Managing Variability and Reuse of Extra-functional Control Software in CPPS

Birgit Vogel-Heuser*, Juliane Fischer*, Dieter Hess**, Eva-Maria Neumann*, Marcus Würr***
*Institute of Automation and Information Systems, Technical University of Munich, Garching near Munich, Germany
{vogel-heuser | juliane.fischer | eva-maria.neumann}@tum.de
**3S-Smart Software Solution GmbH, Kempten, Germany
d.hess@codesys.com
*** Schneider Electric Automation GmbH, Marktheidenfeld, Germany
marcus.wuerr@se.com

***Abstract*—Cyber-Physical Production Systems (CPPS) are long-living and variant-rich systems. Challenges and trends in the context of Industry 4.0 such as a high degree of customization, evolution, and different disciplines involved, e.g., mechanics, electrics/electronics and software, cause a high amount of variability. Mastering the variability of functional control software, e.g., different control variants of an actuator type, is itself a challenge in the development and reuse of CPPS software. This task becomes even more complex when considering the variability of human-machine interface (HMI) software and extra-functional software such as operating modes, diagnosis or fault handling. Moreover, the interdependencies between functional, extra-functional and HMI software pose an additional challenge for variability management and the planned reuse of these software parts. This paper illustrates the challenges in documenting the dependencies of these software parts including their variability using family models. Additionally, the current state-of-practice in industry, which was derived in questionnaires and interview studies, is shown and compared to the potential of increasing the software's reusability and thus its flexibility in the context of Industry 4.0 by concepts using the object-oriented extension of IEC 61131-3.**

Keywords— Cyber-Physical Production Systems, object-oriented control software, extra-functional software, variability, reuse

I. Introduction and Motivation

Cyber-Physical Production Systems (CPPS) are mechatronic, variant-rich and long-living systems connected to digital networks to use globally available data and services. An increasing amount of their functionality is realized by control software [1]. Thereby, software variants arise due to shortened production cycles, continuously changing requirements and variable automation hardware [2] stressing the need for systematic software reuse to enable flexible production in the context of Industry 4.0. Apart from pure functional software, extra-functional software implementing communication tasks, diagnosis, fault handling and operating modes is an essential part of control software: Extra-functional code makes up around 50-75% of industrial control code [3] causing complexity in the control software [4] and thus reducing maintainability and consequently impacting long term cost and the ability to innovate. The dependencies between functional and extra-functional code make variant management and reuse of these software parts even more difficult. To illustrate the link of the human-machine interface (HMI) to functional and extra-functional control software implemented in accordance to IEC 61131-3 on a programmable logic controller (PLC), the interaction of a human operator with the machine is explained from an application perspective (cf. Fig. 1). Due to an error situation, the operator pushes the emergency stop of the CPPS, e.g. a machine being part of a CPPS (cf. Fig. 1, step 1). In the PLC, the operating mode is changed accordingly, and the machine is stopped either immediately or in controlled steps (2). The HMI control panel displays the machine status and an error message to the operator (4), which the PLC previously transmitted to the HMI control panel (3). Depending on the operator's reaction, e.g. removing a jammed work piece

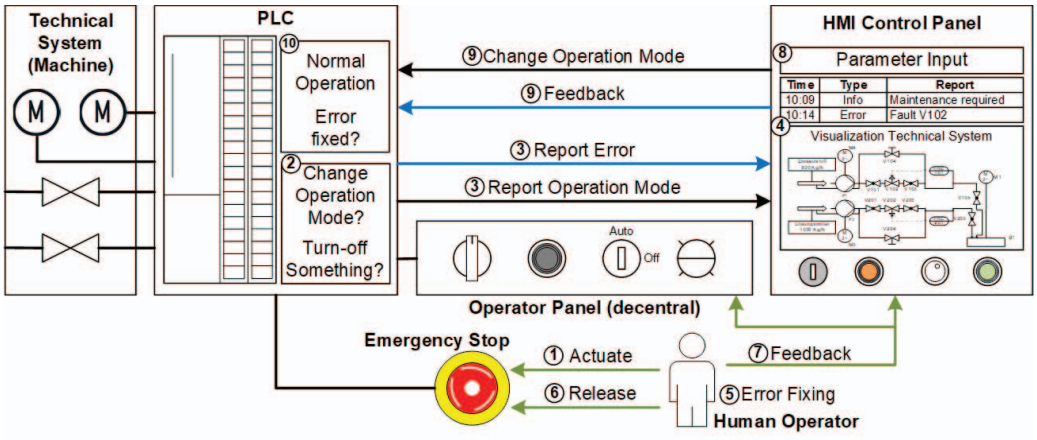

Fig. 1. Relationship between HMI and PLC in case of a fault (emergency stop actuation by operator, CPPS reaction until return to automatic mode)

