similar notations, which are directly mapped to control logic.

the Cypher queries encode both domain specific rules and code generation templates.

Fig. 10 shows the higher-level programming approaches, which usually do not import any specifications, but require manual modeling:

[M4] **UML PA Translator:** A control engineer uses Artisan Real-Time Studio to model units of manufacturing equipment (e.g., a sorting facility) as UML classes, possibly enhanced by using the UML-PA profile for process automation. Each method of a class is further specified via a UML state diagram. The approach proposes a mapping of these models to IEC 61131 function blocks.

[M5] **icsML:** This approach requires engineers to create XML files modeling hardware and software resources, there is no CASE-tool support foreseen. Using XML stylesheets, the files are mapped into PLCopen XML and then into controller-specific languages.

[M6] **PLC-Statecharts:** This more formal approach is focused on providing adequate PLC semantics for UML statecharts. Eventually, the approach was implemented for the CODESYS framework, where a UML class diagram and state diagram editor are provided. The approach aims at a bi-directional mapping of the UML constructs and IEC 61131-3 code, so that ideally, the contol engineer does not have to deal with 61131-3 code at all.

[M7] **MAGICS:** Instead of using UML, the authors of this approach defined their own domain-specific modeling language called ProcGraph. It includes entity diagrams and state transition diagrams and features an IEC 61131 code generator. The whole approach is accompanied by an Eclipse-based modeling tool that allow graphical editing of the models.

[M8] **GRAFCET-Translator:** This approach starts from a GRACET (IEC 60484) specification of processes modeled using Microsoft Visio and a specific stencil set. A transformer coverts the Visio file into a Petri Net XML file (PNML), which is then normalized to certain restricted constructs and finally mapped into PLCopen XML using 28 transformation rules.

[M9] **SysML-AT Transformer:** The authors define the SysML-AT notation to extend SysML requirements and parametric diagrams.

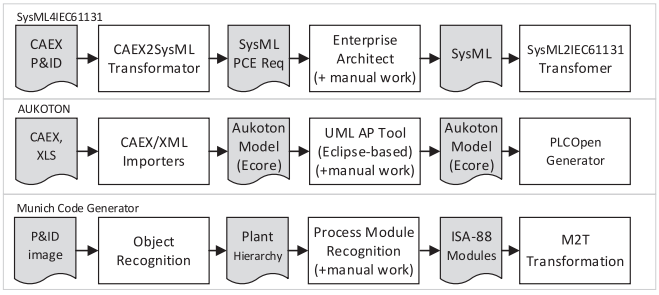

Fig. 11. Hybrid approaches: high-level overview. These approaches extract information from plant topology models into higher-level programming models, let the engineers complete the models, and then generate control logic.

The control engineer models a system using this notation and can trigger a model-to-text transformation to get IEC 61131-3 ST.

[M10] **MeiA Framework:** This Eclipse-based framework provides multiple domain-specific modeling languages for signals and phases, for use cases, and other constructs. While the framework supports different types of analyses, it can also translate instances of the model into PLCopen XML encoding IEC 61131-3 SFCs.

Finally, Fig. 11 shows three hybrid approaches that take the plant structure into account and introduce a form of higher-level programming. None of these approaches however uses rule-based engineering:

[M11] **SysML4IEC61131:** The authors propose a CAEX2SysML Transformator to import IEC 62424-compliant P&IDs into the UML/SysML modeling tool Enterprise Architect[33]. To reduce the gap between CAEX and SysML, the authors propose the SysML4IEC61131 profile. The resulting SysML requirement diagrams are semi-automatically refined with SysML block definition diagrams and can be transformed into PLCopenXML.

[M12] **AUKOTON:** The AUKOTON tooling takes P&IDs, IO lists and C&E matrices as input and maps them into domain-specific AUKOTON models implemented using Ecore[34]. Afterwards, control engineers can refine the models further, before a PLCOpen Generator processes them to produce IEC 61131-3 control logic.

[M13] **Munich Code Generator:** This approach performs an object recognition on P&IDs images. It tries to derive the plant hierachy from this operation and map certain patterns in the P&IDs to pre-specified ISA-88 compliant plant modules. A model-to-text transformation finally produces IEC 61131 code for these modules.

After briefly introducing each approach, the following will define detailed classification criteria to be able to compare the approaches better.

3.3. Classification Criteria

Fig. 12 shows our classification criteria for the IEC 61131-3 code generation approaches. These criteria are defined based on artifacts in practice (Section 2), an assumed comprehensive transformation process (Fig. 1), and discussions with domain experts for processes in oil and gas, chemical, and power generation applications. The input classification was inspired by Steinegger and Zoitl (2012), while the output classification was inspired by Guttel et al. (2008). The following subsections will briefly explain each criterion.

3.2.1. General Properties

The *General Properties* cover high-level information to understand the context and maturity of each approach. The *Publication Venue* states in which journal or conference the authors published

[33] https://www.sparxsystems.de/.
[34] https://www.eclipse.org/modeling/emf/.

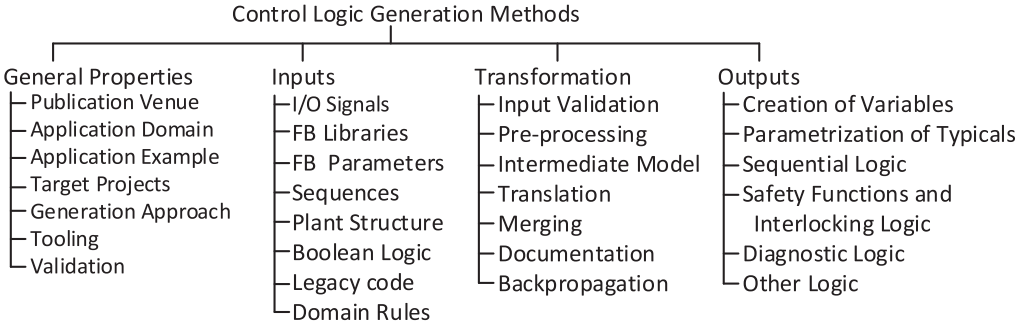

Fig. 12. Classification criteria for automated control logic generation approaches: inputs available in practice, transformation steps, types of output.

the main approach, where journal publications may indicate a higher maturity. The *Application Domain* distinguishes between discrete manufacturing and process automation, because different inputs and constraints may be available in a given domain. A short description of each approach's main *Application Example* helps the reader to better understand an approach's original context and the maturity of each approach.

The *Generation Approach* summarizes the main translation idea. The *Tooling* summary lists accompanying software tools including external libraries and framework. Finally, the *Validation* categorizes how the authors evaluated their approach. This may potentially span from minimal examples executed by the authors themselves as a proof-of-concept up to multiple, heterogeneous real-world case studies carried out in independent studies to demonstrate actual cost and benefits. Ideally, the authors should characterize how much manual work an approach saved or how large the fraction of generated code was in the analyzed cases.

3.2.2. Inputs

The second dimension of our classification framework are the *inputs*. These can be considered as the requirements for the control code. As there is no industry standard, we used different sources for the input classification. Section 2 has surveyed the available input artifacts in practice. Hästbacka et al. (2011) consider P&IDs and spreadlists (I/O lists, C&E lists) as inputs. Steinegger and Zoitl (2012) provided a list of different artifacts potentially involved in code generation, which included P&IDs, recipes, electrical plans, alarm lists, and logic diagrams. We also analyzed ABB-internal guidelines for inputs from customers.

Our classification abstracts from particular artifacts and rather distinguishes the approaches based the content in these artifacts. This avoids overlaps (e.g., I/O signal references both in I/O lists and P&IDs) as well as multiple kinds of artifacts for similar content (e.g., P&ID, SCD, PFD modeling the plant structure). Depending on the particular application domain, an approach may be able to generate more code if it takes more inputs into account.

The first kind of input are *I/O signals*, which may be specified in I/O lists, P&IDs, control narratives or other artifacts. They later form input and output signal references in the IEC 61131-3 control code. *Function Block (FB) Libraries* with a list of typicals may be used to directly include higher-level domain concepts into the generated IEC 61131-3 code, which may potentially ease code generation significantly. Using type attributes for I/O signals from an I/O list, matching typicals can directly be instantiated by the code generation, e.g., function blocks for motors or valves.

FB Parameters may be available for example from detailed I/O lists or control narratives, for example alarm limits, setpoints, or tuning parameters. If available, the code generation can directly copy them into the control logic. Typicals in control engineering libraries can potential have more than 100 input and output parameters, although many of these are often kept on default values. If not directly available, a code generator could potentially synthesize FB parameters using domain rules.

Sequences cover any kind of steps describing the control flow. These may for example refer to recipes in batch application or startup/shutdown or emergency procedures. When provided by a customer or engineering contractor, sequences may be specified for example in control narrative, flow charts, or logic diagrams. *Boolean Logic* may directly describe state-based control or interlocks. Specifications may be available as logic diagrams, tables, control narratives, or C&E Matrices.

Approaches may consider formal specifications of the *Plant Structure* to generate control logic based on the encoded relationships between equipment and instruments. This may support generation of sequencing logic, interlocking logic, or diagnostic functions. Approaches could potentially exploit the topological structure for synthesizing start-up sequences if the material and information flow can be extracted unambiguously. Notations for plant structures include P&ID, SCD, PFD, and building plans and with different levels of formalization (e.g., bitmap, vector-based drawing, object-oriented model with formal semantics).

Legacy code may be available as input for a code generator. According to ARC (Forbes, 2018), only 35 percent of yearly distributed control system projects revenues are for new constructions (greenfield), while the remaining 65 percent are for replacements, upgrades, or expansions (brownfield). Even in greenfield projects, code from similar plants or factories may be available, which could be partially reused. In brownfield projects, also legacy control logic code, potentially in a proprietary programming language may be available. If an approach is able to deal with existing code, it can potentially also better support iterative development with multiple generations corresponding to the incremental availability of customer artifacts. In case of replacements, this may involve translation of legacy control code (Fay, 2003b), while in case of upgrades, this may involve merging new code into existing code (Steinegger et al., 2016).

Finally, *Domain Rules* may allow a code generator to synthesize control logic. These rules may encode interlocking concepts, naming conventions, safety standards, PID parameters, or other concepts. They form a knowledge base, which is potentially applicable across different projects and essentially turn the code generator into an expert system for control logic engineering.

3.2.3. Transformation

The third dimension of the classification pertains the *Transformation* process. The selected criteria map to the generic transformation process sketched in Section 2, which is based on a general description of engineering processes (Gutermuth, 2010) as well as discussions with several domain experts.

Input Validation assesses whether an approach checks for completeness, validity, and consistency within and among the different input artifacts. This is more relevant if an approach integrates

Table 1
Breakdown of different control logic segments: estimated fraction of source code and efforts.

Segment of Control Logic	Description	Estimated fraction of total control code	Estimated implemen-tation effort
Nominal Sequence	1) Instantiation of Typicals 2) Parametrization of Typicals 3) Interconnection of Typicals 4) Definition of sequence control	30%	60%
Interlocks	Protection functions combining two or more types of equipment	20%	10%
Tags / I/O objects	References to signals, IO objects	15%	1%
Alarms	Hardware alarms for range violations, process alarms for process range violations	10%	1%
Start-Up	Initialization Sequences, may require operator interaction	5%	7%
Diagnosis	For example Profibus DP Slave Device Diagnostics, EtherCAT Diagnostics, etc.	5%	5%
Communication	Peer to peer communication between controllers.	5%	5%
Restart / Reset	Manual reset, or reset after energy failure, safe stop	2%	2%
Asset Monitoring	Detect abnormal behavior, react upon error signals or proactively search	2%	1%
Time-out	Monitor whether time limits are exceeded	1%	1%
Operation and Monitoring	Automated population of OPC Servers for HMI stations	n/a	n/a
	SUM	100%	100%

multiple artifacts with possibly overlapping contents. *Pre-processing* ranges from manual interpretation and formalization of inputs to intermediate model transformations that translate the inputs into other formats for rule application and/or simplified code generation.

