Preprints of the 19th World Congress
The International Federation of Automatic Control
Cape Town, South Africa. August 24-29, 2014

Identification of Design Patterns for IEC 61131-3 in Machine and Plant Manufacturing

J. Fuchs, S. Feldmann, C. Legat, B. Vogel-Heuser

Technische Universität München, Faculty of Mechanical Engineering, Institute of Automation and Information Systems, D–85748 Garching
e-mail: {fuchs, feldmann, legat, vogel-heuser}@ais.mw.tum.de

Abstract: Industrial plant software is implemented in the programming languages of IEC 61131-3. As plant software needs to perform many tasks, it is often highly complex and typically characterized by a monolithic structure. Whereas for high-level programming languages, commonly occurring problems are solved using design patterns, such general reusable solution alternatives are not yet available for IEC 61131-3. Thus, an approach for statically analysing the plant software and visualizing the software units' complexity and interconnectedness is proposed in this paper. Furthermore, basic software design patterns are introduced and, subsequently, their appearance within plant software is evaluated using industrial code and interviews with experts. By that, a first step towards providing design patterns for IEC 61131-3 is made.

Keywords: Programmable logic controllers, Software metrics, Machine code, Pattern identification, Programming approaches

1. INTRODUCTION

Machines and plants in the industrial automation domain are mostly controlled by Programmable Logic Controllers (PLCs). The plant code is implemented in the signal-oriented programming languages of IEC 61131-3 and needs to perform many tasks, e.g. system functionality, safety requirements or human-machine interface (Lucas and Tilbury, 2004), resulting in highly complex software, which is typically characterized by a monolithic structure. Hence, plant software structures are often difficult to comprehend.

In object-oriented programming, commonly occurring problems are solved using design patterns (Gamma et al., 1994) that enable increased reuse and modularity of the code and, thus, enhanced software quality and comprehensibility (Thramboulidis and Frey, 2011). However, such general reusable solutions for common problems are not yet available in the classical languages of IEC 61131-3. Nevertheless, plant software engineers often reuse solutions for specific problems, e.g. error handling and modes of operation, which could possibly be abstracted and reused as solution alternatives for various problem types. If these software solution alternatives were identified, named and clustered, a first step towards providing software design patterns in the industrial automation domain would be provided and, hence, the plant software's reusability, modularity as well as comprehensibility could be enhanced. Thus, this paper aims at statically analysing the software structure by considering data exchange between software units as well as code complexity. Hence, software design patterns can be identified providing the basis for analysing common software solutions in PLC programming. By that, modularity and – as a consequence – reusability of software parts can be increased leading to lower engineering costs and reduced time-to-market.

The remainder of this paper is structured as follows: In the next section, an overview on IEC 61131-3 language elements is shown. Subsequently, related work in the fields of plant code analysis and design patterns is discussed. The approach for analysing the code structure of plant software is presented and design patterns for IEC 61131-3 code are introduced in sections 4 and 5, respectively. The appearance of the design patterns within plant code is evaluated based on interviews with industrial experts in section 6. Finally, the paper is concluded by a summary and an outlook on future work in section 7.

2. LANGUAGE ELEMENTS OF IEC 61131-3

Despite efforts towards including object-oriented programming aspects within IEC 61131-3 (IEC, 2013), the standard in its current version has not yet been fully established in industry. Thus, it is focused on IEC 61131-3 without object-oriented extension (IEC, 2009), which is mostly used within state of the art industrial applications (Thramboulidis and Frey, 2011).

The IEC 61131-3 standard contains both textual, i.e. Structured Text (ST) and Instruction List (IL), as well as graphical programming languages, i.e. Function Block Diagram (FBD), Ladder Diagram (LD) and Sequential Function Chart (SFC). Furthermore, the standard defines three types of Program Organization Units (POUs) for structuring and reusing the PLC code. A *Program* (PRG) represents the assembly of logical elements necessary for the machine or process controlled by a PLC. One PRG is the main program and thus provides the entry point into the plant code. *Function blocks* (FBs), which calculate

output values based on input and persistent internal values, and *Functions* (FCs), which yield the calculation of a result value based solely upon input values, can be combined within a PRG. Furthermore, PRG and FB *types* are instantiated during design time and hold their data memories within PRG and FB *instances* during run time. Thus, data exchange is realized between POU instances, namely PRG and FB instances as well as FCs.