manually (5), subsequently releasing the actuated emergency stop switch (6) and then acknowledging the error situation via HMI control panel (7, right fork, 8, 9) or a decentralized operator panel (7, left fork), the PLC checks its status (10). If necessary, the PLC allows an organized restart in manual mode and a subsequent manual change to automatic mode, if all interlocks are correctly parameterized. Alternatively, the operating mode can also be set back to automatic mode after the operator has acknowledged the error on the HMI control panel (8), which is checked by the control system after transmission (9, 10).

This simple example shows that troubleshooting, operating modes, and HMI are strongly interlinked, but they have fundamentally different implementation and code structures. This makes it difficult to manage variability and systematic reuse. In this paper, safety functions are included in the form of extra-functional code in PLCs providing diagnosis and error handling, operating mode switching and HMI interaction for those functions.

II. State of the Art- Managing Variants of Functional and Extra-functional Control Software

This section illustrates an approach for variability management of functional control software using the example of a simple cylinder. Further, it gives an overview on extra-functional aspects in CPPS control followed by details on the extra-functional control software. Finally, reuse by means of object-orientation is introduced.

A. Variability Management of Functional Control Software in CPPS

Most CPPS are controlled by real-time capable PLCs programmed with textual and graphical languages defined in IEC 61131-3. A PLC software project contains global variables and so-called Program Organization Units (POUs) encapsulating functionality as reusable software modules. Each POU consists of a variable declaration and an implementation part. Its functionality can be enclosed even more fine-grained within Actions that represent sub-functionalities of a POU. Three POU types, i.e. Programs, Function Blocks (FBs) and Functions are differentiated, with FBs containing a simple class concept and maintaining an internal state [5].

In the software engineering domain, Software Product Line Engineering (SPLE) is a well-established approach for planned reuse of variant-rich software systems, where artifacts common to all variants are ideally implemented only once [6]. For representing an SPL as 150%-model including all implementation artefacts, Family Models can be used. They depict the software artifacts hierarchically and grouped into variability categories, i.e. mandatory (present in all variants), alternative (can be used interchangeably) and optional (present in only some variants). Figure 2 shows an example of a family model, which documents the variability of a mono- and a bistable pneumatic cylinder typically used in assembly machines. While both cylinders have the action *DO_Extend* to extend (mandatory), the monostable cylinder retracts mechanically with a spring in contrast to the bistable cylinder, which requires an additional action *DO_Retract* to control its second valve (optional).

B. Overview on Extra-functional Aspects of CPPS Software

General quality requirements, also called non- or extra-functional requirements (NFR), should be considered during

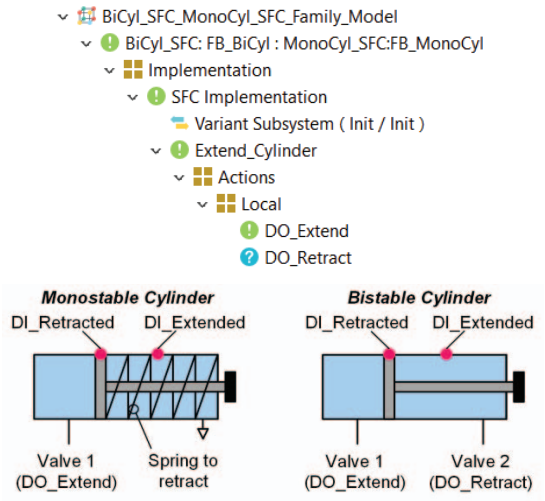

Fig. 2. Excerpt of Family Model depicting the variability of mono- and bistable cylinders (with green exclamation mark = mandatory, arrows = alternatives, blue question mark = optional)

software design. Thereby, the main NFR requirements from ISO/IEC 25010 are reliability, performance efficiency, compatibility, portability and maintainability [7]. During software evolution it has to be monitored that NFRs are met, e.g. after a change, a CPPS should have at least the same level of fault tolerance as before to fulfill reliability and performance. For assigning automation functions to distributed networked control systems, i.e. hardware (nodes and networks) and, thus, the time behavior and required storage, Fay et al. introduce a SysML-based approach [8] describing system characteristics and introducing selected design patterns, but neglecting safety functions [9].

