21th IEEE Int. Conf. on Emerging Technologies and Factory Automation, pages 1–8, 2016

Automatic Generation of Diagnostic Handling Code for Decentralized PLC-based Control Architectures

Michael Steinegger, Martin Melik-Merkumians, Johannes Zajc, and Georg Schitter

Automation and Control Institute, Vienna University of Technology
Gußhausstraße 27-29, A-1040 Vienna, Austria
Email: {steinegger,melik-merkumians,zajc,schitter}@acin.tuwien.ac.at

***Abstract*—In this paper, an ontology-based approach to automatically generate control applications to handle diagnostic information of decentralized control devices is presented. Diagnostic possibilities of modern remote I/O devices are analyzed and software components in terms of function blocks to handle the specific diagnostic information are defined. After a detailed conceptual overview, the application of the proposed knowledge-based code generation approach to a PLC-based control architecture of a hot rolling mill is described. It is shown that the proposed approach significantly reduces engineering time and the error rate in the design processes of industrial control and diagnostic applications, since the application engineering is raised to an abstract level by utilizing pre-defined, tested, and reusable function blocks and a user-definable set of code generation rules to encode repetitive implementation tasks. The rules are defined in the query language SPARQL with additional ARQ functions to reduce the complexity.**

I. Introduction

The overall requirements on safety and security steadily increase together with the complexity of modern manufacturing systems. To protect humans (operating personnel) as well as the plant and the environment from possible harm, system faults have to be detected, localized, and handled at an early stage. In modern topologies of decentralized *process control systems* (PCSs), where the hardware is distributed while the control software still remains (almost) monolithic, each remote I/O device provides several diagnostic possibilities. This information has to be handled in the superior PCS and is normally implemented manually, which represents a time-consuming and error-prone task and takes up a significant part of the complete control application. For example, fault handling in robotic cells takes up to 80% of the complete control code, as observed by Smith and Gini [1]. Similar numbers can be found for industrial control applications, as for flexible manufacturing cells Richardsson and Fabian [2] reported on just 10% of the complete control application for automatic control.

One of the possibilities to reduce engineering time and improve the overall quality of the control application is to apply design patterns. For example, Serna et al. [3] proposed two different design patterns for control applications according to the international standard IEC 61499 [4]. While each *function block* (FB) in the proposed design patterns has its own failure detection methods, Steinegger et al. [5] elaborated two design patterns which strictly separate fault detection and fault handling methods from control code in control applications for discrete manufacturing systems. However, despite the fact that reusable design patterns reduce the effort and increase the overall code quality (cf. Gamma et al. [6]), the manual effort for implementing such fault handling methods is still high and contains a lot of repetitive, tedious, and thus error-prone work.

In order to further reduce the manual effort and manually introduced design errors on implementing industrial control applications, automatic code generation for industrial controllers (e.g., *programmable logic controllers* (PLCs)) became a major research topic. Available concepts for automatic code generation range from transformation of Petri net models (cf. Frey [7]) or deterministic finite automata (see for example Flordal et al. [8]) into executable PLC control applications to approaches that are based on structural information from preceding engineering project phases (cf. Drath et al. [9]). However, the major drawback of automata-based approaches is the additional engineering step that has to be added to the engineering workflow, since it still requires expert knowledge to specify the desired control algorithms in Petri nets or automata. Furthermore, existing knowledge-based code generation approaches are mostly focusing on specific information, like structural knowledge, for example from piping and instrumentation diagrams. Since those approaches neglect the entire integrated plant model, the code generation process is restricted to specific subparts of the complete control applications. By semantically integrating different (data or knowledge) sources to an entire plant model, the generation process could cover a huge part of the overall control application. Here, ontologies are considered as a well-suited model class to achieve the above-mentioned semantic integration (cf. Morbach et al. [10]).

In this paper, a knowledge-based code generation approach is presented, which enables the generation of complete diagnostic handling methods for decentralized PLC-based control architectures. The approach utilizes different ontologies based on AutomationML [11], which enable the semantic integration of different domain knowledge. Due to the ontological base, the transformation process to PLC code is based on SPARQL [12] queries which are executed on the ontologies by an inference engine. The benefit of this approach is that the code generation process can be easily adjusted by the user, by changing or defining new SPARQL rules without affecting the implementation source of the code generator. Furthermore, in contrast to the component-based code generation framework for discrete manufacturing systems as proposed by Güttel [13], the approach presented enables possibilities to automatically change existing PLC applications according to a given hardware configuration.

The remainder of the paper is organized as follows. In Sec. II, the diagnostic possibilities of modern fieldbus devices for Profibus DP (Decentralised Peripherals) and EtherCAT are analyzed, representing the basis for the definition of software components. A conceptual overview of the proposed code generation approach for diagnostic information handling methods is given in Sec. III. Based on this overview, the implementation of the code generator is stated in Sec. IV and its evaluation is described in Sec. V. Section VI concludes the paper.

II. Fieldbus Device Diagnostic Possibilities

Modern remote I/O devices provide, beside the bidirectional transmission of sensor values or control commands, also additional diagnostic possibilities. Typically integrated diagnostic functions encompass the diagnosis of the communication interfaces as well as the status of each module installed on the I/O device or monitoring of wire breakage for each channel (see Fig. 1 for a Profibus DP example). Such additional diagnostic information, if fast and properly handled within the superior control system, can drastically reduce plant downtime, and enables an early detection of possible component failures as well as alerting of operators in case of plant malfunctions.

The international standard series IEC 61158 [14] specifies several different types of industrial fieldbuses. Since this paper focuses on the generation of diagnostic handling information of Profibus DP and EtherCAT devices, these two fieldbus specifications are analyzed regarding their diagnostic possibilities specified in the corresponding international standards.

A. Profibus DP Slave Device Diagnostics

The fieldbus Profibus DP is one of the most prevalent fieldbuses applied for industrial control architectures. Its communication profile is compliant to the *communication profile family* (CPF) 3/1, which is specified in the international standard IEC 61784 [15]. The services of the application layer are defined in the international standard IEC 61158 [14, Part 5-3]. A single Profibus DP configuration supports a maximum of 126 device nodes per communication network and 244 Byte of sent or received data packets for each device. Here, a controlling device takes the communication master part (DP master) and communicates with other DP master devices or field devices, representing DP slaves in the communication topology.