3. RELATED WORK

Contrary to approaches such as model-checking, e.g. (Soliman et al., 2012), which requires a model of the system for analysing and verifying its behaviour, code analysis aims at analysing the implemented software code and is mainly applied to ensure software quality by identifying defects in software programs. In contrast to dynamic code analysis, which aims at executing the program with the intention to observe its behaviour, static code analysis addresses "automatic methods to reason about runtime properties of program code without actually executing it" (Emanuelsson and Nilsson, 2008). Thereby, a program's structure and elements or the set of possible program states are analysed and, thus, the program behaviour is approximated (Artho and Biere, 2005). Although static code analysis is successfully applied to several programming languages and environments, e.g. Lint for C (Johnson, 1978) and FindBugs for Java (Ayewah et al., 2008), IEC 61131-3 is not yet supported sufficiently (Angerer et al., 2013). Up to now, only tools supporting parts or specific languages of IEC 61131-3 are provided, e.g. CoDeSys Static Analysis [1], logi.Lint [2] by Logicals and PLC Checker [3] by Itris. Nevertheless, in (Prähofer et al., 2012) the benefits of static code analysis for IEC 61131-3 software quality improvement are highlighted and an approach for improving compliance with programming conventions and guidelines is proposed, e.g. by identifying incorrect naming conventions, deviating program element complexity or detecting bad code fragments. Thereby, analysis rules are used to express the search criteria. Furthermore, call graphs and points-to analyses were integrated extending the approach for further analyses, e.g. concurrent access to shared variables and coupling of subsystems (Angerer et al., 2013). However, the focus of these analysis approaches is put on the identification of software errors and defects; identification and visualization of commonly occurring designs within IEC 61131-3 compliant plant code is not addressed.

In order to identify and develop solutions for commonly occurring problems in software engineering, reusable design patterns are used as well-documented building blocks (Sanz and Zalewski, 2003). Despite a multitude of work within high-level programming languages, especially within object-oriented languages such as the work by Gamma et al. (1994), design patterns have been scarcely considered within the PLC programming domain. First efforts towards evaluating different methods of implementing logic control algorithms within IEC 61131-3 were conducted (Hajarnavis and Young, 2008), but specific patterns have not been derived yet. However, design patterns within control systems engineering would address a multitude of issues such as controller design, architectural design as well as implementation aspects (Sanz and Zalewski, 2003). Patterns have been especially investigated within the emerging model-based design of automation software, e.g. using Unified Modeling Language (Fantuzzi et al., 2009). Thereby, the authors introduced design patterns for solving typical problems such as alarm handling and motion control using state charts and guidelines for implementing these patterns in IEC 61131-3 programming languages (Bonfè et al., 2013). Preschern et al. (2012) introduce patterns for improving system flexibility and maintainability. For IEC 61499-based applications (IEC, 2011), common solutions and guidelines were proposed for hierarchical automation solutions (Zoitl and Prähofer, 2013), failure management (Serna et al., 2010) and portable automation projects (Dubinin and Vyatkin, 2012). Even software design patterns for IEC 61499 programs, e.g. Distributed Application, Proxy and Model-View-Controller, were defined (Christensen, 2000) and evaluated (Stromman et al., 2005). However, although IEC 61499 runtimes on state of the art controllers exist (Vyatkin, 2011), "IEC 61499 has a long way in order to be seriously considered by the industry" (Thramboulidis, 2013).

As none of the approaches mentioned above is capable of sufficiently and comprehensibly analysing IEC 61131-3 programs for identifying common software solutions, the combination of static code analysis with a comprehensible visualization of the analysed code would provide a first step towards identifying standard solutions for commonly occurring problems in IEC 61131-3 programs.

4. ANALYSIS OF IEC 61131-3 CONTROL SOFTWARE

Currently, there exist few possibilities to structure plant software as IEC 61131-3 code structure is mainly defined through data exchange between POUs. The software is composed of many POUs, i.e. PRG and FB instances as well as FCs, which are connected through calls that form the basis of the code structure.