Functional software for implementing the behavior in automatic mode is only a small part of the control logic compared to the extra-functional software, which is with 50-75% the much larger part [3]. Accordingly, Güttel et al. highlight that well-defined control software suitable for reuse, e.g. a POU for hardware control, includes standardized interfaces to HMI, various operating modes and diagnostic options [10]. Error detection, diagnosis and operating modes are implemented in about 50% of all software modules [11].

C. Details on Extra-functional Aspects of CPPS Software

The following subsections present approaches targeting different extra-functional aspects of CPPS software.

1) Variability and reuse of HMI code: With regard to HMI, Salihbegovic et al. [12] emphasize its importance as a component of control software and its reuse; however, the proposed implementation of an exemplary development process does not include variability. Besides, Urbas and Doherr [13] introduce a method for automatically generating HMIs from a process industry model, i.e., piping and instrumentation diagrams, allowing reuse through abstraction.

2) Operating modes in CPPS control code: In the food and beverage industry, the OMAC standard is used as a part of the PackML [14] for the condition-based realization of operating modes to connect machines from different manufacturers. Based on PackML, Barbieri et al. present a modular software concept, in which operating modes are

implemented as automata [15]. Thereby, the control code of a machine responds to a change of state, but HMI software is not targeted. Ladiges et al. [16] present the DIMA concept also using PackML for handling operating modes in modular process plants and include HMI functionality. Concepts for alarming and diagnosis are not yet included. Bani Younis and Frey et al. [17] focus on reengineering and follow the approach of automatically converting control software into UML class and state diagrams. For documentation, the formalization also enables investigation of the availability and performance of the software. However, no special focus is laid on extra-functional aspects and HMI software.

3) Diagnosis and error handling: For error handling, [18] and [19] work on a framework to investigate reliability in early development phases. Papkonstantinou et al. investigate feature models for functional aspects and integrate fail-safe (extra-functional) aspects to find valid combinations of configuration options [19]. Both integrate functional and extra-functional aspects at a high conceptual level for the process industry, but do not consider concrete control engineering aspects.

Detailed code analyses of industrial PLC software examine the significance and strategies for implementing extra-functional control software [20]. Recent analyses show so-called design patterns in control software and experts from industry confirm their application for specific, extra-functional aspects [4]. Vogel-Heuser analyzed common error handling strategies in industrial PLC code and additionally developed performance metrics to evaluate real-time error detection and error coverage for the CPPS domain [21]. Besides, Priego et al. developed a concept for reconfiguration in case of a PLC error and distinguished different methods of error reaction depending on the severity of the error and the system state [22]. Based on this approach, [23] used a model-based approach to define restart points after a control error.

In classical control code, approaches with a designated alarm or error variable are used, whereby the error variable is checked and updated in every POU controlling any type of actuator, i.e. drives or similar (cf. variable *bStop* in Fig. 3). Errors are looped through the entire code: If a module on the lowest level (e.g. FB_Drive) fails, the error is reported to the higher levels (PRG or OB1).

In summary, various approaches target extra-functional control software and its reuse. However, up to now, none focuses on a comprehensive variability analysis in both, extra- and functional PLC software, to support systematic reuse.

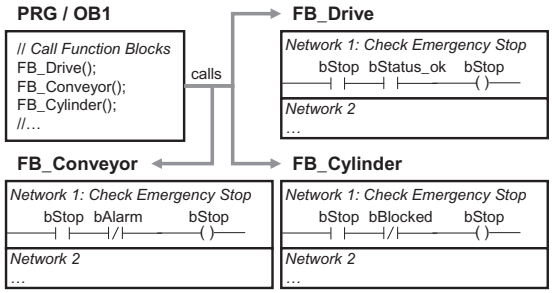

Fig. 3. Error handling (emergency stop) with global variable accessed in all POUs responsible for actuator control (FBs in ladder diagram)