Decentralized Profibus DP components provide different diagnostic and alarm information (depending on the Profibus DP protocol type) as depicted in Fig. 1 and specified in the international standard IEC 61158 (Part 5-3). Therein, the different diagnosis objects are encapsulated in the diagnosis *application service element* (ASE). The ASE incorporates device-related diagnosis of each DP slave, group or identifier-related diagnosis, which usually corresponds to a module of a Profibus DP slave, and detailed diagnosis information for each module as part of the DP slave configuration, as well as channel-related diagnosis. Furthermore, an alarm ASE is specified in the above-mentioned standard, which incorporates for instance diagnostic alarm (e.g., short circuit), process alarm (for example, when the upper limit for the temperature in a steam boiler is reached) or additionally configurable insert / remove alarms if one or more modules (e.g. peripheral modules for additional I/Os) are inserted or withdrawn from the DP slave configuration.

B. Diagnostics of EtherCAT Devices

The EtherCAT communication profile is compliant to CPF 12 and the network is restricted to 65535 communication nodes but without topological restrictions. In contrast to Profibus DP, the standard series IEC 61158 does not specify the diagnostic information that is exchanged between different EtherCAT devices in detail. However, most of the slave implementations provide fieldbus diagnosis and functional diagnosis information. Here, the fieldbus diagnosis includes information about the process data exchange and the operating mode settings. Device specific diagnosis (or functional diagnosis) encompasses,

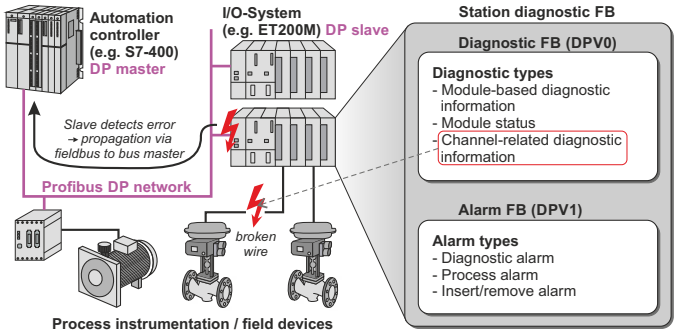


Fig. 1. Example of a decentralized process control system based on the fieldbus Profibus DP with centralized automation controller (DP master) and two remote I/O stations (DP slaves). In this example, a broken wire in the connection between a DP slave and a connected control valve is detected. The detected error is then propagated from the remote I/O station to the DP master, where the received error information has to be properly handled.

for example, indication of clamp over-temperature or drop of internal control power which exceeds the normal tolerances.

The listed standardized diagnosis information, which are commonly implemented for all slaves, or more specific, remote I/O devices (Profibus DP as well as EtherCAT devices), enables the possibility to implement diagnostic handling FBs for each fieldbus and device type. Such FBs can be defined and tested once and then be reused for implementation in consecutive projects, where diagnostic information of fieldbus devices have to be processed. The concept how these FBs can be automatically instantiated, parametrized, and interconnected, or even be deleted in case of changes in the communication topology (hardware configuration) is described in the following section.

III. Conceptual Overview

For a wide acceptance in the manufacturing industry, automatic code generation approaches have to integrate seamlessly into existing workflows of typical industrial engineering processes [16]. Therefore, such approaches have to cope with the information regarding plant automation and control, which is mostly contained in given design documents during the engineering phase of a manufacturing plant. Thus, a seamlessly integrated code generation approach has to cover the following two steps. First, the given design data has to be pre-processed in terms of integrating all information to a data-based plant model and providing the integrated information in a standardized and formal way (semantic integration). The second step then consists of processing the formalized model, extracting the information which corresponds to the control of the plant, and then generate the control application out of it (code generation). In the next section, industrial engineering processes and typical knowledge sources (design documents) to achieve the mentioned two steps are summarized.

A. Summary of a Typical Engineering Process

According to the published NAMUR worksheet NA 35 [17], a typical engineering process in the process industry is subdivided into different planning phases (as depicted at the bottom of Fig. 2). However, the specified project phases can also be mapped to various engineering processes in the discrete manufacturing domain with slight adaptations. After determination of the overall project targets, estimation of the approximate costs, and set up of the plant concept, the basic engineering

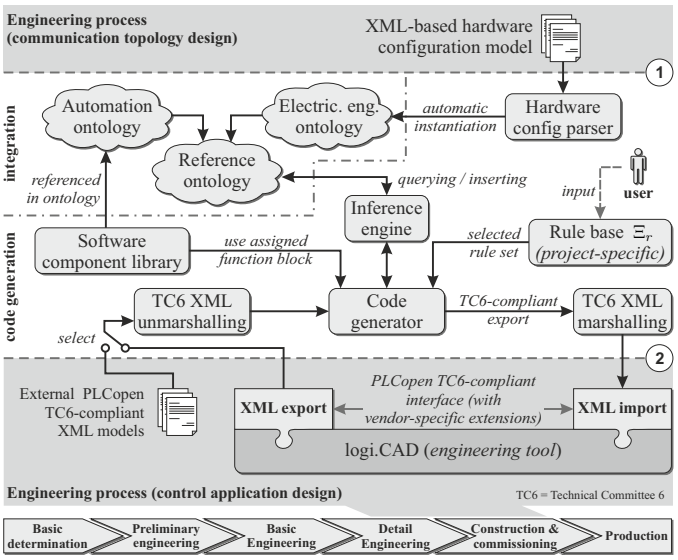


Fig. 2. Overview of the knowledge-based diagnostic code generation approach with interfaces (1) and (2) to the underlying engineering process (gray shaded). The information given in a hardware configuration model is mapped to the associated domain ontologies and is accessible for the code generation process via the reference ontology. The behavior of the code generator can be adjusted by an appropriate set of user-definable code generation rules.

is one of the key-phases for a successful engineering project. Here, required (and detailed) process data are collected and the technical realization of the plant is determined. The planned realization concept is then refined in the detail engineering phase in terms of selecting appropriate equipment (e.g., control devices), creating loop functions and electrical circuit diagrams, as well as specifying the hardware configuration of the fieldbus communication topology. Afterward, the equipment is installed and the control software is implemented and configured. For implementing the control and diagnostic handling applications, the responsible software engineer receives the engineering documents and implements the different functionality according to the therein specified requirements. Thus, a seamlessly integrated approach for automatic generation of the control or diagnostic handling software should be based on the knowledge, that is gathered during the various engineering phases and contained in several different engineering documents, and take this as the input for the code generation process.