4.1 Calls and instances of POUs in IEC 61131-3

Calls form the basis of the plant code structure by connecting POU instances; thus, only POU instances that are called by other instances are executed during run time. A PRG instance can call FB instances and FCs, and a FB instance can call further FB instances and FCs. FCs, by contrast, can only call other FCs. The calling and called types of POU, namely PRG type, FB type or FC, are hence dependent, because modifying one of these POU types may affect other ones. This basic call structure is often used to design the plant code and is thus commonly comprehensible for the programmer.

4.2 The role of data exchange in IEC 61131-3

During a call, *direct data exchange* (DDE) is implemented by passing variables between POU instances. Furthermore, *indirect data exchange* (IDE) is implemented between POU instances, as these can write values into and read values from global variables. Consequently, a hidden data exchange structure is built additionally, which makes the plant code structure even more complex.

[1] http://store.codesys.com/codesys-static-analysis.html
[2] http://www.logicals.com/products/logi.LINT/
[3] http://www.plcchecker.com/

4.3 Visualization of the software structure

By composing calls between POUs as well as data exchange, the entire structure of the software results. In order to enable analysis and examination of the software plant code and the identification of design patterns, the code complexity and the interconnectedness between software units need to be visualized within a directed labelled graph (Fig. 1).

A node of the graph represents a POU type. The node's diameter thereby is directly proportional to its complexity, which is e.g. identified using the POU's Lines of Code (LoC); thus, FB1 is more complex than FB3 However, further metrics such as the ones proposed by Lucas and Tilbury (2005) may be used to determine a POU's complexity. The directed edges between nodes of the graph represent a connection between two POU types by data exchange. In order to visualize the interconnectedness between two POUs, the edges' lengths thereby are indirectly proportional to the data exchange intensity, which is e.g. determined using the number of variables committed between the POUs. To distinguish between data exchange via POU calls and data exchange via global variables, the visualization approach provides a DDE and an IDE view.

The DDE view (Fig. 1, no. I) represents the unidirectional data exchange through calls between POUs. Thus, an edge within this view denotes that a POU calls another POU, whereby the edge points from the calling to the called POU. As the edges' lengths are indirectly proportional to the exchange intensity, thus FB2 and FC1 are more strongly connected than PRG1 and FB1. Furthermore, self-calls of POUs are represented through edges from and to the POU, e.g. for FC2. In this view, POUs which are not connected to other POUs are not called and thus are not executed within the PLC code.

The IDE view (Fig. 1, no. II) represents the unidirectional and bidirectional data exchange through global variables. Thus, an edge within this view denotes that a POU writes into a global variable and another POU reads from this variable, whereby the edge points from the writing POU to the reading POU. POUs that both read from and write into the same global variable are connected via bidirectional edges. POUs not connected to other POUs in this view do not exchange data via global variables. The proposed visualization can be used for analysing the plant code. In the following section, common design patterns within plant code are identified and visualized.

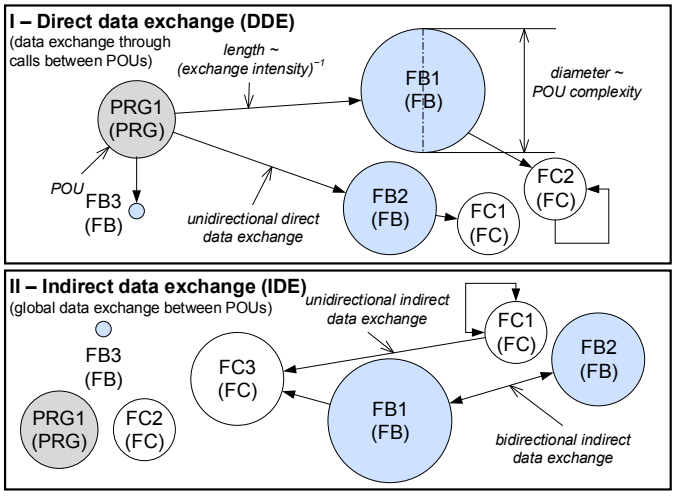
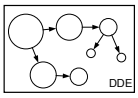
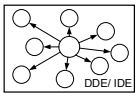
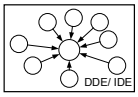
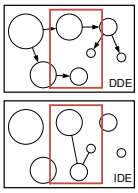
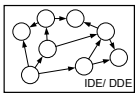