D. Object-oriented IEC 61131-3 Concepts to Ease Reuse

For enhanced reuse of control software, the object-oriented extension of IEC 61131-3 (OO IEC) introduces three new language elements, *methods*, *inheritance* and *interface abstraction* [5]. Thereby, *methods* enable the separation of an FB's tasks, e.g. initialization or error handling, in sub-routines. Further, *inheritance* supports the reuse of common code parts to ease building new variants with the keyword EXTENDS. *Interfaces* serve to improve the cooperation of developers in an IEC 61131-3 project by defining templates, which can be adopted and filled by FBs with the keyword IMPLEMENTS. While an FB can only inherit from one other FB, it may implement various interfaces [5]. Some PLC development environments also support the use of *properties*, which specify a defined access to a value and possess a pair of methods for reading (Get method) and writing (Set method) the respective value. This enables verifying data consistency and value adjustments. Summarizing, OO IEC provides a variety of rewarding solutions to improve reuse and quality of control software in CPPS, and is, therefore, used as the basis for the proposed concepts in this paper.

III. Concepts to Ease Reuse and Increase Reusability of Extra-Functional Software in CPPS

This section presents an approach to enhance the reuse of extra-functional software based on the OO IEC (cf. Sec. II.D). First, an application example including variability and dependencies between different software parts is introduced. Next, OO IEC approaches to ease reuse of (extra-)functional control code are illustrated. Questionnaire results provide quantitative insights for these concepts and, finally, current applications of these strategies by two selected companies are qualitatively illustrated.

A. Cylinders as Standard Component in CPPS

Cylinders represent a simple type of actuator in CPPS and occur in a variety of different applications with small variations (cf. Fig. 2). In industrial CPPS, however, even the control of such a simple component can get very complex and extensive. The following comparison of two FBs for the control of an industrial cylinder provided by a machine manufacturing company and a cylinder in an academic lab-size demonstrator illustrates this: While the industrial cylinder FB provides a connection to the HMI, multiple error messages and different operating and testing modes, the academic implementation excludes these extra-functional aspects. The increased functionality of the industrial FB is reflected by the scope of its implementation, which is written in the language Ladder Diagram (cf. Fig. 3) and comprises 33 networks. The academic implementation, on the other hand, comprises an FB with two actions in Sequential Function Chart with only three steps each. A comparison of defined variables illustrates the difference even clearer: While the academic FB uses only six variables, the industrial FB controls 150 variables, which are partly organized into User Defined Types and Structures.

In both, the academic and the industrial software, different types of cylinders occur, e.g. mono- and bistable cylinders (cf. Fig. 2), leading to variability in functional software and in the related HMI software, which directly accesses variables from the functional software part to illustrate the CPPS's status to the operator or maintenance staff. For variability management we propose to combine the family model of functional control software with the HMI family model via extra-functional

control software, e.g. operating modes or error handling and diagnosis, being orthogonal to both (cf. Fig. 4).

A PLC family model, containing mandatory and optional parts to represent the mono- and the bistable cylinder, depicts variables, actions and OMAC actions (modes of operation). Thereby, OMAC actions use the cylinder actions for control, which in turn set the actuators (digital outputs *DO_Extend* and *DO_Retract*) and read the sensor signals. The HMI visualizes the cylinder (either with one or two valves) and its position (requiring recent sensor values) and displays general information about the cylinder status for the operator. Thus, it receives data from the PLC (variables *Status, DI_Extended. DI_Retracted*). In case of an error, the HMI control panel displays an error message, which is also read from the *Status* variable. In contrast, if the operating mode is set to manual (either by PLC in case of an error or via HMI), the actuator variables are set from the HMI (mandatory *DO_Extend* and optionally *DO_Retract*, depending on the cylinder type). This example illustrates the close interdependencies between PLC and HMI for extra-functional aspects.

B. *Methods, Inheritance, Properties and Interfaces to Ease Reuse of Functional and Extra-functional Control Code*

As highlighted by [5], the usage of OO IEC should tremendously ease modularity, reuse and the management of variants and versions. The OO IEC concepts provide the means to design a well-defined software architecture, which supports reuse of common parts, prevents code clones and enables standardized interfaces. At the example of the two cylinder variants introduced in the previous Section III.B, we illustrate the use of OO IEC and its benefits (cf. Fig. 5).