One of the interfaces to the underlying engineering process of the proposed approach, marked as (1) in Fig. 2, is represented by hardware configuration models, which can be described, for instance, based on the *extensible markup language* (XML). These models are usually defined during the detail engineering phase and contain topological information of the control architecture and specify the number of controllers, decentralized components, and the communication structure. To apply the hardware configuration models for the automatic code generation approach, the information has to be mapped initially to a formalized model (cf. Frey and Litz [18]).

B. Ontology-based Formalization of Engineering Models

To formalize and semantically integrate different engineering models which are elaborated during the planning phases of a typical engineering process, an integrated plant model that is based on domain-specific standards is elaborated and implemented in a modular and scalable ontology-based framework.

1) Formal Ontology Definition: According to Kalfoglou and Schorlemmer [19], a general ontology $\mathcal{O}$ can be formally defined as $\mathcal{O} = (\mathcal{S}, \mathcal{A})$, where $\mathcal{S}$ represents the ontology signature and $\mathcal{A}$ is the set of axioms. A signature is a vocabulary set that describes the domain of interest. Therefore, the ontology signature can be defined in a formal way as the union set

$$\mathcal{S} = \mathcal{C} \cup \mathcal{P} \cup \mathcal{R} \ , \tag{1}$$

where $\mathcal{C}$ is the set of concepts and $\mathcal{P}$ the set of properties. Applying the definition of Däubler [20], the relation set $\mathcal{R}$ contains k-ary relations $\gamma \subset \mathcal{C}^k$, defined over a given set of concepts, such that $\mathcal{R}$ satisfies

$$\mathcal{R} = \left\{ \gamma \mid \gamma \in \mathcal{R}^* = \bigcup_{k \in \mathbb{N}} P(\mathcal{C}^k) \right\} \ , \tag{2}$$

where $P(\cdot)$ represents the partition function. The union set $\mathcal{K} = \mathcal{O} \cup \mathcal{I}$ of the signature, specified axioms, and individuals $\mathcal{I}$ (instances of ontology concepts) can then be defined as the knowledge base $\mathcal{K}$ or instance model of the domain of interest.

2) Integration Framework: In Fig. 3, the overall structure of the ontology-based integration framework applied within the proposed knowledge-based code generation approach is illustrated. It is structured in a modular way, such that all the concepts which are specific for each engineering discipline involved in typical industrial engineering processes are represented by a single ontology. For instance, the electrical engineering ontology is composed out of several sub-ontologies which contain concepts to model communication topologies (like decentralized topologies based on remote I/O devices), their associated devices, as well as signals and signal types exchanged between different devices. Thus, the electrical engineering ontology is subdivided into an ontology containing all hardware-specific concepts (component layer) and an ontology based on the concepts defined in the international standard IEC 61175 [21] for representing the communication-specific information. The modular structure of the integration framework has several advantages. Since the overall model is split into domain-specific ontologies and one reference ontology, which establishes the semantic interoperability between the domain ontologies, new ontological models can be easily integrated. Furthermore, new references or mappings between the signatures $\mathcal{S}_i$ and $\mathcal{S}_j$ of two different domain ontologies can be defined in the axiom set $\mathcal{A}_r$ of the reference ontology. Thus, the different domain ontologies can be defined and maintained completely independent from all the other involved ontologies.

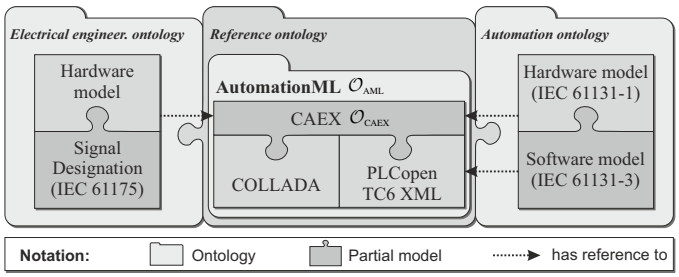


Fig. 3. Overview of the ontology-based integration framework representing a more detailed view of the integration part in Fig. 2. The reference ontology, acting as the bridge between several domain ontologies, is realized based on the exchange format AutomationML standardized in the IEC 62714.

The partial model of the ontology-based integration framework of Fig. 3 is depicted as an UML (*unified modeling language*) class diagram in Fig. 4. It specifies the concepts for representing the hardware devices of a fieldbus topology as well as concepts that represent the software-specific classes, defined in the standard IEC 61131-3. The IEC 61131-3 concepts in Fig. 4 represent just a small subset of the entire model since an extensive IEC 61131 meta model of the standard has already been published by Thramboulidis and Frey [22]. Since the hardware devices are usually arranged in a hierarchical structure of typical fieldbus topologies, the hardware models for the electrical components and for the fieldbus components are defined upon the *computer aided engineering exchange* (CAEX) ontology $\mathcal{O}_{\text{CAEX}}$ as part of AutomationML (standardized in the standard IEC 62714 [11]). Here, the concepts of the underlying sub-models of AutomationML have been chosen for implementing the reference ontology since those models provide a generic base to cope with topological information (CAEX) and data related to the programming of industrial automation systems (PLCopen TC6 (Technical Committee 6) XML). Furthermore, the AutomationML model represents the glue between the mentioned sub-models and specifies basic external data connectors for those sub-models [23]. With the mentioned data connectors, when properly implemented in the specific AutomationML ontology, it is possible to build a generic integration framework based on AutomationML.