Most approaches employ an *intermediate model* that is either manually specified or automatically created. The actual *translation* converts this intermediate model to IEC 61131-3 code (e.g., M2T transformation), for example by performing a direct mapping of concepts (e.g., in case of logic diagrams) or by applying domains rules. In most practical cases, generated code would be subject to *merging* with existing code, which must avoid overwriting manually specified code and creating duplicates.

Providing an appropriate *documentation* of the transformation process (e.g., with indication of applied rules or issues in the input artifacts) is essential for practical adoption and acceptance by control engineers. Finally, *propagation* of manual changes or additions to the control logic back to the inputs is an important feature to keep the plant specification up to date (Jamro and Rzonca, 2018; Julius et al., 2019).

3.2.4. Outputs

Our classification framework considers the *Outputs* of the code generation both in terms of the format (i.e., which IEC 61131-3 language) and the contents (i.e., what parts of control logic. Guttel et al. (2008) provided a detailed breakdown of the different aspects of PLC functions, categorizing more than 20 different functions. We deemed this categorization too detailed for our classification scheme, and thus selected a number of functions and let multiple domain experts from our company rank their estimated implementation effort. Table 1 shows the consolidated result. While the results are difficult to generalize and may vary a lot between different projects, the numbers at least give notion of higher importance, which is sufficient for the creation of a classification scheme.

Table 1 shows that the nominal sequence requires the most efforts implementation wise, although the amount of control code may be low. Especially interconnecting typicals and definition of sequence control requires efforts for interpreting different inputs and designing an appropriate control strategy. Interlocks represent a significant portion of the control logic, but are comparably easy to implement, since they may either be already fully specified by customers due to safety regulations or require the manual application of simple domain rules. Other types of control logic may make up only smaller parts of the overall code.

We included the *creation of variables / instantiation of typicals* as first criteria into our classification framework. Each approach supports this in some sense as it is required for IEC 61131-3 code. The *parameterization of typicals* is another criterion normally requiring the use of a FB library. In our classification, it also includes alarms. We combined different kinds of *sequential logic* into a single criterion, which comprises start-up, shutdown, as well as recipes (called "Definition of sequence control" in Guttel et al. (2008)).

Safety functions and interlocking logic sum up all kinds of boolean logic, such as emergency procedures and resets. *Diagnostic logic* may gather data about the system and field devices and may inform a human operator. Finally, we summed up all other types of logic from Table 1 in *Other Logic*. For each of these classification criteria, we identified whether it was explicitly supported by the approach and which IEC 61131-3 language was used. It should be noted that using IEC 61131-3 code generation, each approach could potentially support each type of output, but the output classification helps to better understand on which particular content an approach focused.

4. Comparison of approaches

Using the classification from Fig. 12, the following compares the approaches. Each subsection deals with one of the four parts of the classification and uses a table to provide an overview to what extent the approaches meet the criteria.

4.1. Comparison of General Properties

Table 2 lists the approaches' general properties. The *publication venues* are mainly journals and conferences on software/system engineering as well as automation and manufacturing.

Since IEC 61131-3 logic is not specific for any application, each approach is applicable in any *application domain*. However, most of the surveyed approaches originate either from a discrete manufacturing setting or a process automation setting, which leads to different inputs, constraints, and assumptions. An indicator is the main *application example* of each approach. These are predominantly small-scale lab automation systems or didactic plants. MAGICS [M7] and Vienna Code Generator [M2] use application examples based on sub-segments of real-world plants.

Table 2
General Comparison: most approaches originate either from discrete manufacturing or process automation. There are diverse generation approaches, validations are often performed using simple examples.

Approach	Venue	Application Domain	Application Example	Generation Approach	Tooling	Validation
[M1] (CAEX Transformer)	IEEE Conf. on Computer Aided Control Systems Design	Process Automation (example for chemical)	Generic plant unit (1 tank, 1 motor, 3 valves)	Matching rules from a knowledge base	Tool prototype processing CAEX files	Authors applied own approach, single sample case
[M2] (Vienna Code Generator)	IEEE Int. Conf. On Emerging Technologies and Factory Automation	Process Automation (examples for hot rolling mill, chemical)	SMS Simag AG Hot Rolling Mill, FESTO Compact Workstation didactic plant (2 tanks, 4 valves, 1 pump)	Translation of ISA-88 recipes, generation based on safety rules	Unnamed tool prototype (logi.CAD, Apache Jena, Protege, JAXB)	Authors applied own approach, two industryderived case studies
[M3] (ACPLT Rule Engineering)	IEEE International Conference on Industrial Informatics	Process Automation (example for chemical)	Minimal plant (1 pump, 1 valve)	Matching rules (defined as graph database queries)	Unnamed tool prototype based on Neo4J, Cypher	Authors applied own approach, single sample case
[M4] (UML PA Translator)	IEEE Int. Conf. on Control and Automation	Discrete Manufacturing / Process Automation	Lab prototype for a sorting facility	Mapping UML to IEC 61131- 3	Tool prototype using Artisan Realtime Studio	Authors applied own approach, single sample case, discussed with industry experts
[M5] (icsML)	Springer Journal on Advanced Manufacturing Technology	Discrete Automation	No specific application example, only generic XML templates	Translation of a selfdefined XML-schema to 61131-3	XML schemas and stylesheets provided	Authors sketched mapping to two commercial IEC 61131-3 runtimes
[M6] (PLC-Statecharts)	Elsevier IFAC Proceedings	Discrete Manufacturing	Lab example involving error handling routine for a pneumatic cylinder	Translation of UML statecharts to IEC 61131-3 ST	CODESYS UML	Student experiment with 30 participants, informal reasoning
[M7] (MAGICS)	Elsevier Journal of Control Engineering Practice	Process Automation	Grinding Titanium Dioxide in Slovenia (50 devices, 400 signals)	Translation of an extended finite state machine + ST, manually derived from P&ID	ProcGraph Eclipse tooling (EMF / GMF / oAW / Mitsubishi GX IEC Developer, CODESYS)	Authors applied own approach, single industryderived case study (68% of code generated)
[M8] (GRAFCET-Translator)	Elsevier Journal of Control Engineering Practice	Discrete Manufacturing / Process Automation	Plant for checking electromech. parts (conveyor belt, rotary table,100 signals)	28 transformation rules from GRACET to 61131-3	GRAFCET Editor & Translator	Authors applied own approach, single industryderived case study
[M9] (SysML-AT Transformer)	Elsevier Journal on Mechatronics	Discrete Manufacturing	Laboratory Pick & Place Unit (stamp, crane, stack, sorter)	Model-to-text transformation using MOFM2T	Unnamed tool prototype for SysML-AT	Student experiment with 36 participants, hypotheses testing
[M10] (MeiA Framework)	IEEE Transactions on Automation Science and Engineering	Discrete Manufacturing	SMC FMS-200 training Mechatronic Station (Conveyor + Actuators)	PLCopen converter framework (M2T transformation)	Eclipse-based MeiA tooling	Developers applied the approach in 15 projects with about 50 I/O signals each
[M11] (SysML4IEC61131)	SciRes Journal on Software Engineering and Applications	Discrete Manufacturing / Process Automation	FESTO Modular ProductionSystem	Translation of SysML block diagrams to 61131-3 via SysML Profile	SysML4IEC61131 Profile, SysML2IEC61131 Translator, MARTE Profile(not implemented)	No implementation, only conceptual approach described
[M12] (AUKOTON)	Elsevier Journal of Systems and Software	Process Automation (example for chemical)	Lab example for a water treatment plant segment	Translation of P&IDs, IO Lists, C&E matrices into an intermediate UML model, IEC61131 FBD	UML AP Tool (Eclipse, Topcased, SmartQVT), includes importers for CAEX and IO lists	Authors applied own approach, single sample case, discussed with industry experts
[M13] (Munich Code Generator)	IEEE International Systems Engineering Symposium	Process Automation	myJoghurt lab-scale CPPS demonstrator (pump, filling, heating, mixing units)	P&ID image recognition, module identification and matching from library, then M2T transformation	C++ prototype using SQLite, RI-CAD, CODESYS, TIAPortal	Authors applied own approach, single sample case

Legend: ■ Rule-based Engineering approaches, ■ Higher-level Programming approaches, ■ Higher-level Programming using Plant Structure

The *generation approaches* are either based on matching domain-specific rules on specifications provided as inputs, or mapping a higher-level notation into IEC 61131-3. For the latter, several approaches employ a model-to-text transformation. The Munich Code Generator [M13] aims at a generation by a mapping to pre-specified modules, although it also foresees manually specifying the control logic for modules that are not yet provided in a library.

The *tooling* is tightly connected to the generation approach. Several approaches use UML modeling tools and extend them with prototypical enhancements. AUKOTON [M12], MAGICS [M7], and MeiA [M10] make extensive use of the Eclipse modeling framework and provide Ecore models besides graphical editors and model-to-text transformations. Only CODESYS UML from PLC-Statecharts [M16] is available as a commercial offering by the company 3S. All other tools remain in a proof-of-concept maturity and are not openly available for independent testing.

For *validation*, most authors apply their own approach on a single example. This provides an initial proof-of-concept validation, but does not qualify an approach for practical use. Whether practitioners could repeat a published approach independently or whether the approach runs into scalability, maintenance, or robustness issues remains unclear. For PLC-statecharts (Witsch et al., 2010) and SysML-AT Transformer (Vogel-Heuser et al., 2014), the researchers carried out experiments with around 30 students each and showed that a higher-level programming language can lead to higher productivity. The MeiA approach (Alvarez et al., 2018) has been applied by a number of developers in 15 simple, experimental projects (approx. 50 I/O signals), but the documented validation results are difficult to interpret.

Researchers have tested rule-based generation approaches only using small rule bases that cover minimal aspects, but are insufficient for practical application. Large rule bases may lead to inconsistencies, overlaps, and scalability issues. Larger experiments, pilot projects, and applications to entire projects would be needed to facilitate technology transfer. The code generation as such is seldom measured or precisely characterized. Only the MAGICS approach (Lukman et al., 2013) states that 80 KByte of executable control logic had been generated from higher-level constructs, which was 68 percent of the overall code.

4.2. Comparison of Inputs

The surveyed approaches based their code generation on a range of different inputs (Table 3).

The approaches take information to define *I/O signals* and control variables either from natural language specifications or use structured customer inputs, such as P&IDs. Higher-level programming approaches require the control engineer to model the required signal references, for example using class diagrams. Rule-based engineering approaches as well as hybrid approaches instead derive the signal references from CAEX representations of P&IDs diagrams. Only the AUKOTON [M12] explicitly imports Excel-based I/O lists which are often available in practice. SysML4IEC61131 [M11] uses a combination of transforming a CAEX-based P&ID to SysML and then manually adding requirements to the model.

Function block libraries are another important ingredient for code generation. The SysML4IEC61131 approach [M11] intends to integrate different kinds of unspecified FB libraries. AUKOTON [M12] uses a self-defined DCS library, and the Vienna Code Generator [M2] created two libraries with one and two function blocks respectively, specifically for diagnostics. The Munich Code Generator [M13] relies on a library of pre-defined process modules according to ISA-88, but details remain to be specified in a given project. Other approaches do not deal with function block libraries, although integration should be fairly easy. The used libraries may determine on what abstraction level the control engineer and the code generation can work. Highly aggregated function blocks as in [M13] may reduce the required manual work significantly, since it only requires to connect a few large blocks, instead of creating a detailed low level specification. However, function block libraries are often domain-specific or even project-specific.