Reuse of control software is enabled by encapsulating functionalities, e.g. functional software such as an actuator's control or extra-functional software like error handling within designated FBs. The OO IEC concept *Method* takes this a step further by enabling the encapsulation of sub-functionalities inside an FB, e.g. extending a cylinder. This introduces more flexibility in the code, makes it clearer than monolithic software and enhances reuse, since a sub-functionality does not have to be implemented for every hardware module individually, e.g. *Method METH_Extract* can be used for both, mono- and bistable cylinders, by inheriting from a common base class. Further, by using state machines for implementation, the states of all modules are known at all times, which eases error handling, e.g. by handling all queries of all modules prior to starting an actuator or by resetting all

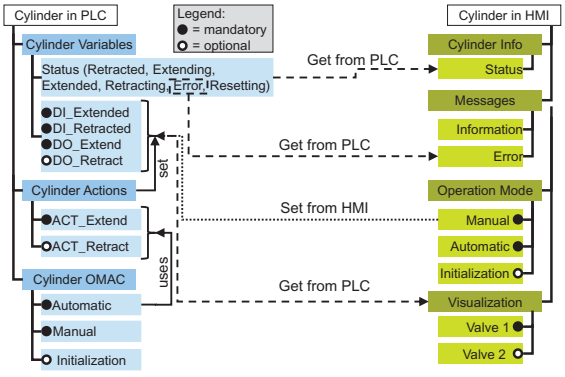

Fig. 4. Combination of functional PLC code and HMI code via extra-functional aspects, i.e operating modes, diagnosis and error handling

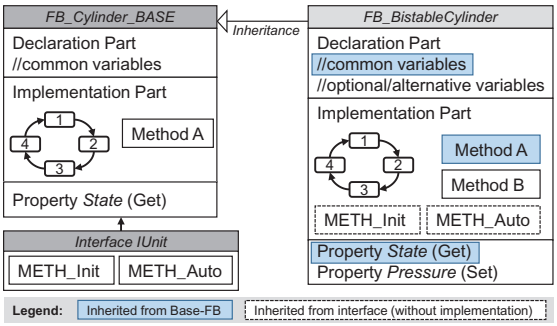

Fig. 5. Using OO IEC for eased reuse of (extra-)functional software

modules via their reset-method, by iterating over an array of interface references of all units of the machine.

Concerning variability within the software, the cylinder family model (cf. Fig.2) illustrates common and variable parts. Encapsulation itself is not enough to avoid code clones, which decrease the maintainability of the respective software. The OO IEC concept *Inheritance* successfully addresses this challenge: Parts used by several modules (e.g. *METH_Extract* used by both cylinder variants) are defined once in the base class and then inherited by the specific module variants. This is depicted in Fig. 5, where Method A is defined in the base and inherited by FB_BistableCylinder, which defines an additional Method B for retracting. This kind of reuse reduces the programming effort and facilitates the maintainability of the modules, e.g. in case of errors, when a correction of the inherited parts is necessary: it does not have to be performed for each module variant, but only once in the base class.

Regarding modularity, the principle of information hiding is essential, meaning that all module data is encapsulated internally and only specified information is shared externally with other software entities. For this purpose, it is advantageous if values can be transferred into an FB from outside (e.g. a command to change the operating mode) and, in addition, values can be queried from outside (e.g. the current state of a cylinder module, i.e. retracted, extending, extended, retracting or error). The OO IEC concept *Properties* enables the access of defined variables, which oftentimes are already available in the module, without having to access global variables. If values that are available in each module should be obtained, e.g. to query the status of the individual modules, these values can be accessed externally using properties instead of obtaining the values of local variables separately for each module. In the example considered here, this is the current state of the cylinders. To avoid having to create properties for each module separately, a base module can be used. In this base module, necessary properties are created that all modules of a certain type should have (cf. Fig. 5, FB_Base, Property State). These properties can then be used by the FBs of the different modules (e.g. of different cylinders) by inheriting from the base module (cf. FB_BistableCylinder, which inherited the property State).