Fig. 1. Overview of the code visualization

Table 1. Identified design patterns

Name	Description	Visualization	Characteristics
Tree	POUs are called in a Tree structure		One incoming edge per node
Cornflower	One POU calls many other POUs		One node points to many other nodes
Central Unit	Many POUs call one POU		Many nodes point to same node
Cuckoo	Data exchange between POUs in IDE view but no data exchange in DDE view		Connection between nodes in IDE view but not in DDE view
Uniform Complexity	Complexity of all POUs is nearly the same		Uniform node diameters

5. DESIGN PATTERNS IN IEC 61131-3

Although programmers in industrial automation do not engineer software using software patterns, they build their software structure according to specific criteria, e.g. companies' programming guidelines or personal experiences, without being explicitly aware of these solution alternatives. Patterns resulting from these alternatives can be identified by evaluating calls and different kinds of data exchange, namely DDE and IDE, as presented in section 4. Using the approach of visualizing the complexity and interconnectedness of POUs, five elementary patterns, which are introduced in this section, can be identified (Table 1). In the following, these patterns are explained in detail using their visualization, interpretation and characteristics making them measureable.

5.1 Tree

Within the software pattern *Tree*, a POU should only be called if it has not been called by another POU within the Tree structure yet. Hence, any two nodes, i.e. calling and called POU, are connected by exactly one directed edge and each node has only a single incoming edge, i.e. each POU type is only called by one POU type. If each branch represents an accumulation of dependent POUs, a Tree offers the possibility to avoid too many dependencies in between POUs and thus encapsulates functionality. In case different branches of a Tree use the same POU,

the software structure deviates from the hierarchical Tree structure. On the one hand, reuse of a POU is desired and on the other hand, if a POU, which is used by multiple branches, is modified, undesired behaviour for dependent POUs may arise. The pattern Tree can be identified in DDE, whereby the edges' length and the nodes' diameter do not correlate to the pattern.

5.2 Cornflower

Within the pattern *Cornflower*, a single POU calls multiple dependent POUs; thus, directed edges point from a central node to circularly arranged nodes within the directed graph, whereby both nodes' diameter and edges' length do not contribute to the pattern. A special case of the pattern Cornflower arises, if each directed edge from the central POU to dependent POUs has the same length. Using this special case, standardized interfaces between POUs can be implemented, e.g. to exchange process data within IDE. Cornflowers may be used within DDE and IDE as well as within a combination of both data exchange types.

5.3 Central Unit

In contrast to the Cornflower pattern, multiple dependent POUs call a single central POU within the *Central Unit* pattern; hence, directed edges point from circularly arranged nodes to a central node within the directed graph. Analogously to the Cornflower, both nodes' diameters and edges' lengths do not have a correlation to the pattern. This pattern may be used if a central data aggregation point for further data commitment is necessary, e.g. for human-machine interfaces. It may also occur in the case that multiple POUs call commonly used functionality, e.g. conversion functions. However, the application of the Central Unit depends on the type of data exchange: Within DDE, the pattern increases reuse of a single POU, whereas within IDE, data aggregation is implemented.

5.4 Cuckoo

If a POU is called, data is exchanged via DDE. However, further data exchange may be implemented by IDE, making the analysis of the entire software structure necessary. Within DDE, code structure can seem very well organized, but if IDE is analysed additionally, the global data exchange often shows different connections between POUs ruining the software structure intended by the programmer. Hence, patterns may also be identified by analysing the entire data exchange, i.e. analysing IDE and DDE at the same time. The pattern *Cuckoo* considers the interaction of DDE and IDE and occurs if a POU exchanges data with another POU via IDE but not via DDE. If the pattern Cuckoo is identified, the programmer has to determine whether the global data exchange has any benefit and whether it can be resolved using another solution.

5.5 Uniform Complexity

The patterns introduced within the preceding subsections are characterized by the connections between POUs. Nevertheless, further patterns can be identified considering the POUs' complexity. The pattern *Uniform Complexity* aims at a uniform POU complexity within the plant code. Thus, all nodes within the directed graph have a similar diameter. A uniform code distribution of POU complexity may be desired in order to increase comprehensibility and reuse of the software code. However, the challenge is to find the appropriate POU size in between having many small POUs making it difficult to reuse and combine POUs and few large ones constraining flexibility (Jazdi et al., 2011).