Function block parameters may pertain configuration parameters, alarm ranges, set points, and other values, which could be derived from information provided by the customer or also based on experience in similar projects. Most approaches seem to neglect that these parameters could be derived from customer specifications or using domain rules. In AUKOTON [M12], ranges and setpoints can be taken from the I/O list, while the Vienna code generator provides a concept to retrieve parameters using electrical plans. Higher-level programming approaches require the user to set these parameters in UML classes that are later mapped to function blocks.

Multiple approaches use notations based on UML statecharts to model *sequences* due to the similarities with IEC 61131-3 SFCs. The UML PA Translator [M4] uses UML state charts enhanced with the UML PA profile and maps them with included variables to SFCs. icsML [M5] provides an XML schema for software resources that can express sequences. PLC-statecharts [M6] enhance UML statecharts with user-defined priorities for transitions. Due to IEC 61131-3's cyclic nature, these statecharts cannot be triggered by events, and the approach introduces behavioral semantics as in polling real-time systems. The Vienna code generator [M2] may use recipes being available in a notation conforming to ISA-88, while the MAGICS approach [M7] provides self-defined state transition diagrams and state dependency diagrams that the control engineer models.

The GRACET translator [M8] is centered around the GRAFCET notation (IEC 604848) for specifying sequences. The approach provides formal semantics for GRAFCET enclosing steps, forcing orders and time constraints, and is thus able to utilize the full GRAFCET notation in code generation. A number of student experiments (Vogel-Heuser, 2014) suggest that UML-based modeling can be more productive than directly writing IEC 61131-3 code, if well integrated into an IEC 61131-3 development environment with round-trip functionality. Challenges for adopting such approaches in industry are developer training, work processes, and integration with legacy code. Approaches processing P&IDs often do not deal with sequential specifications explicitly, but instead focus on logic directly derived from the P&ID.

Boolean logic may be used for interlocks, emergency shutdowns, but also model sequences. While logic diagrams are often available in practice as requirements, only the Vienna Code Generator [M2] discusses processing them, but provides no implementation. AUKOTON [M12] can import C&E matrices that express interlocks as a special kind of boolean logic and turn them into control logic. Tools are available to transform C&E matrices directly into IEC 61131-3 control logic [M1].

Approaches in process automation usually take the *plant structure* or topology into account as a P&ID [M1,M2,M3,M11,M12,M13]. The CAEX Transformer [M1] uses the connections between equipment an instruments in P&IDs to apply rules to generate interlocking logic. The SysML4IEC61131 approach [M11] extracts process control engineering (PCE) requests, control functions, and loops from P&IDs and maps them to SysML requirements diagrams. AUKOTON [M12] follows a similar approach and maps PCE requests in P&IDs to UML class diagrams stereotyped as Automation Requirements. The Vienna Code Generator [M2] uses CAEX and PandIX as inputs and extracts PCE requests, signal connections needed for interlocks, and PCE control functions to group multiple signals. Schüller and Epple (2012) showed prototypically how PandIX P&IDs could be extracted from a native XML-export of COMOS P&ID.

MAGICS [M7] in turn requires the control engineer to manually interpret the P&ID and formulate the contents in the ProcGraph notation. The Munich Code Generator imports P&IDs as SVG[35] files to perform image recognition in order to identify modules that can be mapped to higher-level function blocks.

Higher-level programming approaches often originate from discrete manufacturing scenarios without an explicitly modeled plant structure. They foresee modeling structures using UML class diagrams. In this context, Grüner et al. (2014a) proposed so-called product flow diagrams (PFDs) as a notation analog to P&IDs but for discrete manufacturing scenarios.

Except for the Vienna code generator [M2], none of the approaches takes existing, potentially proprietary *legacy control code* into account for the code generation. This could feed a direct translation into low-level IEC 61131-3 control logic (Fay, 2003a), but could also require mapping existing higher-level function block libraries. There are numerous commercial offerings to migrate control logic between control systems of different vendors (e.g., migrations to Emerson Delta V[36] to Siemens SIMATIC PCS7[37] to ABB 800xA[38]). Legacy code could also be available as IEC 61131-3 in a migration project, then the goal of code generation would be to seamlessly integrate with the existing code. [M2] considers this case explicitly, as it adds interlocking logic to already existing

[35] https://www.w3.org/TR/SVG2/.
[36] https://bit.ly/2U6HxSx.
[37] https://sie.ag/2VAZNoE.
[38] https://bit.ly/2UPzWvY.

Table 3
Comparison of the Inputs of each approach: most approaches support only specific inputs, some of them require manual modeling of customer requirements.

Approach	I/O Signals	FB Libraries	FB Parameters	Sequences	Boolean Logic	Plant Structure	Legacy code	Domain Rules
[M1] (CAEX Transformer)	Instruments from CAEX (IEC 62424)	n/a	n/a	n/a	n/a	P&ID in CAEX (IEC 62424) + self-defined logistics model called LogIX	n/a	Interlocking rules (XML)
[M2] (Vienna Code Generator)	Instruments from CAEX, elec. plans (EN 60617) + IEC 81346	61131-3 lib for Profibus DP / EtherCAT Diagnostics	Instruments from CAEX, elec. plans (EN 60617) + IEC 81,346	Recipes (ISA-88)*	Logic Diagrams (ISAS5.2), C&E Diagrams (ISO 10418)	P&ID in CAEX (IEC 62424) + PandIX / IEC 62424 based Ecore Model	PLCopen (XML) exported from Rockwell RSLogix 5000	Interlocking + diagnotic rules (SPARQL)
[M3] (ACPLT Rule Engineering)	P&ID in CAEX (IEC62424) + PandIX	n/a	n/a	n/a	n/a	P&ID in CAEX (IEC 62424) + PandIX	n/a	Interlocking rules (Cypher)
[M4] (UML PA Translator)	Attributes in Class Diagrams	n/a	n/a	Activity Diagrams, State Charts	n/a	n/a	n/a	n/a
[M5] (icsML)	XML schema for software resources	n/a	n/a	XML schema for software resources	XML schema for software resources	n/a (XML schema for controllers and I/O devices)	n/a	n/a
[M6] (PLC-Statecharts)	Attributes in Class Diagrams	n/a	n/a	PLC-statecharts	n/a	n/a	n/a	n/a
[M7] (MAGICS)	Signal List (manually processed)*	n/a	Signal List (manually included)	ProcGraph State Transition Diagrams, Functional Specification	Safety Requirements	P&ID diagram (manually processed)	n/a	n/a
[M8] (GRAFCET-Translator)	Variables in Grafcet (IEC 60848)	n/a	n/a	Grafcet (IEC 60848), Control Interpreted Petri Net	Grafcet (IEC 60848), Control Interpreted Petri Net	n/a	n/a	n/a
[M9] (SysML-AT Transformer)	SysML Parametric Diagram + SysML AT Profile	n/a	SysML Parametric Diagram + SysML AT Profile	SysML Parametric Diagram + SysML AT Profile	n/a	n/a	n/a	n/a
[M10] (MeiA Framework)	Part of MeiA model	n/a	Part of MeiA model	Design Organization Units (DOU) derived from MeiA model	Design Organization Units (DOU) derived from MeiA model	n/a	n/a	n/a
[M11] (SysML4IEC61131)	Instruments from CAEX (IEC 62424), SysML Requirement Diagram	Undefined standard / domain-/ projectspecific libraries (PLCopenXML)	SysML Requirement Diagram	n/a	SysML Requirement Diagrams for interlocks / safety and security requirements	P&ID in CAEX format (IEC62424)	n/a	n/a
[M12] (AUKOTON)	I/O List (Excel)	AUKOTON DCS Library	I/O List (Excel)	n/a	C&E Matrix (Excel)	P&ID in CAEX format (IEC62424)	n/a	n/a
[M13] (Munich Code Generator)	Instruments recognized from P&ID	Self-defined Process Module library with ISA-88 compliant modules	n/a	From module references recognized in P&ID	n/a	P&ID as SVG file	n/a	n/a

Legend: ■ Rule-based Engineering approaches, ■ Higher-level Programming approaches, ■ Higher-level Programming using Plant Structure

PLC code and uses an interlock redundancy elimination mechanism based on the Z3 theorem prover.

Three approaches use *Domain rules* in order to generate interlocking or diagnostic logic ([M1,M2,M3]). Other areas for applying domain rules (e.g., generating alarms, startup/shutdown sequences, recipes, etc.) may be more complicated and remain unexplored. The CAEX Translator [M1] directly encodes these rules in XML (Schmidberger and Fay, 2007), while the Vienna code generator [M2] uses SPARQL queries, and the ACPLT Rule Engineering [M3] executes Cypher queries on Neo4J graph databases. Krausser et al. (2011) proposed to use directly IEC 61131-3 to formulate such rules, so that control engineers can work with a familiar notation.

In summary, UML/SysML-based approaches put much of the requirements formalization workload on the control engineer, who models requirements and design in UML or SysML. In other approaches, which are mostly from process automation, the control engineer may already work with semi-formal or formalized requirements and derive a design and implementation.

Regarding completeness of input artifacts, the Vienna code generator [M2] provides the most holistic view, addressing all of the input elements of our classification. However, the approach does not provide tooling to process these inputs automatically, but instead assumes that an appropriate CAEX model is already available from these artifacts.

4.3. Comparison of Transformations

Table 4 compares different aspects of the model transformations. None of the surveyed approaches explicitly deals with *input validations*, except icsML [M5] which executes syntax checks on the used XML models. Several approaches assume a valid, i.e., consistent and syntactically correct CAEX files as input. Other approaches require a formalization of unstructured inputs by the control engineer, which then implicitly includes an input validation performed during manual interpretation. Syntax, semantic, and plausibility checks on individual artifacts could be tackled by the CAD tools creating them. Consistency checks between different artifacts may be facilitated by mapping them into a common representation, e.g., a database, graph, ontology, or CAEX file, so that the mapping logic could also ensure consistency.

The surveyed approaches show a wide range of *pre-processings* and *intermediate models*. Several approaches require a manual modeling of requirements and designs in UML [M4,M6,M12], SysML [M9,M11], or other proprietary notations, such as icsML [M5], ProcGraph [M7], and MeiA [M10]. UML/SysML models are augmented using profiles, such as UML PA (Vogel-Heuser et al., 2005), SysML4IEC61131 (Thramboulidis and Frey, 2011), UML AP (Hästbacka et al., 2011), and SysML AT (Vogel-Heuser et al., 2014).

UML PA [M4] adds time constraints on an architectural level, timed state charts, as well as ports, capsules, protocols, and roles from the UML-RT profile. Furthermore, it uses special object diagrams to express mappings between hardware and software. SysML4IEC61131 [M11] follows a slightly different approach and is closely modelled after IEC 61131-3 concepts, such as Controller, PLC, Application, POU, Program, Task, and Physical I/O. AUKOTON [M12] consists of four subprofiles for requirements (e.g., instrumentation requirements), automation concepts (e.g., control loop, PID algorithm), devices and resources (e.g., EDDL, FDT), and distribution and concurrency (e.g., distributed components and concurrency mechanisms). SysML-AT [M9] adds function block instances, classifiers, flows, and restrictions to SysML to allow an easier mapping to IEC 61131-3 ST. None of these profiles has been applied outside of the respective authors' own work so far. icsML [M5] uses XML to describe hardware, software, and relationships among them. The ProcGraph [M3] domain specific language proposes entity diagrams, state transition diagrams, and state dependency diagrams to specify structural and behavioral information and ultimately an easy mapping to IEC 61131-3 FBD and ST. The MeiA framework [M10] uses four different models: MeiA_MM (e.g., phases, steps, signals), UseCase_MM (e.g., actor, use case, NonFuncReq), GEMMA_MM (e.g., state, line, operation), and Design_MM (i.e., the intermediate modeling layer of AutomationML, which includes GRAFCET and SFC elements).