In some cases, inheritance is not sufficient as an FB can only inherit from one Base-FB, but not from multiple bases. More precisely, if several FBs implement a similar functionality, but the definition of a single, suitable Base_FB is not possible (e.g. due to variants of required operating modes), this functionality would have to be created separately for each FB – even though parts of the control code are

redundant. The OO IEC extension *Interfaces* avoids this challenge and is especially beneficial regarding the reuse of extra-functional tasks, which are implemented by each module (e.g., fault handling or particularly critical tasks such as an emergency stop). An interface declares templates for methods and properties that an FB should have without specifying their implementation. Thereby, the central idea is to create standardized module interfaces without knowing what should be controlled afterward; there is no direct connection between implementation and functionality. An interface has only declaration parts, in which the methods and properties are defined. The implementation parts are created for each FB individually. If an FB implements an interface, the FB must adopt all methods and properties contained therein (cf. Fig. 5, FB_BistableCylinder inherits the Methods from Interface IUnit). If an FB is accessed via an interface, this underlying FB can be exchanged with another one implementing the same interface, even at runtime. By using the interface in several FBs, the programming effort can be reduced significantly, especially for large systems. In addition, interfaces can be used to generate a list of all included modules, which can then be processed for general module functionalities using an array processed in a for-loop for all modules. The functionalities have the same meaning but do not necessarily have the same implementation. If new modules are added, they can take over the interface and need not be programmed from scratch. Industrial CPPS often comprise several hundred modules, which usually need to provide multiple operating modes (e.g., automatic, manual or jog mode) that need to be switched on demand as part of extra-functional control software. To minimize the programming effort, the switching of these operating modes can be defined as methods in an interface and then be implemented for all FBs of the modules. By means of a generated module list (array), the methods of all modules can be called in a for-loop, enabling an efficient changing of the operating mode of all modules within a CPPS.

The use of IEC OO makes control software development initially more complex. However, the following, qualitatively described industrial example illustrates its advantages regarding software maintainability, extensibility and reuse: In industry, CPPS manufacturing companies often produce several thousand modules per year, which are then combined to form many production lines. Accordingly, the number of reused modules is very high, which makes the additional effort in the engineering process profitable. A developer needing a few hours longer to implement a cylinder using OO IEC than for conventional programming will benefit in the long term since the cylinder is implemented once and can often be reused several thousand times. Thereby, the software quality increases since more frequent use leads to improved testing and bug fixing is improved as errors only need to be fixed once in the common code parts.

C. Successful Industrial Strategies to Connect Functional Code, HMI and Diagnosis Using OO IEC 61131-3

This subsection introduces insights gained from questionnaires and strategies derived within expert interviews.

1) Questionnaire-based Results

Works on code analysis identified interfaces for data exchange as enablers for easier reuse. More mature modules also include operating modes, diagnosis and fault handling. However, a survey showed low usage of OO IEC: only 10% of the participating companies use OO IEC by default, 48% apply it partially and 42% do not use it at all [11]. Our recent questionnaire study with 61 companies confirmed this low usage rate of OO IEC with around 40% for machine and plant manufacturers. Only 24 companies indicate to use interface / properties and OO IEC. We analyzed whether they include fault handling, diagnosis, operating modes, HMI and their combinations as standard. Ten of ten companies include fault handling and diagnosis as standard, nine operating modes, seven HMI and six include all three. Further, a deeper analysis of category combinations like PLC type, interfaces, functionalities included within modules, checklists, version tracking and reuse strategy with its sub-categories universal module and libraries showed that participants answered unexpectedly: usage of interface/properties (without the limitation to IEC OO) reduces the usage of data exchange via global variables significantly. However, use of interface / properties would require usage of OO IEC as a prerequisite (cf. Sec. II.D). Companies answering the questionnaire neglected this prerequisite and must hence refer to other mechanisms. In summary, we can assume that the mechanism and concepts introduced in Sec. II.B are not widely used yet.

2) Interview-based Results

Within a qualitative study to examine control software architecture in machine and plant manufacturing companies from the domain of packaging machinery, it was analyzed through interviews with PLC software experts how different companies implement (extra-)functional tasks in their software. Regarding the implementation of fault handling, different strategies could be observed (cf. Fig. 6): One interviewed company (Company A) uses the error handling template provided by the platform supplier, which provides a superordinate global error list for all potential errors of the machine. In case of an error, the affected module writes a structured error information (including, e.g., the error ID and its severity) in the list by calling a central exception handling function. For each cycle, the error list is checked on the highest level of the software hierarchy. In case there are entries, an algorithm combines the severity value of all entries and derives an appropriate reaction for each module based on a matrix specifying the error reaction for different events.