6. EVALUATION

To provide an appropriate evaluation of the identified patterns and to prove the feasibility of the proposed approach, different automation plant codes were analysed (Table 2). In order to give an impression of the complexity of the different codes, the Lines of Code (LoC) value was evaluated. The graphical languages LD, FBD and SFC were converted into the textual languages IL and ST in order to measure the LoC value in a similar manner. There are different LoC values from rather low complexity, e.g. lab demonstrators (Table 2, codes #5 and #6), to high complexity, e.g. industrial plants such as code #2 and #4. Nevertheless, in each plant code evaluated, the patterns introduced in section 5 can be identified. As the same patterns either can fulfill the intentions of a programmer or can cause undesired plant code structure, the aims of the code were discussed with the developers of the analysed PLC code within expert interviews. The results are abstracted due to confidentiality agreements.

6.1 Tree

The pattern Tree, as introduced in section 5, is used for the entire software structure by arranging POUs in a strictly hierarchical manner resulting in a modular structure. Thereby, every branch of the Tree represents a single module, as was identified e.g. in code #3. In some cases, the Tree was repeatedly used by being arranged as sub-branches of the main branch, namely sub-modules. The application of the Tree pattern increased the opportunity to extract modules, as there are few dependencies between POUs of other modules, as e.g. implemented in parts of code #2. There can be derivations from the strictly modular structure of the Tree if common functionalities are used by more than one module. In some cases, this can result in unexpected effects on the plant code behaviour, e.g. if a commonly used POU is modified, dependent POUs may behave differently than expected. By analysing the Tree's structure, strongly dependent POUs can be identified using the proposed approach. Hence, the Tree pattern and its derivations can help to draw conclusions concerning modularity and modifiability of the code structure.

6.2 Cornflower

The entry into the plant software is often a Cornflower. Furthermore, it can be part of a Tree, thus providing the entry into a module which occurs in almost each code analysed in the context of this paper. The interfaces of the POUs that are connected to the central POU are standardized; hence, an easy change of POUs is possible. In IDE the pattern Cornflower was used for a standardized

Table 2. Overview of the evaluated PLC code

	Industrial Plants				Demonstrators		
No.	1	2	3	4	5	6	7
Application Domain	PT	PT	MT	MT	PT	MT	MT
Area	PU	P	M	PU	M	PU	M
Lines of Code	332k	234.5k	170k	1,246k	41.9k	24.9k	1.3k
No. of POUs	581	315	208	351	199	294	5
No. of DDE | no. of IDE	1,123 | 2,082	407 | 789	n.a.	626 | 1061	316 | 145	618 | 365	1 | 3
Significant use of patterns	alarm collectors with Central Units, Cornflowers for signal propagation	sectional Trees, which are reused as modules; many Cuckoos	in DDE a strict Tree structure, many Cuckoos	very interconnected, many Cornflowers and Central Units, only DDE	almost exclusively DDE, composed of Tree sections, Cornflowers and Central Units	Uniform Complexity, very interconnected	few POUs, only pattern Cornflower

(PT – process technology, MT – manufacturing technology, P – plant, PU – plant unit, M – machine))

transfer of signals, as e.g. used in code #1. Thus, each module receives all signals and influences all actuators enabling simple adding of new modules.

6.3 Central Unit

A Central Unit often fulfils a standardized or commonly needed functionality, e.g. alarm handling or calculation functions in code #4. Changing a Central Unit can have an impact on the behaviour of many other POUs. Within DDE, this pattern can be identified easily and occurs in many cases exposing intensive dependencies in between POUs as was confirmed . Furthermore, the Central Unit pattern is used within IDE for communication between modules. However, expert interviews showed that this is often unintended, because this results in huge interfaces. A possible solution is e.g. to concentrate data exchange in one POU. Another typical application of the Central Unit within IDE is an alarm collector. Each relevant POU communicates an alarm directly to the alarm collector – implemented as a Central Unit – whereby the other POUs can be directly informed, as e.g. used in code #1.

6.4 Cuckoo

There are only a few cases in which the pattern Cuckoo is desired although it occurs in almost every analysed code. To get a well-structured code, these undesired global data exchanges without direct calls have to be dissolved. Often, too many Cuckoos prevent an adequate modularization and reuse of parts of the code. This was confirmed in the interviews with the programmers of almost each code.