Approaches operating on a plant structure often use the CAEX format as a common syntactical container and add additional information, such as the LogIX logistics model [M1] to identify equipment between two junction points, or the PandIX model for PCE requests [M2,M3]. [M11] sketches a CAEX2SysML transformation and a manual addition of requirements by the control engineer. The ACPLT Rule Engineering [M3] proposes converting CAEX-based P&IDs into a Neo4J graph notation, so that a graph query language can be used to process the model during translation. The Munich code generator [M13] uses a tree-based model to express the plant hierarchy extracted from P&IDs.

The GRAFCET Translator [M8] requires the control engineer to model sequences in GRAFCET using a self-defined shape library for Microsoft Visio. The GRAFCET Translator can then convert the Visio XML format into the Petri Net Markup Language (PNML), which includes a normalization of the model. Besides code generation, the PLC-Statecharts approach [M8] provides a mapping to UPPAAL timed automata to enable model checking.

Although all approaches produce IEC 61131-3 code, their intermediate processings vary a lot. This is caused by different inputs but also by different focuses regarding the code generation (e.g., interlocks, state-based behavior). Augmenting standardized notations, such as UML or CAEX, for intermediate models can utilize the abstractions developed and accepted by a standardization body in a formal process, which can foster tool support and developer training. There is also potential to combine or link different notations into a comprehensive approach, e.g., plant structures and behavioral notations, as for example shown in AutomationML. Ontological notations may lend themselves to a reuse of existing inference engines to support code generation.

In most approaches, the actual *translation* of the intermediate model to IEC 61131-3 constructs is rather straightforward. Function blocks expressed in UML are mapped to IEC 61131-3 function blocks, states in state charts are mapped to steps in SFCs. The GRAFCET Translator [M8] applies 28 transformation rules to map GRAFCET to SFCs. There are also M2T transformations to ST [M6,M7,M10]. Other approaches use a rule engine to apply domain-specific rules on the intermediate models and generate the code out of these rules [M1,M2]. The ACPLT Rule Engineering approach applies the Cypher graph query language to apply such rules [M3].

Merging generated code into existing code is only considered by the Vienna Code Generator [M2]. Dealing with generated and hand-written code in parallel is not discussed by any of the approaches. None of the approaches lays special emphasis on *documenting* the code generation process, so that the control engineer receives useful feedback on how the inputs were processed. An implicit assumption is that the control engineer inspects the generated code and continues working with it.

Backpropagation is also rarely considered by the approaches, although PLC-Statecharts [M6] assumes a bidirectional mapping between UML and FBDs, so that the control engineer can ideally only work on the level of UML diagrams. The authors of the GRAFCET-Translator (Julius et al., 2019) discuss concepts for bidirectional transformations between GRAFCET and SFCs. However, backpropagating, for example, tag names changes or parameter updates into the requirements specifications is not considered so far.

Table 4
Comparison of Model Transformations: all approaches use an intermediate representation before generating code. Support for merging, documentation, and backpropagation is limited.

Approach	Input Validation	Pre-processing	Intermediate Model	Translation	Merging	Documentation	Backpropagation
[M1] (CAEX Transformer)	n/a (assumes valid CAEX files as input)	Augmentation with flow path using LOGIX model	CAEX (IEC 62424) enhanced with LOGIX line model	Applying XML interlocking rules to generate a C&E Matrix, then FBD/ST	n/a	n/a	n/a
[M2] (Vienna Code Generator)	n/a (assumes valid CAEX and other files as input)	Mapping inputs to reference ontology (based on CAEX)	CAEX (IEC 62424) + PandIX model	Applying rules from a knowledge base to create FBD (SPARQL queries)	Importer for existing control logic code to reference ontology	n/a	n/a
[M3] (ACPLT Rule Engineering)	n/a (assumes valid CAEX files as input)	Translation of CAEX into Graph (neo4J)	Self-defined graph notation	Applying rules encoded as Cypher graph queries to generate FBD	n/a	n/a	n/a
[M4] (UML PA Translator)	n/a	Manual modelling of UML class diagrams and state charts	UML for Process Automation profile (UML PA)	Direct translation of state charts to SFC	n/a	n/a	n/a
[M5] (icsML)	Applying rules on XML models (syntax checks)	Manual modeling of hardware, software in XML	icsML including hardware, software, and relationsships	XML stylesheet to map to PLCopen XML, then ISaGRAF and Simatic	n/a	n/a	n/a
[M6] (PLC-Statecharts)	n/a	Manual modelling of UML class diagrams and state charts	UPPAAL timed automata	Direct translation of state charts to ST	n/a	n/a	Bidirectional mapping between FB and UML class diagrams
[M7] (MAGICS)	n/a (manual re-modeling of input artifacts)	Creation of ProcGraph model, writing of ST statements	ProcGraph DSML (Entity Diag., State Trans. Diag., State Depend. Diag.)	Mapping to ProcGraph elements to FBD, copying ST statements	n/a	n/a	n/a
[M8] (GRAFCET-Translator)	n/a	Transform Visio to PNML, normalize PNML file	GRAFCET in PNML(ISO/IEC15909-2)	Applying 28 transformation rules from GRAFCET to SFC	n/a	n/a	Multiple concepts for round-trip engineering discussed
[M9] (SysML-AT Transformer)	n/a (assumes valid SysML Requirements diagrams)	Manual modeling of SysML Parametric Diagrams	SysML Parametric Diagram + SysML-AT profile	MOFM2 T / OCL transformation into ST	n/a	n/a (generated code to be reviewed by developer)	n/a
[M10] (MeiA Framework)	n/a	Manual modeling of MeiA model	MeiA model (signal/phases), Use Case Model, GEMMA model, AML IML	IML to PLCOpen XML Generation for SFCs	n/a	n/a	n/a
[M11] (SysML4IEC61131)	n/a (assumes valid CAEX files as input)	CAEX2SysML transformation, add requirements	SysML requirements diagram + proposed SysML4IEC61131 profile	SysML2IEC61131 Model Transformer	n/a	n/a	n/a
[M12] (AUKOTON)	n/a (inputs are not validated before automated import)	Manually adding requirements for feedback control	UML AP profile(Requirements, Aut. Concepts, Devices, Distribution)	Direct mapping of UML AP to FBD, use of platform-specific profiles	n/a	n/a	n/a
[M13] (Munich Code Generator)	n/a	Object recognition on P&ID, connection analysis, module recognition	Tree-based representation of plant hierarchy (C+/SQLite)	Mapping to Process Module library, M2T to PLCopen	n/a	n/a	n/a

Legend: ■ Rule-based Engineering approaches, ■ Higher-level Programming approaches, ■ Higher-level Programming using Plant Structure

4.4. Comparison of Outputs

Table 5 summarizes the outputs produced by the surveyed approaches. Several approaches create *signal references* by a direct mapping from UML class diagrams, or derived from P&IDs. Only AUKOTON [M12] imports an I/O list for signal creation. The Vienna Code generator [M2] instantiates diagnostic typicals into FBDs from a library. The Munich Code Generator [M13] instantiates typicals for process modules from a library. There are no common standard libraries used by the approaches for code generation. Some vendors may consider their function block libraries as differentiator and competitive advantage. Domain- or project-specific typicals are discussed [M3,M11], but not demonstrated in any of the surveyed code generations.

A few proposals discuss the *parametrization* of typicals. SysML4IEC61131 [M11] intends to incorporate timing constrains using the UML MARTE profile, but remains vague on details. AUKOTON [M12] parameterizes instantiated function blocks with ranges, alarm limits, and engineering units taken from an I/O list. The Vienna code generator [M2] takes parameters for diagnostic functions from hardware configuration files into account. Due to the heterogeneous inputs and different generation approaches, there is no consensus on this element of code generation. Mechanisms to map information on parameter values to domain- or project-specific typicals could improve code generation.

Most of the approaches express sequential logic for recipes and start-up sequences as SFCs or ST. A few of them use FBDs, but no approach generates LL or IL. As data format, most approaches use PLCOpenXML TC6, which some of them map to vendor-specific formats. For example, icsML [M5] creates an ISaGRAF XML representation and a Siemens STEP7 representation from PLCopenXML. MAGICS [M7] generates code for Mitsubishi PLCs. The Vienna code generator [M2] creates RSLogix5000 code for Rockwell controllers.

The approaches express *safety functions and interlocking logic* mostly as FBDs, some also use SFCs and ST. The CAEX transformer [M1] first produces a C&E matrix for interlocking logic, which can be manually adapted and extended. The final matrix can then be converted into either FBD or ST for an ABB controller.

Most of the other functions of control logic (Guttel et al., 2008) are not explicitly considered by the approaches. The Vienna code generator produces FBDs for diagnostic logic, and the SysML4IEC61131 approach considers timing properties. The SysML-AT transformer [M9] considers control logic for controller-to-controller communication.

4.5. Comparison Summary

The previous subsections have shown that the identified approaches cannot be directly compared in terms of efficiency, quality, or expressiveness, because they require different kinds of inputs, apply different kinds of transformations, and generate different kinds of outputs. Also their tooling is often not or no longer available for independent replication studies or benchmarking. Nevertheless, the survey allowed a categorization into 1) rule-based engineering approaches, 2) higher-level programming approaches, and 3) higher-level programming approaches using a plant structure. The next section will interpret and analyze the survey findings further.

5. Discussion

This section discusses multiple observations and analyzes of the surveyed approaches.

Heterogeneous Research Scopes Although all surveyed approaches produce IEC 61131-3 code, their individual goals and scopes are diverging. Selecting an approach for a particular project mainly depends on the available inputs and the expected outputs.

Rule-based generation approaches try to limit manual coding efforts, by letting control engineers encode simple engineering tasks into a rule base. The aim is not to create a higher-level programming interface, but rather to avoid manual coding as much as possible by automating simple, repetitive tasks. The approaches demonstrated rule-based engineering only on simple examples, scalability and robustness remains unexplored. Expert systems based on rules are known to become brittle if facing unfamiliar settings (Wiriyacoonkasem and Esterline, 2000), so a more thorough investigation is needed to better assess the usefulness of rule-based engineering.

Approaches based on higher-level programming languages, such as UML or SysML intend to make manual coding more productive, in terms of understandability and quality by utilizing object-oriented concepts and raising the abstraction level. They include automated code generation as a translation to a lower-level notation, but rather treat it as a side-effect while the focus is on providing an adequate higher-level notation. Accordingly, the evaluations of such approaches focus on programming efficiency and user satisfaction and do not differentiate between manually written and generated code. They implicitly assume that all code can be expressed in the higher-level notation and that a bidirectional mapping to the lower-level IEC 61131-3 notation is possible.

Although the classes of surveyed approaches have different goals, they do not exclude each other and could be combined. To some extent, this is demonstrated by the hybrid approaches that take a plant structure into account to generate a UML/SysML-based structure model, which then can be manually enhanced in a modeling tool.