The control software of the packaging machine manufacturer Company B comprises two parts, i.e. a standard part for controlling the machine and an additional part for process handling to ensure sterility. Company B uses the same supplier template as Company A, but enlarges the error handling using OO IEC interfaces and properties to enhance modularity, reuse, and thus the overall software quality. Thereby, FBs in the application software inherit from basic FBs from supplier libraries containing predefined errors and reactions, and implement respective interfaces of an internal exception-handling library (cf. Fig. 6). The interfaces define several properties including a Boolean property to set exceptions. Once an erroneous situation has been identified, the error is triggered in an FB by assigning an error condition to a local property. By assigning unique numbers to the error messages, which are used consistently in the PLC and the HMI, the machine operator can quickly identify the machine module causing an error. To specify the error reaction, Company B also uses a matrix. Analogously, Company B uses interfaces and properties to implement other extra-functional tasks, e.g., the integration of the standard control structure for axes movement into the process handling structure of the software. Here, OO-IEC strongly facilitates the simulation of the machine by imitating its behavior using interfaces and,

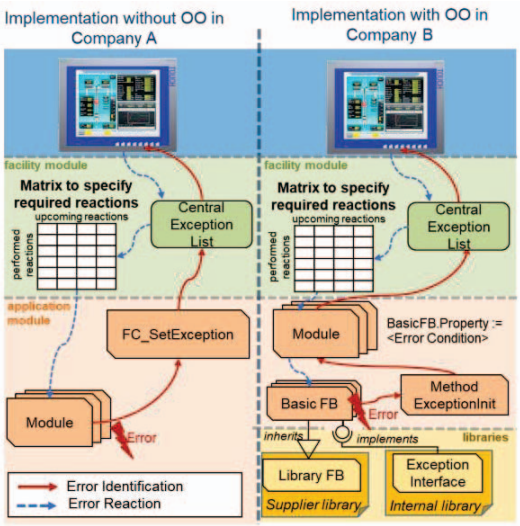

Fig. 6. Comparison of the HMI connection, error identification and reaction of two companies based on the same supplier template using OO IEC mechanisms

thus, simulating error triggering, and the software complexity is significantly reduced. Depending on the task to be implemented, different interfaces are defined in the internal libraries to access different types of FBs, e.g., interfaces for connecting actuators, such as valves or drives, or for connecting safety devices. PLC experts of Company B confirmed that the usage of OO IEC represents an essential design element to enhance the software architecture by optimizing reusability and exchangeability of extra-functional software parts while keeping software complexity low.

IV. Conclusion and Future Work

The challenge well known by companies from the machine and plant manufacturing industry to maintain variability of functional control code in combination with HMI code and extra-functional control code like diagnosis, fault handling and operating mode including managing their interdependencies was introduced. Up to now, research focused either functional or selected aspects of extra-functional code. Considering the variant management and planned reuse of both, we introduced the challenge to connect family models of functional code and HMI code related to the orthogonal extra-functional code. Additionally, we discussed how the usage of OO IEC 61131-3 concepts could ease the reuse of these orthogonal parts, which link the variability models of functional and extra-functional code and, thus, enabling flexibility in the context of Industry 4.0. Concluding, reuse strategies for extra-functional aspects from industry derived with questionnaires and interviews were shown.

References

[1] K. Thramboulidis, "The 3+1 SysML View-Model in Model Integrated Mechatronics," *JSEA*, vol. 03, no. 02, pp. 109–118, 2010.

[2] J. Fischer, S. Bougouffa, A. Schlie, I. Schaefer and B. Vogel-Heuser, "A Qualitative Study of Variability Management of Control Software for Industrial Automation Systems," in Int. Conf. on Software Maintenance and Evolution, Madrid, Sep. 2018, pp. 615–624.

[3] M.R Lucas and D.M Tilbury. "A study of current logic design practices in the automotive manufacturing industry," Int. Jn. of Human-Comp. Studies, vol. 59, no. 5, pp. 725-753, Nov. 2003.

[4] E. Neumann et al., "Formalization of design patterns and their automatic identification in PLC software for architecture assessment," in 21st IFAC World Congress, Berlin, July 2020, in press.

[5] B. Werner, "Object-oriented Extensions for IEC 61131-3," IEEE Industrial Electronics Magazine, vol. 3, no. 4, pp. 36-39, 2009.

[6] K. Pohl, G. Böckle and F. van der Linden, Software Product Line Engineering. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005.

[7] T. Frank et al., "Dealing with non-functional requirements in distributed control systems engineering," in Int. Conf. on Emerging Technologies and Factory Automation, Sept. 2011 Toulouse, p. 1-4.