Referring to Fig. 4, the concepts `EtherCAT_Device` and `DP_Device` can be defined as CAEX role families with all relevant fieldbus-specific attributes. An associated role class (e.g. `EK_Device` for EtherCAT couplers) can then be assigned as a supported role class to the concept `ControllingDevice`, depending on the fieldbus type. For this paper, due to presentation purposes, the above-mentioned domain ontologies import the AutomationML ontology signature $\mathcal{S}_{\text{AML}}$ with all its sub-models (cf. Fig. 3). This enables the subclassing of the desired concepts defined in $\mathcal{S}_{\text{CAEX}}$ (see class hierarchy on the left-hand side of Fig. 7). Mapping the XML-based hardware configuration of a fieldbus topology onto the ontology concepts defined in the domain ontologies and instantiating them accordingly, makes the topology information accessible via the reference ontology for a rule-based code generation process.

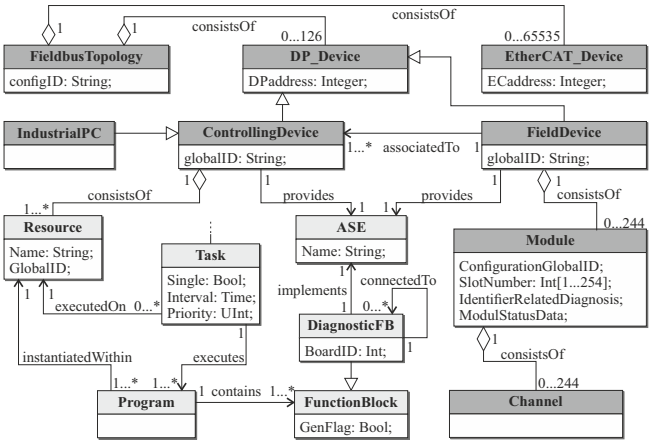


Fig. 4. UML model of a relevant part of the modular ontology-based integration framework as depicted in Fig. 3. It shows some hardware-specific concepts of the electrical engineering ontology (dark-gray shaded) as well as some software-specific concepts of the automation engineering ontology.

C. Rule-based Generation Process of Diagnostic Code

The second interface (mark ② in Fig. 2) to the underlying engineering process is established by PLCopen TC6 compliant XML representations of PLC control applications. The benefit of this XML representation, which has to be generated by the proposed knowledge-based code generation approach, is that today most of the PLC programming tools support this standardized exchange format as part of AutomationML at least on the basic level. Therefore, the underlying rule-based code generation process can be considered as a parametrized transformation $T(\Xi_r)$. Here, Ξ_r is defined as a rule base with a set of one or more user-definable transformation rules

$$r_i : \Phi_i(x_i) \rightarrow \Psi_i(y_i) \ , \tag{3}$$

where $\Phi_i(\cdot)$ is the condition function with the set of condition parameters x_i and $\Psi_i(\cdot)$ is the action function with the set of action parameters y_i. The behavior of the transformation process can then be formally defined as follows.

First, the antecedent $\text{Ant}(r_i) = \Phi_i(x_i)$ of a code generation rule r_i has to be specified by the user. This part of the rule is applied for querying the ontologies via the inference engine in Fig. 2. Thus, the antecedent represents a query pattern and can be defined with x_i, being a user-defined clause based on the overall ontology signature, which is formally defined as

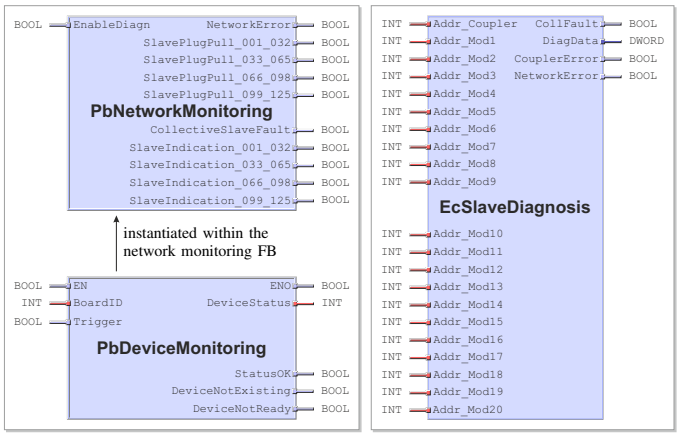
$$\mathcal{S}_{\text{REF}} \cup \left(\bigcup_{k=1}^{n} \mathcal{S}_k \right) \ , \tag{4}$$

where $\mathcal{S}_{\text{REF}}$ is the signature of the reference ontology and n is the number of domain ontologies $\mathcal{O}_k$. When $\text{Ant}(r_i)$ is sent as querying request to the ontology-based integration framework by the inference engine and the query pattern matches a subset of the overall instance set $\mathcal{I}$, defined in the similar way as the overall signature (4), the right-hand side of (3) is fired.

The second and last step is to specify the code generation part represented by the consequent $\text{Con}(r_i) = \Psi_i(y_i)$ of the rule r_i. Here, the user can specify how the code generator should behave in case of a matching query pattern. Consider the example of automatically instantiating diagnostic FBs according to a hardware configuration of a specific fieldbus topology stored in the integration framework. As depicted in Fig. 1, one station diagnostic FB can be assigned to each remote I/O system and the FB has to be parametrized, for instance, with the respective fieldbus address of the hardware device. From the hardware configuration, which has been transformed to the ontology representation as depicted in Fig. 4, this address can be easily retrieved by defining a query (for instance a SPARQL [12] query) which represents the left-hand side of rule (3). Since, according to Fig. 4, it is also specified in the ontology which `DiagnosticFB` implements the ASE of a specific `FieldDevice`, the required information set (address, diagnostic FB type) can be requested by extending the beforementioned query accordingly. This retrieved information set can then be forwarded to and processed by the right-hand side of (3). The code generation part $\text{Con}(r_i)$ of the rule then has to insert an instance of the corresponding FB to the PLCopen TC6 compliant XML and parametrize it with the address.

IV. Implementation

In this section, the implementation of the previously discussed code generation approach and the technical realization of the interfaces to the underlying engineering process are described.



(a) FBs for Profibus DP diagnosis **(b)** FB for EtherCAT diagnosis

Fig. 5. Implemented function blocks (FBs) according to IEC 61131-3 to handle diagnostic information of Profibus DP (a) and EtherCAT devices (b). For Profibus DP networks, the device diagnosis FB *PbDeviceMonitoring* is instantiated within the *PbNetworkMonitoring* block for each DP slave device.