The scope of the academic research was focused on the three different classes of approaches shown in the survey. In practice, there are additional methods for lowering control engineering efforts, which are hardly investigated by the surveyed approaches:

- **Copying Code:** Often, individual engineers simply copy and adapt code from former projects to speed up programming. Methods could be developed to recommend similar former source code or to support creating reusable engineering libraries based on analyzing engineering artifacts of similar projects, which requires extensive domain knowledge.
- **Engineering Libraries:** Engineers create engineering libraries encoding higher-level functionality in reusable function blocks. This works well in application domains with limited customized functionality, e.g, steel making or aluminium smelting, but is harder for other domains, such as chemical plants, that often exhibit an amount of specialized functionality.
- **Modular Automation:** As a more coarse-grained approach to control logic reuse, engineers segment a production process into smaller package units, which independent subcontractors build together with an included automation. The engineering configuration of these package units follows the NAMUR Module Type Packages (MTP) (Bernshausen et al., 2016). Besides standardized hardware connectors, each unit provides a module type package that serves as module description and input to system-wide engineering tools. An orchestrating, supervising control system can then provide high-level commands to each unit (e.g., start, stop) and use the modules to execute specific recipes.

The Munich code generator included in this survey (Koltun et al., 2018) is potentially applicable for Modular Automation, since it works with higher-level ISA-88 based module descriptions similar to NAMUR MTP. However, the challenges for automatic code generation change significantly in case of modular engineering. Because the modules are too small to be concise, they

Table 5
Comparison of Transformation Outputs: all approaches produce IEC 61131-3 SFC, FBD, and ST. Primary focus is on sequential logic and interlocking logic.

Approach	Creation of Variables / Signal References + Instantiation of Typicals	Parametrization of Typicals (incl. set points, alarm limits, PID parameters)	Sequential Logic (incl. recipes, start-Up, shutdown, ...)	Safety Functions and Interlocking Logic (incl. reset, emergency stop, etc.)	Diagnostic Logic (e.g., process/system diagnosis, field devices)	Other Logic (e.g., asset monitoring, controller communication, etc.)
[M1] (CAEX Transformer)	n/a (implicitly extracted from CAEX)	n/a	n/a	Cause&Effect Matrix converted to 61131-3 ST, FBD (ABB)	n/a	n/a
[M2] (Vienna Code Generator)	Yes, from self-defined Profibus + Ethercat Diagnotics FB Lib	Yes, some parameters from hardware config file	IEC 61131-3 SFC (PLCopen XML)	IEC 61131-3 FBD (PLCopen XML)	IEC 61131-3 FBD (PLCopen XML)	n/a
[M3] (ACPLT Rule Engineering)	Yes, via Cypher query.	Yes, queries refer to function block types.	IEC 61131-3	FBD n/a	n/a	n/a
[M4] (UML PA Translator)	Yes, from UML class diagram	Possible from UML class diagram	IEC 61131-3 SFC, ST	n/a	n/a	n/a
[M5] (icsML)	Yes, from XML file.	n/a	IEC 61131-3 ST (PLCOpenXML)	n/a	n/a	n/a
[M6] (PLC-Statecharts)	Yes, from UML class diagram	Possible from UML class diagram	IEC 61131-3 ST (CoDeSys)	n/a	n/a	n/a
[M7] (MAGICS)	n/a (interpretation of P&ID, creation of ProcGraph model)	n/a	IEC 61131-3 SFC, ST (PLCopen XML)	IEC 61131-3 SFC, ST (PLCopen XML)	n/a	n/a
[M8] (GRAFCET-Translator)	n/a (implicitly extracted from GRAFCET)	Possible using time constraints	IEC 61131-3 SFC, ST (PLCopen XML)	IEC 61131-3 SFC, ST (PLCopen XML)	n/a	n/a
[M9] (SysML-AT Transformer)	Yes, generation of FB based on SysML model	n/a	IEC 61131-3 ST	n/a	n/a	Controller to Controller Communication
[M10] (MeiA Framework)	Yes, signal references manually created in Meia model	n/a	IEC 61131-3 SFC (PLCopen XML)	n/a	n/a	n/a
[M11](SysML4IEC61131)	n/a (possible via SysML Stereotypes)	Via OMG MARTE profile	IEC 61131-3 SFC, FBD (PLCopen XML)	IEC 61131-3 FBD (PLCopen XML)	n/a	Timing properties via OMG MARTE profile
[M12](AUKOTON)	Yes, by mapping IO list to AUKOTON DCS Library	Parameters taken from IO List	n/a	IEC 61131-3 FBD (PLCopen XML)	n/a	n/a
[M13](Munich Code Generator)	Yes, instantiating predefined process modules	n/a	IEC 61131-3 ST (PLCopenXML)	n/a	n/a	n/a

Legend: ■ Rule-based Engineering approaches, ■ Higher-level Programming approaches, ■ Higher-level Programming using Plant Structure

usually do not include a large amount of IEC 61131-3 logic. Thus, the benefit of code generation for each module provider is rather limited. Module providers can rather exploit economies of scale if they produce their firmly defined modules in large quantities, each time reusing the same control logic.

Based on the survey findings, it seems unlikely that there ever will be an all-accompanying approach for automatic code generation in industrial automation. The different approaches may work well in specific domains, but not in others, depending on how standardized the processes are. Combining approaches that exploit plant topology models, execute code generation rules, and rely on higher-level programming to some extent is possible and should be investigated in future research. Rule-based engineering can reduce the amount of trivial, repetitive implementation tasks, while higher-level programming can reduce efforts for implementing required custom control logic.

Iterative Code Generation As user requirements usually become available in several iterations, code generation cannot assume a one-shot translation of the inputs. The survey has shown that many approaches work with the assumption of a one-shot translation and thus may be misaligned with practice. Some approaches assume that the generated IEC 61131-3 code may not be altered manually (Vogel-Heuser et al., 2005), others allow a bidirectional mapping between UML/SysML notations and IEC 61131-3 (Witsch et al., 2010). The Vienna code generator (Steinegger et al., 2017) first imports and analyzes existing control logic before adding generated code in an informed manner.

Best practices for model-driven software development (Voelter and Bettin, 2004; Stahl et al., 2006) suggest to separate generated and non-generated code, for example in different files, so that an iterative development is supported. A code generator can then easily overwrite files with generated code in a re-engineering scenario without affecting manually written code. There are different methods for joining generated and non-generated code, including interfaces, design patterns (e.g., factory, strategy, bridge), and template methods, which are specific for object-oriented programming languages. For IEC 61131-3 code generated and non-generated code may for example be arranged in different diagrams. During manual IEC 61131-3 coding, the control engineer needs to take care not to contradict the generated control logic.

Versioning and file comparison tools are another means for merging generated and manually written code during iterative code generation. This often works almost automated in case of additive actions, but may require manual intervention in case of changes to existing code. None of the surveyed approaches discusses these possibilities though.

Input Data Validation Validation of customer inputs is often a time-consuming problem in practice, which may require multiple feedback cycles between engineering contractor and automation vendor. Most of the surveyed approaches do not deal with this issue beyond syntactical XML checks or assuming that a prior mapping to AutomationML/CAEX produced well-formed inputs.

In practice, there are different methods to address this issue. Engineering contractors can use a common database, such as SmartPlant Instrumentation[39] to assure consistency of instrument indices. Additionally, they can utilize advanced CAD tools that allow validations of diagrams. But this approach is challenged if multiple subcontractors are involved in the engineering (Drath and Barth, 2011). There are also engineering platforms, such as COMOS[40] EPlan[41] or EBase[42] which may support automation vendors in consistency and completeness checks. Nevertheless, more research into cross-tool data consistency and completeness checks may be valuable.

Dealing with Natural Language Requirements Another finding of the survey is that none of the approaches is able to process informal customer requirements. As Section 2 showed, part of the customer requirements are often formulated using informal language in user requirement documents or control narratives. Additionally, P&IDs or logic diagrams may contain annotations in prose writing, which may provide important information for control logic engineering.

Requiring customers and engineering contractors to fully use formal requirements specifications may be unrealistic. However, there is a large body of literature on text mining (Feldman and Sanger, 2007) and information retrieval (Baeza-Yates et al., 2011), which have been applied on informal requirements in other application domains.

Gelhausen and Tichy (2007) propose to annotate requirement documents using a purpose-built language, so that they can be transformed into UML diagrams. Deeptimahanti and Babar (2009) propose a tool to generate UML models from natural language requirements by identifying classes and stereotypes. The tool implements a set of syntactic reconstruction rules to process complex requirements into simple requirements. Le et al. (2013) present a system for synthesizing smartphone automation scripts from domain-specific natural language descriptions. System components and their partial dataflow relations are inferred from the natural language description using a specialized parser. Tichy et al. (2015) presents nlrpBENCH, a benchmark consisting of over 50 requirements documents to train model extraction and text correction approaches. Chalkidis et al. (2017) describe and experimentally compare several contract element extraction methods that use manually written rules and linear classifiers (logistic regression, support vector machines) with hand-crafted features, word embeddings, and part-of-speech tag embeddings.

Most of these approaches still work on rather simple requirements texts that are sometimes heavily constrained or annotated to allow processing. If texts are constrained into semi-formal language, the customer benefit of creating such requirements fast and with limited special expertise may be invalidated. If pre-annotation of the text requires too much effort, control engineers would likely fall back to manual interpretation, which in addition may be more robust. Practical problems, such as local languages, ambiguities in the requirements, and non-standard terminology may complicate natural language processing approaches further. However, the limits for text mining in this context are still not well understood.

Standardized File Formats Both the rule-based approaches and higher-level programming approaches using the plant structure in this survey rely on standardized input file formats. However, in practice, the file formats are often not yet readily available. As described in Section 2, many artifacts from customers are still exchanged on paper or as unstructured PDF-files that complicate automated processing (Arroyo et al., 2016). This has multiple reasons as detailed in the following.

In some cases (e.g., for P&ID diagrams) there is a *lack of standardized formats* so far, or a *lack of adoption* of existing formats from the software vendors. Smart P&IDs with an database underlying the drawing capturing properties of instruments and equipment are still uncommon, the AutoCAD DWG format is often used for convenience, since it is also used for many other types of drawings and is supported by many general purpose drawing tools. The DEXPI initiative is working on an ISO15926-based exchange format for P&IDs. AutomationML (IEC 62714) is gaining more support, but remains far from broad industry adoption (Drath and Ingebrigtsen, 2018). There is no agreed format for I/O lists, but companies often work with specific guidelines. GRAFCET is a standardized notation, but suffered from a missing agreed file format and tool-

[39] https://hexagonppm.com/.
[40] https://www.siemens.com/comos.
[41] https://www.eplan.de/.
[42] https://www.aucotec.com/.

ing, and it is known only in specific regions (Schumacher et al., 2013a).

Furthermore, some automation customers or engineering contractors are concerned about losing intellectual property if specifications are exchanged in object-oriented and standardized notations. They fear that production processes may be more likely replicated by competitors if the files are easier to distribute and process. These concerns need to be addressed in a manner that does not complicate automated code generation, e.g., filtering or obfuscating certain IP sensitive specifications or enforcing confidentiality using processes.

Another factor is the high heterogeneity of application domains, engineering workflows, and information models. The CAEX standard (IEC 62464) acknowledged this fact explicitly and separated syntactic standardization (in XML) from semantic standardization (using role class libraries). This allows to syntactically treat engineering artifacts in a uniform way, even if the semantics are understood only partially. Drath and Barth (2011) developed additional concepts concepts reqarding tool interoperability and data ownership on top of CAEX. Such concepts have so far been applied only in company-internal settings (e.g., Daimler (Burlein et al., 2018), ABB (Bihani and Drath, 2017)) for the exchange of engineering data, but not across different organizations (Biffl et al., 2017).

Nevertheless, numerous customer organizations, such as NAMUR, OPAF, DEXPI are pushing for more standards and their adoption, which may prove beneficial for automated code generation approaches directly processing customer artifacts.