[8] A. Fay, B. Vogel-Heuser, T. Frank, K. Eckert, T. Hadlich and C. Diedrich, "Enhancing a model-based engineering approach for distributed manufacturing automation systems with characteristics and design patterns," Journal of Systems and Software (JSS), vol. 101, pp. 221-235, 2015.

[9] T. Hadlich, S. Höme, C. Diedrich, K. Eckert, T. Frank, A. Fay and B. Vogel-Heuser, "Time as non-functional requirement in distributed control systems," in IEEE Int. Conf. on Emerging Technologies and Factory Automation, Sep. 2012, pp. 1-7.

[10] K. Güttel, P. Weber and A. Fay, "Automatic Generation of PLC Code Beyond the Nominal Sequence," in IEEE Int. Conf. on Emerging Technologies and Factory Automation, Hamburg, Sep. 2008, pp. 1277-1284.

[11] B. Vogel-Heuser and F. Ocker, "Maintainability and evolvability of control software in machine and plant manufacturing — An industrial survey," Control Eng. Pract., vol. 80, no. Nov., pp. 157–173, 2018.

[12] A. Salihbegovic, Z. Cico, V. Marinkovi and E. Karavdi, "Software Engineering Approach in the Design and Development of the Industrial Automation Systems," in Int. Workshop on Software Engineering in East and South Europe, New York, May 2008, pp.15-22.

[13] L. Urbas and F. Doherr, "autoHMI: a model driven software engineering approach for HMIs in process industries," in Int. Conf. on Computer Science and Automation Engineering, Shanghai, June 2011, pp. 1-5.

[14] Organization for Machine Automation and Control, "PackML Unit / Machine Implementation Guide Part 1: PackML Interface State Manager", 2016. [Online] http://omac.org/wp-content/uploads/2016/11/PackML_Unit_Machine_Implementation_Guide-V1-00.pdf, accessed 06.10.2020.

[15] G. Barbieri, N. Battilani and C. Fantuzzi, "A PackML-based Design Pattern for Modular PLC Code," IFAC-PapersOnLine, vol. 48, pp. 178-183, 2015.

[16] J. Ladiges et al., "Integration of Modular Process Units Into Process Control Systems," IEEE Trans. on Industry Applications, vol. 54, no. 2, pp. 1870-1880, 2018.

[17] M. B. Younis and G. Frey, "UML-based Approach for the Re-Engineering of PLC Programs," Annual Conf. on IEEE Industrial Electronics, Paris, 2006, pp. 3691-3696.

[18] S. Sierla, B. O'Halloran, T. Karhela, N. Papakonstantinou and I. Tumer. "Common cause failure analysis of cyber–physical systems situated in constructed environments," Research in Engineering Design, vol. 24, no. 4, pp. 375–394, Oct. 2013.

[19] N. Papkonstantinou, S. Proper, B O'Halloran and I. Tumer. "Simulation Based Machine Learning For Fault Detection in Complex Systems Using the Functional Failure Identification and Propagation Framework," in Computers and Information in Engineering Conference, Buffalo, Aug. 2014, pp. 1-10.

[20] B. Vogel-Heuser, J. Fischer, S. Rösch, S. Feldmann and S. Ulewicz, "Challenges for Maintenance of PLC-Software and Its Related Hardware for Automated Production Systems: Selected Industrial Case Studies," in Int. Conf. on Software Maintenance and Evolution, Sep. 2015, pp. 362-371.

[21] B. Vogel-Heuser, S. Rösch, J. Fischer, T. Simon, S. Ulewicz and J. Folmer, "Fault handling in PLC-based Industry 4.0 automated production systems as a basis for restart and self-configuration and its evaluation," Journal of Software Engineering and Applications, vol. 9, no. 1, pp. 1 – 43, 2016.

[22] R. Priego, D. Schütz, B. Vogel-Heuser and M. Marcos, "Reconfiguration Architecture for Updates of Automation Systems During Operation," in IEEE Int. Conf. on Emerging Technologies and Factory Automation, Luxembourg, Sep. 2015, pp. 1-8.

[23] P. Bareiß, D. Schütz, R. Priego, M. Marcos and B. Vogel-Heuser, "A Model-based Failure Recovery Approach for automated Production Systems combining SysML and Industrial Standards," in Int. Conf. on Emerging Technologies and Factory Automation, Sep. 2016, pp. 1-7.