A. Component Library for Fieldbus Device Diagnostics

Based on the analysis of diagnostic possibilities of industrial fieldbus devices, as described in Sec. II, a software component library in terms of a set of FBs compliant to the international standard IEC 61131-3 has been developed. The implemented FBs are illustrated in Fig. 5. For Profibus DP networks two different FBs have been implemented. The first FB, which is denoted as *PbNetworkMonitoring* on the left side of Fig. 5, monitors an entire Profibus DP network with a maximum of 125 remote I/O devices. It generates one *NetworkError* when communication errors between one or more fieldbus devices occur and provides outputs for indicating that components like peripheral modules have been inserted or removed from a slave configuration. For instance, the output *SlavePlugPull_001_032* indicates a pending insert / remove alarm from at least one of the first 32 Profibus DP slaves in the network. Furthermore, the outputs tagged with *SlaveIndication* indicate the presence of additional diagnostic information of one or more DP slaves. The second FB, which is denoted as *PbDeviceMonitoring*, is automatically instantiated within the first FB for each projected remote I/O device (DP slave). It provides the detailed diagnostic information for the corresponding hardware device and is configured with the device address (or *BoardID* in Fig. 5).

For EtherCAT networks a single FB has been implemented, which indicates through the *CollFault* output that one of the underlying devices of the corresponding sub-network has a present error, provides further diagnostic information denoted as *DiagData*, and generates EtherCAT coupler and network error indications. All described FBs have been implemented according to company-specific requirements and are externally stored as PLCOpen TC6 compliant XML files which are then referenced during the code generation process. Therefore, the internal logic of the FBs to handle the diagnostic information can be changed without affecting the generation process itself.

B. Engineering Process Interfaces

As already described in Sec. III-A, hardware configuration files represent the input for the proposed code generation approach. Some of the applied engineering tools support the export of the engineered configuration as XML representation, for instance, the EtherCAT configuration tool ET9000. However, the export functionality of several (typically company-specific) configuration tools is often restricted to flat-table representations (Excel sheets), where the hierarchical network structure is also flattened. Therefore, an algorithm has been implemented that reconstructs the hierarchical topology information from the flattened representation. Since Profibus DP communication networks are topologically constrained due to restrictions of the number of communication participants and the master-slave communication principle, the hierarchical information can be easily extracted out of the linear encoded address space. For topologically unconstrained EtherCAT networks, the structure of the address space is restricted to the convention, that each sub-topology is linearly encoded and each sub-device contains the address of the interfacing device (coupler) to the superior topology as leading numbers of the sub-device address. Applying the mentioned address convention also allows an automatic reconstruction of the topology out of flattened representations.

The interface to the PLC programming tool logi.CAD [24] is realized by its internal XML import and export functionality. As depicted in Fig. 6, the functionalities are programmatically interfaced in Java, in order to enable the automatic export of existing PLC software as well as the automatic import of generated or adapted PLC code. These interfaces are denoted as PLCopen TC6 XML unmarshalling (XML to Java objects) and marshalling (Java objects to XML) in Fig. 2 and Fig. 6. Therefore, the PLCopen *XML schema definition* (XSD) is processed by the JAXB (*Java Architecture for XML Binding*) compiler which generates the according annotated Java classes which can then be instantiated during the unmarshalling process.

C. Ontology-based Integration Framework

The ontologies of the integration framework depicted in Fig. 3 are implemented with Protégé [25]. To instantiate the implemented ontologies according to the XML hardware configuration, an ontology interface with an algorithm for automatic instantiation (denoted as hardware config parser in Fig. 2) is realized based on Apache Jena [26, Version 2.11.1]. Furthermore, the implemented ontology interface contains the inference engine as well as the handling of the SPARQL rules in order to query the ontologies and retrieve the required information.

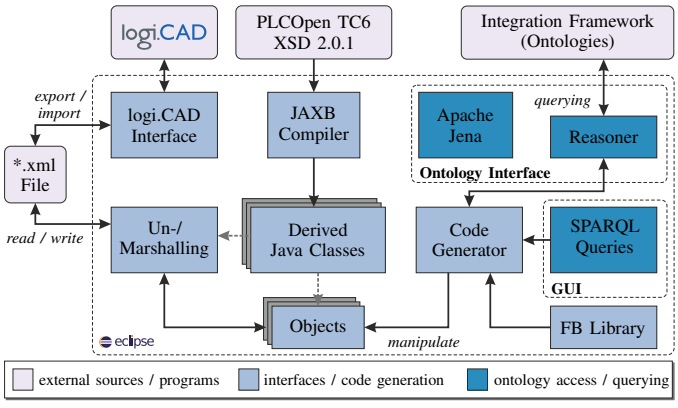


Fig. 6. Overview on the implemented code generation framework. Existing PLC control applications, given as PLCopen TC6-compliant XML files, are unmarshalled to the according Java objects. The resulting object tree is then manipulated by the code generator according to the information given in the ontologies. The behavior of the code generator can be defined by SPARQL queries which in turn can be defined through a graphical user interface.