Cost/Benefits of Code Generation The survey did not find any cost/benefit studies for automated code generation in industrial automation. Return on investment for IEC 61131-3 code generation is not yet well validated and remains vague. Setting up a tool chain for code generation may be costly and require a series of multiple similar projects to actually pay off. Control engineers need to create importers for their engineering tools, set up knowledge bases, and elicit domain rules. They may need to train developers in modified UML/SysML versions. Whether the code generation is robust and reliable in a realistic project is not well investigated.

While several experiments (Vogel-Heuser, 2014; Obermeier et al., 2015) have demonstrated that control engineers can be more productive using a UML-based notation, it remains unclear if this benefit outweighs the drawback of introducing an additional UML tool. Control engineers may be bound to the engineering tools their employer provides for a commercial system, which may make it hard to introduce additional external tools.

It is unclear how much code can be generated, and how much code is better created manually. For example, complex multicascading logic involving sophisticated algorithms may require human expertise for the foreseeable future and might be difficult to generate automatically. An elaborated code generation approach may make the application implementation intransparent for control engineers, who might lose confidence in the generation if it proves unreliable.

Benefits regarding higher code quality or saved time in relation to the overall engineering time lack hard evidence. Case studies need to be carried out with industry to better characterize cost and benefits of code generation considering different application domains and their constraints.

6. Threats to validity

The comparison of code generation methods in this paper aimed to analyze the prerequisites and capabilities of the all relevant published methods. The following threats to validity of this survey have been explicitly considered and addressed:

- **Non-Representative Selection of Approaches:** Section 3 has documented the used search facilities, search terms, and inclusion criteria of this paper. Excluding approaches generating IEC 61499 control logic as well as other programming languages is a threat to the representativeness of the survey, but was decided for conciseness and practicality (Thramboulidis, 2013).
- **Irrelevant Classification Criteria:** The classification criteria depicted in Fig. 12 are derived from the inputs required in practice, the necessary transformation steps for a working code generation, and the typical outputs of such approaches. We used external references (Steinegger and Zoitl, 2012; Guttel et al., 2008) to make these criteria representative, more relevant, and reduce subjective author bias. We also contacted domain experts and reviewed numerous customer requirements specifications to get a deeper understanding of the inputs available in practice.
- **Incorrect Analysis of Approaches:** We were not able to test the approaches by executing their tooling on self-defined cases. For almost all of the approaches, the tooling was not readily available for reproduction. Thus, the analysis of the approaches is purely based on literature review.
- **Author Bias:** The authors of this survey are affiliated with an automation company that provides commercial products for industrial automation. This may lead to a bias towards the contexts and perspectives of the authors. However, none of the surveyed approaches is currently used by the authors' company, therefore assuring a level of objectivity.

7. Related work

Related Surveys Although there is no classification and comparison of IEC 61131-3 control logic generation in literature, there are several publications providing overviews of different model-driven approaches and code generators for industrial automation applications. Vyatkin's state-of-the-art review of software engineering in industrial automation (Vyatkin, 2013) summarizes several approaches for model-driven engineering according to the OMG standards MDA, UML, and MOF. This coarsely maps to the higher-level programming approaches from our survey, but also includes approaches dealing with IEC 61499 control logic. The paper does not deal with ruled-based code generation approaches and describes its included approaches on a higher abstraction level.

The survey by Yang et al. (2014) is also broader and less detailed than our survey, reviewing MATLAB Simulink, SCADE, OpenRTM, and IEC 61499 engineering concepts and tools. Vogel-Heuser et al. (2015) describe general challenges and research directions for software development for automated production systems. They point out that round-trip engineering, tool integration, tool usability, and education are major challenges for model-driven engineering in industrial automation. They also list a number of UML/SysML related approaches, however without classifying them.

Lukman et al. (2013) provided a short state-of-the-art analysis of process control engineering approaches, without including rule-based approaches. They also included IEC 61499 approaches as well as approaches not generating classical control logic. They attributed the limited industry adoption of these approaches to a "lack of automatic PLC code generation, lack of development process definition and guidelines, and non-existent or immature tools support, besides the use of device-centric abstractions". Liebel et al. (2014) assessed the state-of-practice regarding model-driven engineering for embedded systems, but did not specifically deal with IEC 61131 control logic.

Related but Excluded Approaches Several related approaches have been excluded for different reasons. FUJABA (Schäfer et al., 2004) is an integrated tool environment, where Siemens PLC code can be

generated from SDL block diagrams, UML class diagrams, and UML behavior diagrams. This approach has been developed more than 15 years ago. The FAVA approach (Fay et al., 2015) translates SysML requirements diagrams and manually specified models to Continuous Function Charts that can be loaded into many PLCs.

Researchers have proposed several IEC 61499 approaches involving code generation, but none of them were developed into a robust tooling. Hussain and Frey (2006) generate IEC 61499 FB networks and test cases out of UML use case, component, sequence, activity, and state diagrams. Panjaitan and Frey (2006) map UML and component diagrams to IEC 61499 FB networks. Thramboulidis and Buda (2010); Thramboulidis (2010) propose the 3+1 SysML model to generate IEC 61499 code. Wenger et al. (2009) propose a converter to transform IEC 61131-3 control logic into IEC 61499 logic to be able to utilize its additional features. Dai and Vyatkin (2009) discuss migrating distributed IEC 61131-3 PLC code to IEC 61499 function blocks and illustrate three different migration methods on a conveyor belt system.

Yang et al. (2017) propose a method to transform IEC 61850 system configuration language (SCL) specifications into logical connections in an IEC 61499 substation automation control application. This involves creating an ontology from the specifications in the Web Ontology Language (OWL), which is then transformed into logical connections of IEC 61499 function blocks using the enhanced Semantic Web Rule Language (eSWRL). Voinov et al. (2017) created an automatic HMI generator for this method. Furthermore, the authors extended the approach to use restricted natural language (RNL) using Boilerplate models as input (Yang et al., 2020). While this overall method was demonstrated for smart grid systems, the mechanisms behind it could be transferred to other domains as well.

Furthermore, several older code generation approaches use formal modeling notations, such as Petri nets or timed automata. Cutts and Rattigan (1992) propose a PLC code generation method with Petri Net-based modeling techniques for a three-stage manufacturing system. Jörns et al. (1995) introduced signal interpreted Petri nets and translated them to LD or IL code. Frey (2000) propose a PLC code generation approach by arranging code fragments which directly correspond to Petri net elements. Thieme and Hanisch (2002) present a modeling formalism to generate modular control logic using IEC 61131 functions blocks. Music et al. (2005) apply real-time petri nets to generate control logic implementations. Sacha (2005) proposes a formalization for transformations of timed finite state machines into PLC control code. Endsley et al. (2006) generate control logic for a conveyor system out of finite state machines but do not specifically aim at IEC 61131-3 code. Flordal et al. (2007) generate executable interlocking policies implemented in PLC programming languages for industrial robot cells. Bergert et al. (2007) generate IEC 61131 SFCs from Pert charts encoded in XML. Wang et al. (2009) use timed automata and create an automated translation into LD.

8. Conclusions

This paper classified 13 control logic generation approaches for IEC 61131-3 programming languages. Automating the task of software implementation for real-time controllers in industrial automation could result in a significant cost reduction for the engineering process of industrial plants. The classification criteria used in this paper originate from the customer specifications available in practice (inputs), an idealized model transformation process, and the typical outputs of control logic formats and kinds. The analysis of the 13 approaches showed that they have different goals and follow different patterns, which complicates direct comparisons for quality attributes, such as performance or specification coverage. Most approaches remain in a proof-of-concept maturity.

The classification framework in this paper can benefit practitioners and researchers. Practitioners get a condensed state-of-the-art review for code generation in industrial automation and can thus faster assess the benefits and drawbacks of each approach. Although practitioners cannot apply most approaches directly in real projects due to the likely mismatch of available inputs and immature tool support, they can evaluate the potential of code generation independently and work towards using the standard file formats suggested by the approaches. Researchers get a blueprint to better compare and categorize existing and future approaches. Research challenges derived from the comparison revolve around powerful and robust knowledge bases for rule-based engineering, processing natural language requirements, and supporting iterative engineering processes with input validation, bi-directional transformations, and code merging. Combinations of rule-based engineering approaches and higher-level programming approaches should be investigated.

Upon a maturation of the surveyed approaches, a benchmark for automatic IEC 61131-3 code generation should be developed to better compare their capabilities and give out research challenges to extend the code generation capabilities. Such a benchmark should include representative inputs artifacts available in practice in different file formats. It could include natural language requirements to test text mining approaches. Benchmark scores could be defined for specification coverage, efficiency, and user-friendliness. Such a benchmark would need to be maintained periodically and be updated with new community challenges.

In the future, control logic generation approaches may produce IEC 61499 control logic or other programming languages and interface with approaches for Modular Automation (NAMUR MTP (Bernshausen et al., 2016)), if these languages and initiatives achieve a higher market penetration. User studies should be conducted to assess control logic generation approaches. Criteria or specific project contexts should be defined to help practitioners to decide for or against code generation given certain project constraints.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Heiko Koziolek: Conceptualization, Methodology, Writing - original draft, Visualization, Supervision. **Andreas Burger:** Conceptualization, Investigation, Writing - review & editing. **Marie Platenius-Mohr:** Conceptualization, Investigation, Writing - review & editing. **Raoul Jetley:** Conceptualization, Investigation, Writing - review & editing.

References

Alvarez, M.L., Sarachaga, I., Burgos, A., Estévez, E., Marcos, M., 2018. A methodological approach to model-driven design and development of automation systems. IEEE Trans. Autom. Sci. Eng. 15 (1), 67–79.

Arroyo, E., Hoernicke, M., Rodríguez, P., Fay, A., 2016. Automatic derivation of qualitative plant simulation models from legacy piping and instrumentation diagrams. Comput. Chem. Eng. 92, 112–132.

Baeza-Yates, R., Ribeiro, B.d.A.N., et al., 2011. Modern information retrieval. New York: ACM Press; Harlow, England: Addison-Wesley,.

Barth, M., Drath, R., Fay, A., Zimmer, F., Eckert, K., 2012. Evaluation of the openness of automation tools for interoperability in engineering tool chains. In: Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012). IEEE, pp. 1–8.

Barth, M., Fay, A., 2013. Automated generation of simulation models for control code tests. Control Eng. Pract. 21 (2), 218–230.

Bergert, M., Diedrich, C., Kiefer, J., Bar, T., 2007. Automated PLC software generation based on standardized digital process information. In: 2007 IEEE Conference on Emerging Technologies and Factory Automation (EFTA 2007). IEEE, pp. 352–359.

Bernshausen, J., Haller, A., Holm, T., Hoernicke, M., Obst, M., Ladiges, J., 2016. Namur modul type package–Definition. atp magazin 58 (01–02), 72–81.

Biffl, S., Mordinyi, R., Steininger, H., Winkler, D., 2017. Integrationsplattform Für Anlagenmodellorientiertes Engineering. In: Handbuch Industrie 4.0 Bd. 2. Springer, pp. 189–212.

Bihani, P., Drath, R., 2017. Concept for automationml-based interoperability between multiple independent engineering tools without semantic harmonization: Experiences with automationml. In: 2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA). IEEE, pp. 1–8.

Breyfogle III, F.W., Cupello, J.M., Meadows, B., 2000. Managing six sigma: A practical guide to understanding, assessing, and implementing the strategy that yields bottom-line success. John Wiley & Sons.

Burlein, J., Rassl, M., Schmidt, N., 2018. Introducing AutomationML in a heterogeneous software tool landscape - a success story, 5th AutomationML User Conference, 2018. https://bit.ly/2LOfN4p.