6.5 Uniform Complexity

On the one hand, significantly huge POUs can be identified in almost each analysed code. Amongst others, a reason could be the use of SFCs, which results in a high LoC value through conversion into text, especially in the process technology domain, i.e. codes #1, #2 and #5. Expert interviews showed that changing the code has no benefit as encapsulating parts of the functionality within smaller POUs would lead to increased complexity. In other cases, huge POUs contain much functionality making it hard to detect errors, especially by someone who did not develop the code. Discussions with experts exposed the need to split the functionality and encapsulate parts of it within smaller POUs. On the other hand, in some cases the functionality is distributed into many small POUs, in which cases the entire behaviour of the code is difficult to comprehend. Concluding, a suitable POU size must be found to support the comprehensibility of the code structure and to fit the application-specific requirements.

6.6 Entire Structure

The patterns introduced in this paper can be composed to an entire plant code structure. In industrial code, there often exist parts of Trees that form modules, Cornflowers used for structuring calls, Central Units that are used to exchange data in between modules and Cuckoos mostly undesired within the code. The most significant findings concerning the patterns' intentions are summarized for each PLC code in Table 2. In code #1, the patterns are used to solve recurrent requirements such as data exchange between units. Reuse is realised within code #2 by implementing sectional Trees within the code that form modules. In some cases, reuse of a module is limited because too many Cuckoos produce interdependencies. An overall strict Tree pattern was identified in code #3, but there also occur many Cuckoos similar to code #2. No reuse was planned initially for code #4; hence, the DDE structure is mostly interconnected. However, also patterns such as Central Units or Cornflowers were identified, which are used for data exchange and hierarchical structuring of functionality. There are almost no connections between POUs within IDE; hence, Cuckoos do not appear within the code. Nearly the same structure exists in code #5, but DDE is well-structured through patterns, e.g. Tree and Cornflower. The complexity in code #6 is distributed equally, but POUs are mostly interconnected. Code #7 corresponds to a demonstration plant and is thus quite small; nevertheless, the pattern Cornflower was identified.

7. CONCLUSION

In this paper, a first step towards identifying typical solution alternatives for commonly occurring problems within automation software engineering, namely plant software patterns for IEC 61131-3, was proposed. Based on a detailed analysis of plant code, an approach for analysing and visualizing IEC 61131-3 plant code was introduced. Using this approach, dependencies and encapsulations of software units can be analysed and, hence, software design patterns can be derived. Using industrial applications, the appearance of the plant code design patterns was evaluated and discussed regarding their benefits and programmer's intention with industrial experts.

Future work will include the extension of the proposed patterns towards a more detailed pattern library in order to increase modularity, reuse as well as software quality and comprehensibility. Further research will be investigated towards identifying appropriate metrics that support rating and, thus, evaluating software quality and reuse. The existing software tool will be extended by these metrics and further patterns; thus, a support system for identifying reusable artifacts can be provided. Moreover, as slight differences in software complexity are sometimes difficult to recognize in the graphical representation, additional mechanisms will be integrated within the tool to support the engineer. To evaluate these enhancements, benchmarks of different industrial plants in addition to expert interviews can be done. By doing this evaluation an illustration of specific properties of the analyzed code is aspired.

REFERENCES

Angerer, F., Prähofer, H., Ramler, R., and Grillenberger, F. (2013). Points-To Analysis of IEC 61131-3 Programs: Implementation and Application. In *18th IEEE Int. Conf. Emerg. Technol. Fact. Autom.* Cagliari, Italy.

Artho, C. and Biere, A. (2005). Combined Static and Dynamic Analysis. *Electron. Notes Theor. Comput. Sci.*, 131, 3–14.

Ayewah, N., Hovemeyer, D., Morgenthaler, J.D., Penix, J., and Pugh, W. (2008). Using Static Analysis to Find Bugs. *IEEE Softw.*, 25(5), 22–29.

Bonfè, M., Fantuzzi, C., and Secchi, C. (2013). Design patterns for model-based automation software design and implementation. *Control Eng. Pract.*, 21(11), 1608–1619.

Christensen, J.H. (2000). Design patterns for systems engineering in IEC 61499. In *Fachtagung Verteilte Autom.*, 63–71. Magdeburg, Germany.