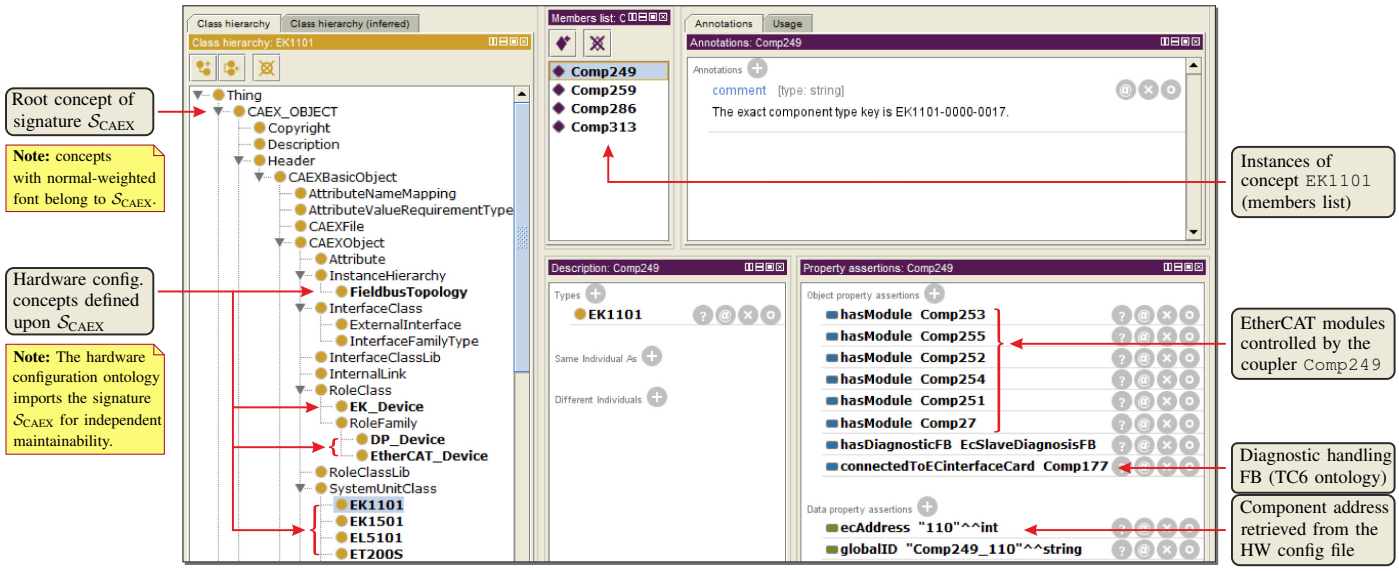


Fig. 7. Implementation example of the ontology in Protégé with concept instances for the hot rolling mill as described in Sec. V-A. The concepts for describing fieldbus topologies (bold font) are defined upon the CAEX ontology concepts (normal-weighted font) and extend the standardized and generic concepts. These concepts are then instantiated according to the given hardware configuration file and the information can be retrieved via SPARQL queries from the ontologies.

An example of the implemented and automatically instantiated ontologies is depicted in Fig. 7. It shows how the concepts presented in Fig. 4 are defined upon the implemented CAEX ontology. Here, the classes of the signature $\mathcal{S}_{\text{CAEX}}$, which is part of the sub-ontology $\mathcal{O}_{\text{CAEX}}$ of the entire reference ontology, are denoted with normal-weighted font. The concepts which are part of the electrical engineering ontology and represent fieldbus topologies and their typical communication devices are represented with bold font. Furthermore, it can be seen how a hardware configuration is mapped onto the defined ontologies in terms of ontology instances (see individuals in the members list in Fig. 7). For instance, the individual `Comp249` is an instance of the ontology concept `EK1101`, which itself is defined as a subclass of the classical AutomationML concept `SystemUnitClass`. The component name for the EtherCAT coupler, the address, as well as the topological information (assigned modules and other EtherCAT slaves) are retrieved from the hardware configuration file. However, the assigned diagnostic FB is retrieved from an additional configuration file of the code generator. This information has to be entered by the user and cannot be retrieved automatically from the hardware configuration file, since diagnostic information of different remote I/O devices could be handled by different FBs. Each fieldbus device has an assigned diagnostic handling FB which can be retrieved through the object property `hasDiagnosticFB`. This object property links the specific fieldbus device with a single instance of the concept `Block` in the automation ontology. Here, the concept `Block` represent a definition of a FB with the entire interface specification according to the standard IEC 61131-3. An instance of this concept in turn specifies one of the FBs depicted in Fig. 5 and additionally references an XML file in the software component library as depicted in Fig. 2 and discussed in Sec. IV-A.

D. Diagnostic Code Generation Process

As mentioned in Sec. III-C, the transformation rules (3) can be implemented in the RDF query language SPARQL. While the definition of the query part of the rules can easily be achieved with the standard SPARQL notation, the implementation of the code generation part (right-hand side of rule (3)) can be very complex and time-consuming. Consider, for example, the case of instantiating a new diagnostic handling FB. To achieve this, the consequent $\text{Con}(r_i)$ of a code generation rule would represent again a set of INSERT SPARQL queries to instantiate the block concept accordingly and, in case of interconnections with other FBs, instantiate those interconnections and relate them with the inserted FB via SPARQL CONSTRUCT statements.

In order to reduce the complexity and the engineering effort for the SPARQL queries, code generation templates are implemented as ARQ functions for Apache Jena. In general, Apache Jena ARQ functions enable the implementation of customized querying or data transformation functions in the programming language Java, which can afterward be directly registered and applied within the SPARQL queries for code generation. A detailed description of an exemplary implementation of such query and code generation rules is given in the next section.

V. Application and Evaluation

To evaluate the proposed rule-based code generation approach, it is applied to the control architecture of a hot rolling mill. The corresponding hardware configuration, used as input for the code generation process, has been exported as Excel sheets from a company-specific fieldbus planning tool.

A. Control Architecture of the Hot Rolling Mill

The control architecture consists of different types of decentralized controllers. One sub-network is based on Profibus DP with in total 224 components and five remote I/O stations (Siemens ET 200S), where the topology is similar to the one depicted in Fig. 1. The devices of the other sub-network communicate through EtherCAT where 75 components and five EtherCAT-couplers (Beckhoff EK1101 and EK1501) are installed. Both networks are connected to an industrial PC with corresponding fieldbus interfaces and an installed Soft-PLC (logi.RTS of the vendor Logi.cals) for controlling the entire hot rolling mill.

B. Evaluation of the proposed Approach

In order to implement the software to handle the diagnostic information of each device mentioned above, the software engineer typically has to select an appropriate diagnostic handling FB depicted in Fig. 5, instantiate it and at least parametrize the inserted FB with the respective device address given in the hardware configuration file. This represents a time-consuming, tedious task and is also error-prone when, for example, the software has to be adapted according to changes in the hardware configuration (danger of forgotten changes). To prove the effectiveness of the code generation approach, three different use cases have been implemented. These are (i) generating complete new PLC applications for handling the diagnostic information, (ii) inserting additional FBs to existing software, and (iii) deleting FBs from PLC software in case of subsequent changes of the hardware configuration.