Chalkidis, I., Androutsopoulos, I., Michos, A., 2017. Extracting Contract Elements. In: Proceedings of the 16th Edition of the International Conference on Articial Intelligence and Law. ACM, New York, NY, USA, pp. 19–28. doi:10.1145/3086512.3086515.

Cutts, G., Rattigan, S., 1992. Using Petri nets to develop programs for PLC systems. In: International Conference on Application and Theory of Petri Nets. Springer, pp. 368–372.

Dai, W.W., Vyatkin, V., 2009. A case study on migration from iec 61131 plc to iec 61499 function block control. In: 2009 7th IEEE International Conference on Industrial Informatics. IEEE, pp. 79–84.

Deeptimahanti, D.K., Babar, M.A., 2009. An Automated Tool for Generating UML Models from Natural Language Requirements. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering. IEEE Computer Society, Washington, DC, USA, pp. 680–682. doi:10.1109/ASE.2009.48.

Doherr, F., Urbas, L., Drumm, O., Franze, V., 2013. Bedienbilder auf Knopfdruck. atp magazin 53 (11), 30–39.

Drath, R., Barth, M., 2011. Concept for interoperability between independent engineering tools of heterogeneous disciplines. In: ETFA2011. IEEE, pp. 1–8.

Drath, R., Fay, A., Schmidberger, T., 2006. Computer-aided design and implementation of interlock control code. In: IEEE Conference on Computer Aided Control System Design. IEEE, pp. 2653–2658.

Drath, R., Ingebrigtsen, I., 2018. Digitalization of the IEC PAS 63131 Standard with AutomationML. In: 2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA), 1, pp. 901–909. doi:10.1109/ETFA.2018.8502458.

Endsley, E., Almeida, E., Tilbury, D., 2006. Modular finite state machines: development and application to reconfigurable manufacturing cell controller generation. Control Eng. Pract. 14 (10), 1127–1142.

Fay, A., 2003. A knowledge-based system to translate control system applications. Eng. Appl. Artif. Intell. 16 (5–6), 567–577.

Fay, A., 2003. A knowledge-based system to translate control system applications. Eng. Appl. Artif. Intell. 16 (5–6), 567–577.

Fay, A., Drath, R., Bort, P., 2001. Design and implementation of a Java-based industrial control system configuration tool. In: ETFA 2001. 8th International Conference on Emerging Technologies and Factory Automation. Proceedings (Cat. No. 01TH8597), 2. IEEE, pp. 553–558.

Fay, A., Schmidberger, T., Scherf, T., 2009. Knowledge-based support of HAZOP studies using a CAEX plant model. Inside Funct. Saf. 2009 (2), 5–15.

Fay, A., Vogel-Heuser, B., Frank, T., Eckert, K., Hadlich, T., Diedrich, C., 2015. Enhancing a model-based engineering approach for distributed manufacturing automation systems with characteristics and design patterns. J. Syst. Softw. 101, 221–235.

Feldman, R., Sanger, J., 2007. The text mining handbook: Advanced approaches in analyzing unstructured data. Cambridge university press.

Flordal, H., Fabian, M., Åkesson, K., Spensieri, D., 2007. Automatic model generation and PLC-code implementation for interlocking policies in industrial robot cells. Control Eng. Pract. 15 (11), 1416–1426.

Forbes, H., Clayton, D., 2018. Distributed Control Systems Global Market 2017–2022. ARC Market Analysis, ARC Advisory Group, https://www.arcweb.com/market-studies/distributed-control-systems.

Frey, G., 2000. Automatic implementation of Petri net based control algorithms on PLC. In: Proceedings of the 2000 American Control Conference. ACC (IEEE Cat. No. 00CH36334), 4. IEEE, pp. 2819–2823.

Friedrich, D., Vogel-Heuser, B., 2007. Benefit of system modeling in automation and control education. In: 2007 American Control Conference. IEEE, pp. 2497–2502.

Gelhausen, T., Tichy, W.F., 2007. Thematic role based generation of UML models from real world requirements. In: International Conference on Semantic Computing (ICSC 2007). IEEE, pp. 282–289.

Grüner, S., Weber, P., Epple, U., 2014. A model for discrete product flows in manufacturing plants. In: Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA). IEEE, pp. 1–8.

Grüner, S., Weber, P., Epple, U., 2014. Rule-based engineering using declarative graph database queries. In: 2014 12th IEEE International Conference on Industrial Informatics (INDIN). IEEE, pp. 274–279.

Gutermuth, G., 2010. Collaborative Process Automation Systems. ISA, pp. 156–182.

Guttel, K., Weber, P., Fay, A., 2008. Automatic generation of PLC code beyond the nominal sequence. In: 2008 IEEE International Conference on Emerging Technologies and Factory Automation. IEEE, pp. 1277–1284.

Hästbacka, D., Vepsäläinen, T., Kuikka, S., 2011. Model-driven development of industrial process control applications. J. Syst. Softw. 84 (7), 1100–1113.

Hussain, T., Frey, G., 2006. UML-based development process for IEC 61499 with automatic test-case generation. In: 2006 IEEE Conference on Emerging Technologies and Factory Automation. IEEE, pp. 1277–1284.

Jamro, M., Rzonca, D., 2018. Agile and hierarchical round-trip engineering of IEC 61131-3 control software. Comput. Ind. 96, 1–9.

Jörns, C., Litz, L., Bergold, S., 1995. Automatische erzeugung von SPS-Programmen auf der basis von petri-Netzen. Automatisierungstechnische Praxis 37 (3), 10–14.

Julius, R., Fink, V., Uelzen, S., Fay, A., 2019. Konzept zur bidirektionalen transformation zwischen GRAFCET-Spezifikationen und IEC 61131-3 steuerungscode. at-Automatisierungstechnik 67 (3), 208–217.

Julius, R., Schürenberg, M., Schumacher, F., Fay, A., 2017. Transformation of GRAFCET to PLC code including hierarchical structures. Control Eng. Pract. 64, 173–194.

Katzke, U., Vogel-Heuser, B., 2005. UML-PA As an engineering model for distributed process automation. IFAC Proceed. 38 (1), 129–134.

Kitchenham, B., Charters, S., 2007. Guidelines for performing systematic literature reviews in software engineering. Technical Report. Technical report, Ver. 2.3 EBSE Technical Report. EBSE.

Koltun, G., Kolter, M., Vogel-Heuser, B., 2018. Automated Generation of Modular PLC Control Software from P&ID Diagrams in Process Industry. In: 2018 IEEE International Systems Engineering Symposium (ISSE). IEEE, pp. 1–8.

Krausser, T., Quirós, G., Epple, U., 2011. An IEC-61131-based rule system for integrated automation engineering: Concept and case study. In: 2011 9th IEEE International Conference on Industrial Informatics. IEEE, pp. 539–544.

Le, V., Gulwani, S., Su, Z., 2013. SmartSynth: Synthesizing Smartphone Automation Scripts from Natural Language. In: Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services. ACM, New York, NY, USA, pp. 193–206. doi:10.1145/2462456.2464443.

Liebel, G., Marko, N., Tichy, M., Leitner, A., Hansson, J., 2014. Assessing the state-of-practice of model-based engineering in the embedded systems domain. In: International Conference on Model Driven Engineering Languages and Systems. Springer, pp. 166–182.

Ljungkrantz, O., Akesson, K., 2007. A study of industrial logic control programming using library components. In: 2007 IEEE International Conference on Automation Science and Engineering. IEEE, pp. 117–122.

Lukman, T., Godena, G., Gray, J., Heričko, M., Strmčnik, S., 2013. Model-driven engineering of process control software–beyond device-centric abstractions. Control Eng. Pract. 21 (8), 1078–1096.

Martin, P., Hale, G., 2010. Automation made easy: everything you wanted to know about automation and need to ask. Int. Soc. Autom.

Mens, T., Van Gorp, P., 2006. A taxonomy of model transformation. Electron. Notes Theor. Comput. Sci. 152, 125–142.

Music, G., Gradisar, D., Matko, D., 2005. IEC 61131-3 compliant control code generation from discrete event models. In: Proceedings of the 2005 IEEE International Symposium on, Mediterrean Conference on Control and Automation Intelligent Control, 2005.. IEEE, pp. 346–351.

Obermeier, M., Braun, S., Vogel-Heuser, B., 2015. A model-driven approach on object-oriented PLC programming for manufacturing systems with regard to usability. IEEE Trans. Ind. Inf. 11 (3), 790–800.

Otto, A., Hellmann, K., 2009. IEC 61131: A general overview and emerging trends. IEEE Ind. Electron. Mag. 3 (4), 27–31.

Panjaitan, S., Frey, G., 2006. Combination of UML modeling and the IEC 61499 function block concept for the development of distributed automation systems. In: 2006 IEEE Conference on Emerging Technologies and Factory Automation. IEEE, pp. 766–773.

Sacha, K., 2005. Automatic code generation for PLC controllers. In: International Conference on Computer Safety, Reliability, and Security. Springer, pp. 303–316.

Schäfer, W., Wagner, R., Gausemeier, J., Eckes, R., 2004. An EngineerâS Workstation to Support Integrated Development of Flexible Production Control Systems. In: Integration of Software Specification Techniques for Applications in Engineering. Springer, pp. 48–68.

Schmidberger, T., Fay, A., 2007. A rule format for industrial plant information reasoning. In: 2007 IEEE Conference on Emerging Technologies and Factory Automation (EFTA 2007). IEEE, pp. 360–367.

Schmidberger, T., Horch, A., Fay, A., Drath, R., Breitenecker, F., Troch, I., 2006. Rule based engineering of asset management system functionality. 5th Vienna Symposium on Mathematical Modelling, 8.

Schmitz, S., Epple, U., 2007. Automated engineering of human machine interfaces. In: VDI/VDE Gesellschaft Mess-und Automatisierungstechnik, pp. 127–138.

Schüller, A., Epple, U., 2012. PandIX – Exchanging P&I diagram model data. In: Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies Factory Automation (ETFA 2012), pp. 1–8. doi:10.1109/ETFA.2012.6489537.

Schumacher, F., Fay, A., 2014. Formal representation of GRAFCET to automatically generate control code. Control Eng. Pract. 33, 84–93.

Schumacher, F., Schröck, S., Fay, A., 2013. Tool support for an automatic transformation of GRAFCET specifications into IEC 61131-3 control code. In: 2013 IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA). IEEE, pp. 1–4.

Schumacher, F., Schröck, S., Fay, A., 2013. Transforming hierarchical concepts of GRAFCET into a suitable Petri net formalism. IFAC Proceed. 46 (9), 295–300.

Stahl, T., Voelter, M., Czarnecki, K., 2006. Model-driven software development: Technology, engineering, management. John Wiley & Sons, Inc.