Dubinin, V.N. and Vyatkin, V. (2012). Semantics-Robust Design Patterns for IEC 61499. *IEEE Trans. Ind. Informatics*, 8(2), 279–290.

Emanuelsson, P. and Nilsson, U. (2008). A Comparative Study of Industrial Static Analysis Tools. *Electron. Notes Theor. Comput. Sci.*, 217, 5–21.

Fantuzzi, C., Bonfè, M., and Secchi, C. (2009). A Design Pattern for Model Based Software Development for Automatic Machinery. In *13th IFAC Symp. Inf. Control Probl. Manuf.*, 1429–1434. Moscow, Russia.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, US-MA.

Hajarnavis, V. and Young, K. (2008). An Assessment of PLC Software Structure Suitability for the Support of Flexible Manufacturing Processes. *IEEE Trans. Autom. Sci. Eng.*, 5(4), 641–650.

IEC (2009). *Programmable Logic Controllers Part 3: Programming Languages. IEC Standard 61131-3.*

IEC (2011). *Function Blocks. IEC Standard 61499.*

IEC (2013). *Programmable Logic Controllers Part 3: Programming Languages. IEC Standard 61131-3.*

Jazdi, N., Maga, C.R., and Göhner, P. (2011). Reusable Models in Industrial Automation: Experiences in Defining Appropriate Levels of Granularity. In *18th IFAC World Congr.*, 9145–9150. Milano, Italy.

Johnson, S. (1978). Lint, a C Program Checker. Technical report, Bell Laboratories.

Lucas, M.R. and Tilbury, D.M. (2004). The Practice of Industrial Logic Design. In *Am. Control Conf.*, 1350–1355. Boston, US-MA.

Lucas, M. and Tilbury, D. (2005). Methods of measuring the size and complexity of PLC programs in different logic control design methodologies. *Int. J. Adv. Manuf. Technol.*, 26(5-6), 436–447.

Prähofer, H., Angerer, F., Ramler, R., Lacheiner, H., and Grillenberger, F. (2012). Opportunities and challenges of static code analysis of IEC 61131-3 programs. In *17th IEEE Int. Conf. Emerg. Technol. Fact. Autom.*, 1–8. Kraków, Poland.

Preschern, C., Kajtazovic, N., and Kreiner, C. (2012). Applying Patterns to Model-Driven Development of Automation Systems: An Industrial Case Study. In *17th Eur. Conf. Pattern Lang. Programs*, 1–6. Kloster Irsee, Germany.

Sanz, R. and Zalewski, J. (2003). Pattern-based control systems engineering. *IEEE Control Syst. Mag.*, 23(3), 43–60.

Serna, F., Catalan, C., Blesa, A., and Rams, J.M. (2010). Design patterns for Failure Management in IEC 61499 Function Blocks. In *15th IEEE Int. Conf. Emerg. Technol. Fact. Autom.*, 1–7. Bilbao, Spain.

Soliman, D., Thramboulidis, K., and Frey, G. (2012). Function Block Diagram to UPPAAL Timed Automata Transformation Based on Formal Models. In *14th IFAC Symp. Inf. Control Probl. Manuf.*, 1653–1659. Bucharest, Romania.

Stromman, M., Sierla, S., and Koskinen, K. (2005). Control Software Reuse Strategies with IEC 61499. In *10th IEEE Int. Conf. Emerg. Technol. Fact. Autom.*, 749–756. Catania, Italy.

Thramboulidis, K. (2013). IEC 61499 as an Enabler of Distributed and Intelligent Automation: A State-of-the-Art Review. *J. Eng.*, 1–9.

Thramboulidis, K. and Frey, G. (2011). Towards a Model-Driven IEC 61131-Based Development Process in Industrial Automation. *J. Softw. Eng. Appl.*, 04(04), 217–226.

Vyatkin, V. (2011). IEC 61499 as Enabler of Distributed and Intelligent Automation: State-of-the-Art Review. *IEEE Trans. Ind. Informatics*, 7(4), 768–781.

Zoitl, A. and Prähofer, H. (2013). Guidelines and Patterns for Building Hierarchical Automation Solutions in the IEC 61499 Modeling Language. *IEEE Trans. Ind. Informatics*, 9(4), 2387–2396.