1) Generation of new Applications: For the generation of complete new PLC applications to handle the diagnostic information of fieldbus devices, an empty TC6-compliant project XML is applied wherein the required FBs are automatically instantiated. The implemented diagnostic FBs, described in Sec. IV-A and depicted in Fig. 5, also exist as TC6-compliant XML representations in the software component library and have been exported from the engineering tool logi.CAD.

For automatic generation of the diagnostic handling code, the ontologies are initially instantiated with the hardware configuration of the control architecture, as described in the previous section (see also Fig. 7). In Listing 1 an exemplary transformation rule for generating the according diagnostic handling code for EtherCAT devices is listed. Therein, lines 1 and 2 are defining the prefixes which can be used in the following query definition, where the implemented code generation templates are registered under the ARQ function prefix in line 2. The outer part of the nested SPARQL statement (lines 3–8) represents the right-hand side of the rule (3) and retrieves the required information for the code generation process. In this case, the query searches for EtherCAT couplers of the types `EK1101` and `EK1501` (lines 5 and 6), retrieves its address (line 7), gets the information of the required FB (line 8), and then delivers the queried information set to the code generation part of the rule (line 9). The implemented ARQ function *ecInstance* of line 9 then loads the according XML file of the desired FB from the software component library (cf. Fig. 2). The loaded XML file is specified by the retrieved information in line 8 of Listing 1, since the queried object property references the concrete type of the FB in the automation ontology which again contains the file reference. By loading the software component the interface definition and further details of the FB like layout information (required

Listing 1. SPARQL QUERY TO GENERATE ETHERCAT DIAGNOSTICS.

```
1 PREFIX hw: <http://acin.ac.at/mst/hwconfig#>
2 PREFIX arq: <http://jena.hpl.hp.com/ARQ/function#>
3 SELECT ?coupler ?address ?fb
4 WHERE {
5       values ?type { hwc:EK1101 hwc:EK1501 } .
6       ?coupler rdf:type ?type .
7       ?coupler hw:ecAddress ?address .
8       ?coupler hw:hasDiagnosticFB ?fb
9       SELECT (arq:ecInstance(?coupler,?address,?fb)
            as ?returnValue)}}
```

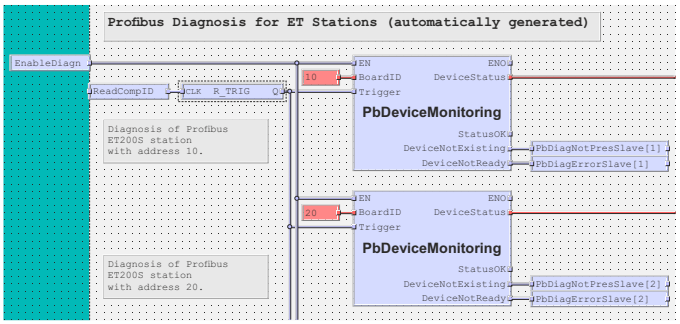


Fig. 8. Snippet of the automatically instantiated and parametrized function blocks (FBs) for Profibus DP diagnosis on device level. It shows the internal functionality of an instance of the *PbNetworkMonitoring* FB depicted in Fig. 5

for automatic placement of the FB) are accessible. Thus, the code generator can insert an instance of the FB into the PLC application template and parametrize it with the retrieved data. A snippet of the resulting FB network for handling diagnostic information on device level for the Profibus DP part of the hot rolling mill is depicted in Fig. 8.

2) Insert FBs to existing Software: Inserting additional FBs to an existing PLC software project in logi.CAD works similar as the previously described use case. The main difference is that in this case the hardware configuration parser identifies the changes with respect to the previously imported configuration file, instantiates the concepts for the new hardware components in the ontology and triggers the code generation process with the additional subset of the ontology. Therefore, the additional concepts are marked by setting a data property flag assigned to the concepts. The code generator then automatically exports the respective PLC software project via the TC6 XML parser in Fig. 2 from logi.CAD, inserts the new FBs at the next free position in the FB network, resets the data property flag of the corresponding ontology concept, and finally re-imports the adapted XML project description to the engineering tool. Access of the implemented ARQ functions for code generation to the ontology for resetting flags is provided by an implemented ARQ function factory (singleton implementation).

3) Delete FBs from existing Software: Deleting FBs is implemented accordingly to the previously described generation process. The hardware configuration parser instead sets the *delete* flag of a concept (representing a hardware device in the configuration) and then triggers the code generation process for deleting the respective FBs in the existing PLC software. The FBs which should be deleted are uniquely identified by their ID and the parametrized address. After deleting the corresponding FBs, the XML project description is re-imported. This, together with generating new applications and adding FBs enables the automatic generation of the entire diagnostic handling code.

C. Discussion

The benefits of the presented code generation approach are manifold. For instance, compared to the automatic generation approach presented by Estévez et al. [27], which additionally introduces a markup language for describing industrial control systems, the knowledge-based approach can cope with standard models applied in industries without adding additional steps or models to the engineering process. The only additional language is the ontology query language SPARQL for specifying

the internal behavior of the code generation process. However, the complexity of the SPARQL queries needed for code generation has already been reduced by applying pre-defined ARQ functions which hide the complexity of the generation of the PLCopen TC6 XML from the user. Additionally, the specification of the SPARQL queries can, for instance, be further simplified by applying graphical SPARQL editors or by defining an domain-specific language upon SPARQL for simplifying the rule engineering. However, the effort for implementing the diagnostic handling software has been significantly reduced by the presented approach since repetitive task can be encoded into a single SPARQL query or code generation pattern which is then just executed on the instantiated ontologies. Several company tests showed that the pure implementation time can be reduced from the range of some days to the minute range, which of course depends on the size of the engineered plant. Once the code generation rules like the one shown in Listing 1 are defined, the only remaining task for similar plants is to load the hardware configuration file for automatically mapping it on the ontology-based integration framework and additionally trigger the described code generation process.

VI. Conclusion

This paper presents a knowledge-based approach to automatically generate diagnostic handling code for decentralized PLC-based control architectures. The information about the planned fieldbus topology is formalized in terms of mapping it onto a developed ontology-based integration framework. This ontological base enables the proposed rule-based code generation process. Here, the rules are implemented as SPARQL rules with ARQ functions and enable the code generator to retrieve required information from the ontologies, automatically instantiate FBs according to each projected hardware device, and parametrize the FBs accordingly with the required parameters like, for instance, the fieldbus address of the device.