Steinegger, M., Melik-Merkumians, M., Schitter, G., 2017. Ontology-based framework for the generation of interlock code with redundancy elimination. In: Proc. 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA). IEEE, pp. 1–5.
Steinegger, M., Melik-Merkumians, M., Zajc, J., Schitter, G., 2016. Automatic generation of diagnostic handling code for decentralized PLC-based control architectures. In: Proc. IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA). IEEE, pp. 1–8.
Steinegger, M., Zoitl, A., 2012. Automated code generation for programmable logic controllers based on knowledge acquisition from engineering artifacts: Concept and case study. In: Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012). IEEE, pp. 1–8.
Sun, Q., 2013. A method for generating process topology-based causal models. Aalto University School of Chemical Technology.
Suresh, V.P., Chakrabarti, S., Jetley, R., 2019. Automated Test Case Generation for Programmable Logic Controller Code. In: Proceedings of the 12th Innovations on Software Engineering Conference (formerly known as India Software Engineering Conference). ACM, p. 29.
Tan, W.C., Chen, I.-M., Tan, H.K., 2016. Automated identification of components in raster piping and instrumentation diagram with minimal pre-processing. In: 2016 IEEE International Conference on Automation Science and Engineering (CASE). IEEE, pp. 1301–1306.
Thieme, J., Hanisch, H.-M., 2002. Model-based generation of modular plc code using iec61131 function blocks. In: Proceedings of the International Symposium on Industrial Electronics, 1, pp. 199–204.
Thramboulidis, K., 2010. The 3+ 1 SysML view-model in model integrated mechatronics. J. Softw. Eng. Applic. 3 (02), 109.
Thramboulidis, K., 2013. IEC 61499 Vs. 61131: a Comparison based on misperceptions. arXiv:1303.4761. https://arxiv.org/ftp/arxiv/papers/1303/1303.4761.pdf.
Thramboulidis, K., Buda, A., 2010. 3+ 1 SysML view model for IEC61499 Function Block control systems. In: 2010 8th IEEE International Conference on Industrial Informatics. IEEE, pp. 175–180.
Thramboulidis, K., Frey, G., 2011. An MDD process for IEC 61131-based industrial automation systems. In: ETFA2011. IEEE, pp. 1–8.
Thramboulidis, K.C., 2004. Using UML in control and automation: a model driven approach. In: 2nd IEEE International Conference on Industrial Informatics, 2004. INDIN'04. IEEE, pp. 587–593.
Tichy, W.F., Landhäußer, M., Körner, S.J., 2015. nlrpBENCH: a benchmark for natural language requirements processing. Multikonferenz Software Engineering & Management 2015. Gesellschaft für Informatik eV.
Tiegelkamp, M., John, K.-H., 1995. IEC 61131-3: Programming industrial Automation Systems. Springer.
Tikhonov, D., Schütz, D., Ulewicz, S., Vogel-Heuser, B., 2014. Towards industrial application of model-driven platform-independent PLC programming using UML. In: IECON 2014-40th Annual Conference of the IEEE Industrial Electronics Society. IEEE, pp. 2638–2644.
Urbas, L., Krause, A., Ziegler, J., 2012. Process Control Systems Engineering. Oldenbourg Industrieverlag GmbH Munich.
Voelter, M., Bettin, J., 2004. Patterns for Model-Driven Software-Development. In: EuroPLoP, pp. 525–560.
Vogel-Heuser, B., 2014. Usability experiments to evaluate UML/SysML-based model driven software engineering notations for logic control in manufacturing automation. J. Softw. Eng. Applic. 7 (11), 943.
Vogel-Heuser, B., Fay, A., Schaefer, I., Tichy, M., 2015. Evolution of software in automated production systems: challenges and research directions. J. Syst. Softw. 110, 54–84.
Vogel-Heuser, B., Schütz, D., Frank, T., Legat, C., 2014. Model-driven engineering of manufacturing automation software projects–A sysML-based approach. Mechatronics 24 (7), 883–897.
Vogel-Heuser, B., Witsch, D., Katzke, U., 2005. Automatic code generation from a UML model to IEC 61131-3 and system configuration tools. In: 2005 International Conference on Control and Automation, 2. IEEE, pp. 1034–1039.
Voinov, A., Yang, C., Vyatkin, V., 2017. Automatic generation of function block systems implementing hmi for energy distribution automation. In: 2017 IEEE 15th International Conference on Industrial Informatics (INDIN), pp. 706–713. doi:10.1109/INDIN.2017.8104859.
Vyatkin, V., 2013. Software engineering in industrial automation: state-of-the-art review. IEEE Trans. Ind. Inf. 9 (3), 1234–1249.
Wang, R., Gu, M., Song, X., Wan, H., 2009. Formal specification and code generation of programable logic controllers. In: 2009 14th IEEE International Conference on Engineering of Complex Computer Systems. IEEE, pp. 102–109.
Wenger, M., Zoitl, A., Sunder, C., Steininger, H., 2009. Transformation of iec 61131-3 to iec 61499 based on a model driven development approach. In: 2009 7th IEEE International Conference on Industrial Informatics. IEEE, pp. 715–720.
Wiriyacoonkasem, S., Esterline, A.C., 2000. Adaptive learning expert systems. In: Proceedings of the IEEE SoutheastCon 2000.'Preparing for The New Millennium'(Cat. No. 00CH37105). IEEE, pp. 445–448.
Witsch, D., Ricken, M., Kormann, B., Vogel-Heuser, B., 2010. PLC-statecharts: an approach to integrate umlstatecharts in open-loop control engineering. In: 2010 8th IEEE International Conference on Industrial Informatics. IEEE, pp. 915–920.
Wohlin, C., 2014. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: Proceedings of the 18th international conference on evaluation and assessment in software engineering. Citeseer, p. 38.
Yang, C., Dubinin, V., Vyatkin, V., 2017. Ontology driven approach to generate distributed automation control from substation automation design. IEEE Trans. Ind. Inf. 13 (2), 668–679. doi:10.1109/TII.2016.2634095.
Yang, C., Dubinin, V., Vyatkin, V., 2020. Automatic generation of control flow from requirements for distributed smart grid automation control. IEEE Trans. Ind. Inf. 16 (1), 403–413. doi:10.1109/TII.2019.2930772.
Yang, C.-H., Vyatkin, V., Pang, C., 2014. Model-driven development of control software for distributed automation: a survey and an approach. IEEE Trans. Syst. Man Cybernet. 44 (3), 292–305.
Yim, S.Y., Ananthakumar, H.G., Benabbas, L., Horch, A., Drath, R., Thornhill, N.F., 2006. Using process topology in plant-wide control loop performance assessment. Comput. Chem. Eng. 31 (2), 86–99.
Alvarez, M.L., Sarachaga, I., Burgos, A., Estevez, E., Marcos, M., 2018. A methodological approach to model-driven design and development of automation systems. IEEE Trans. Autom. Sci. Eng. 15, 67–79.
Drath, R., Fay, A., Schmidberger, T., 2006. Computer-aided design and implementation of interlock control code, in: IEEE conference on computer aided control system design. IEEE 2653–2658.
Estevez, E., Marcos, M., Orive, D., 2007. Automatic generation of plc automation projects from component-based models. The Int. J. Adv. Manuf. Technol. 35, 527–540.
Grüner, S., Weber, P., Epple, U., 2014. Rule-based engineering using declarative graph database queries, in: 2014 12th IEEE international conference on industrial informatics (INDIN). IEEE 274–279.
Hästbacka, D., Vepsäläinen, T., Kuikka, S., 2011. Model-driven development of industrial process control applications. J. Syst. Softw. 84, 1100–1113.
Koltun, G., Kolter, M., Vogel-Heuser, B., 2018. Automated Generation of Modular PLC Control Software from P&ID Diagrams in Process Industry. In: 2018 IEEE International Systems Engineering Symposium (ISSE). IEEE, pp. 1–8.
Lukman, T., Godena, G., Gray, J., Hericko, M., Strmcnik, S., 2013. Model-driven engineering of process control software - beyond device-centric abstractions. Control Eng. Pract. 21, 1078–1096.
Schumacher, F., Schröck, S., Fay, A., 2013. Tool Support for an Automatic Transformation of GRAFCET Specifications into Iec 61131-3 Control Code. In: 2013 IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA). IEEE, pp. 1–4. F. Schumacher, S. Schröck, A. Fay, Transforming hierarchical concepts of GRAFCET into a suitable petri net formalism, IFAC Proceedings Volumes 46 (2013) 295–300. F. Schumacher, A. Fay, Formal representation of GRAFCET to automatically generate control code, Control Engineering Practice 33 (2014) 84–93. R. Julius, M. Schürenberg, F. Schumacher, A. Fay, Transformation of GRAFCET PLC code including hierarchical structures, Control Engineering Practice 64 (2017) 173–194
Steinegger, M., Zoitl, A., 2012. Automated Code Generation for Programmable Logic Controllers Based on Knowledge Acquisition from Engineering Artifacts: Concept and Case Study. In: Proc. of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012). IEEE, pp. 1–8. M. Steinegger, M. Melik-Merkumians, J. Zajc, G. Schitter, Automatic generation of diagnostic handling code for decentralized plc-based control architectures, in: Proc. IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA), IEEE, 2016, pp. 1–8. M. Steinegger, M. Melik-Merkumians, G. Schitter, Ontology-based framework for the generation of interlock code with redundancy elimination, in: Proc. 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), IEEE, 2017, pp. 1–5
Thramboulidis, K., Frey, G., 2011. An MDD Process for IEC 61131-Based Industrial Automation Systems. In: ETFA2011. IEEE, pp. 1–8.
Vogel-Heuser, B., Schütz, D., Frank, T., Legat, C., 2014. Model-driven engineering of manufacturing automation software projects - a sysML-based approach. Mechatronics 24, 883–897.
Vogel-Heuser, B., Witsch, D., Katzke, U., 2005. Automatic Code Generation from a UML Model to IEC 61131-3 and System Configuration Tools. In: 2005 International Conference on Control and Automation, volume 2. IEEE, pp. 1034–1039.
Witsch, D., Vogel-Heuser, B., 2009. Close Integration between UML and IEC 61131-3: New Possibilities through Object-oriented Extensions. In: 2009 IEEE Conference on Emerging Technologies & Factory Automation. IEEE, pp. 1–6. D. Witsch, M. Ricken, B. Kormann, B. Vogel-Heuser, PLC-Statecharts: An approach to integrate UML statecharts in open-loop control engineering, in: 2010 8th IEEE International Conference on Industrial Informatics, IEEE, 2010, pp. 915–920. D. Witsch, B. Vogel-Heuser, PLC-Statecharts: An approach to integrate UML-statecharts in open-loop control engineering - Aspects on behavioral semantics and model-checking, IFAC Proceedings Volumes 44 (2011) 7866–7872

Heiko Koziolek is a Senior Principal Scientist at ABB Corporate Research in Ladenburg, Germany. He leads research projects on software architectures for industrial process control and coordinated different projects on sustainable architectures. He studied computer science at the Carl von Ossietzky University of Oldenburg and obtained a PhD in software engineering in 2008. His research is concerned with performance prediction for software systems as well as analyzing software maintainability. He serves on the program committee of leading conferences on software engineering (ICSE) and software architecture (ICSA) and as an Associate Editor for Elsevier "Journal of Systems and Software".

Andreas Burger is a Research Team Manager at ABB Corporate Research in Ladenburg, Germany. He leads research projects on software architecture and software product lines in industrial automation. He studied computer science at the University of Tbingen, Germany, and graduated with a PhD in 2015 while working for Forschungszentrum Informatik (FZI) in Karlsruhe, Germany. He has experience in embedded system development and the automotive domain.

Marie Platenius-Mohr is a Scientist at ABB Corporate Research in Ladenburg, Germany. She is conducting research in the industrial automation domain. Her research interests are software architecture, information modeling, internet-of-things, and model-driven development. She studied computer science at the University of Paderborn, Germany, and graduated with a PhD in 2016. Afterwards, she worked as a PostDoc in the Collaborative Research Center On-the-Fly Computing and joined ABB in 2018.

Raoul Jetley is a Senior Principal Scientist at ABB Corporate Research in Bangalore, India. He leads cross-center research projects to improve the efficiency of industrial power systems and software. He studied computer science at the University of Mumbai and North Carolina State University and graduated with a PhD in 2006. He has developed program analysis tools for industrial automation systems and has more than 15 years of experience in industry and academic environments.