The herein proposed approach has several advantages. First, repetitive tasks like instantiating and parametrization of FBs in PLC software application can be automatized by encoding them once in SPARQL rules. Those rules can also be re-used in additional projects if the same task has to be performed with different hardware configuration as input. Also the engineering time of implementing the diagnostic handling methods significantly reduces since the task is reduced to specification of the code generation rules. Furthermore, field tests indicate that the proposed approach can reduce the error rate as well as the probability of forgotten changes since the generation process can be triggered automatically on each change in the hardware configuration. An implementation of the code generation approach is currently applied for the engineering of additional plants and represents an important step to decrease the engineering costs while increasing the software quality.

Acknowledgments

The authors would like to thank SMS Siemag AG for providing the data of the hot rolling mill for evaluation purposes. Financial support by Festo AG & Co. KG Research (Esslingen, Germany) is gratefully acknowledged.

References

[1] R. E. Smith and M. Gini, "Reliable realtime robot operation employing intelligent forward recovery," *J. of Rob. Sys.*, vol. 3, pp. 281–300, 1986.

[2] J. Richardsson and M. Fabian, "Modeling the control of a flexible manufacturing cell for automatic verification and control program generation," *Int. Jour. of Flexible Manufacturing Systems*, vol. 18, pp. 191–208, 2006.

[3] F. Serna, C. Catalán, A. Blesa, and J. Rams, "Design patterns for failure management in IEC 61499 function blocks," in *Proc. of the IEEE Int. Conf. on Emerging Technologies & Factory Automation*, 2010, pp. 1–7.

[4] Int. Electrotechnical Commision, "IEC 61499 – Function blocks, Part 1: Architecture," January 2012.

[5] M. Steinegger, A. Zoitl, M. Fein, and G. Schitter, "Design patterns for separating fault handling from control code in discrete manufacturing systems," in *Proc. of the 39th Annual Conf. of the IEEE Industrial Electronics Society*, 2013, pp. 4366–4371.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co. Inc., 1995.

[7] G. Frey, "Automatic implementation of Petri Net based control algorithms on PLC," in *Proc. of the American Control Conference*, vol. 4, 2000, pp. 2819–2823.

[8] H. Flordal, M. Fabian, K. Åkesson, and D. Spensieri, "Automatic model generation and PLC-code implementation for interlocking policies in industrial robot cells," *Control Engineering Practice*, vol. 15, no. 11, pp. 1416–1426, 2007.

[9] R. Drath, A. Fay, and T. Schmidberger, "Computer-aided design and implementation of interlock control code," in *IEEE Int. Conf. on Computer-Aided Control Systems Design*, 2006, pp. 2653–2658.

[10] J. Morbach, A. Yang, and W. Marquardt, "OntoCAPE - A large-scale ontology for chemical process engineering," *Engineering Applications of Artificial Intelligence*, vol. 20, no. 2, pp. 147–161, 2007.

[11] Int. Electrotechnical Commision, "IEC 62714 – Engineering data exchange format for use in industrial automation systems engineering – Automation markup language," June 2014.

[12] E. Prud'hommeaux and A. Seaborne, "SPARQL query language for RDF," http://www.w3.org/TR/rdf-sparql-query/, 2008.

[13] K. Güttel, *Konzept zur Generierung von Steuerungscode für Fertigungsanlagen unter Verwendung wissensbasierter Methoden [german]*, ser. VDI Fortschritt-Berichte. VDI Verlag, 2013, vol. 444.

[14] Int. Electrotechnical Commision, "IEC 61158 – Industrial communication networks - Fieldbus specifications," July 2014.

[15] Int. Electrotechnical Commision, "IEC 61784 – Digital data communications for measurement and control," May 2014.

[16] M. Steinegger and A. Zoitl, "Automated code generation for programmable logic controllers based on knowledge acquisition from engineering artifacts: Concept and case study," in *Proc. of the 17th IEEE Int. Conf. on Emerging Technologies & Factory Automation*, 2012.

[17] User Association of Automation Technology in Process Industry (NAMUR), "NA 35: Handling of process control technology projects," 2003.

[18] G. Frey and L. Litz, "Verification and validation of control algorithms by coupling of interpreted petri nets," in *Proc. of the IEEE Conf. on Systems, Man, and Cybernetics*, 1998, pp. 7–12.

[19] Y. Kalfoglou and M. Schorlemmer, "Ontology mapping: The state of the art," *The Knowledge Engineering Rev.*, vol. 18, no. 1, pp. 1–31, 2003.

[20] L. Däubler, "Über Ontologien und deren Anwendung in der Automatisierungstechnik," *at - Automatisierungstech.*, vol. 57, pp. 93–99, 2009.

[21] Int. Electrotechnical Commision, "IEC 61175 – Industrial systems, installations and equipment and industrial products – Designation of signals," May 2015.

[22] K. Thramboulidis and G. Frey, "An MDD process for IEC 61131-based industrial automation systems," in *Proc. of the 16th IEEE Int. Conf. on Emerging Technologies & Factory Automation*, 2011, pp. 1–8.

[23] R. Drath, Ed., *Datenaustausch in der Anlagenplanung mit AutomationML – Integration von CAEX, PLCopen XML und COLLADA*. Springer, 2010.

[24] Logicals, "Logi.CAD – Open IEC 61131-3 automation platform," http://www.logicals.com/en/portfolio/logi-cad, accessed: 2016-02-22.

[25] Stanford University, "Protégé – A free, open-source ontology editor and framework for building intelligent systems," http://protege.stanford.edu/, accessed: 2016-02-22.

[26] The Apache Software Foundation, "Apache Jena – A free and open source Java framework for building semantic web and linked data applications," https://jena.apache.org/, accessed: 2016-02-22.

[27] E. Estévez, M. Marcos, and D. Orive, "Automatic generation of PLC automation projects from component-based models," *Int. Journal of Advanced Manufacturing Technology*, vol. 35, pp. 527–540, 2007.