A Hierarchical Domain-Specific Language Supporting Variants of CPPS Software and Integrating Asset Administration Shells

Christoph Lehnert, Grischan Engel, Thomas Greiner
Institute of Smart Systems and Services
Pforzheim University
Tiefenbronner Straße 65
Pforzheim, Germany
{christoph.lehnert, grischan.engel, thomas.greiner}@hs-pforzheim.de

Abstract

Cyber-physical production systems (CPPS) are formed by a flexible and heterogeneous system architecture. Therefore, a comprehensive automation software design requires methods for creating and managing automation software variants. In scope of CPPS, existing approaches do not consider the design of variants combined with software structuring principles. In addition, information from Asset Administration Shells (AAS) is not used sufficiently. Therefore, we propose a novel approach for a comprehensive design of automation software variants based on a domain-specific language (DSL). Thereby, software structuring is provided by the use of several layers with different levels of abstraction. Automation software variants are defined on the mentioned abstraction layers using specific language elements. In order to determine appropriate variants of control programs for particular automation systems, information from AAS is used. Finally, the advantages of the proposed approach are demonstrated in the field of process engineering.

Index Terms

software variants, domain-specific language (DSL), asset administration shells (AAS), cyber-physical production systems (CPPS)

I. Introduction

The distributed service-oriented system architecture of cyber-physical production systems (CPPS) enables flexible production. This flexibility requires software variants which are customized to different products and their manufacturing processes. In addition, the individual variants of the automation software must support the distributed service-oriented architecture of CPPS, which makes a comprehensive software design difficult. A contribution to flexible software design for CPPS can be domain-specific languages (DSL) [1], [2], which consider integration of services as well as asset administration shells (AAS) [1], hierarchical abstraction layers [1], [3] and modularization [4].

In context of process technology, software variant consideration affects in particular process modules that implement identical process operations but use different actuator and sensor assemblies for this purpose. They differ with regard to specific properties of the installed assemblies (e.g. sensor type, motor type, etc.) [4]. A common automation software architecture can be used for these modules, but the assembly unit differences require a service and parameter configuration specific to the considered process module. So far, there is no DSL that addresses these requirements and several variants of the automation software must be created and maintained. Programmable logic controllers (PLCs) with their heterogeneous execution platforms coded in IEC 61131 languages [5] must be adapted manually when changing hardware or porting to another system. Existing DSLs are essentially limited to hierarchical abstraction or modularization without supporting the special requirements of automation technology regarding process module-specific software variant consideration and parameterization with language elements. Software variant consideration using information from AAS is not taken into account sufficiently.

The contribution of this paper is a novel software variant solution concept for CPPS. It uses a hierarchical DSL with several abstraction layers and specific language elements for software structuring and definition of software variants. In addition, information from AAS to determine software variants is applied. For this purpose, several services for an automation program are defined using the DSL. The corresponding service configurations for different modules are stored in the AAS. Thus, the AAS instances describe the services which are used depending on the process module type. This information is taken into account for intermediate code generation of the corresponding program flow. The software variant of the appropriate module

This work has been published at 2022 IEEE 17th Conference on Industrial Electronics and Applications (ICIEA), DOI: 10.1109/ICIEA54703.2022.10006111

© 2022 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

is therefore configured on a module-specific basis using information from AAS. Then, the generated control software can be executed on the respective target platform.

The remainder of this paper is structured as follows: Section II gives an overview of related work. Then, in Section III, a novel concept of software variant consideration using a hierachical DSL and AAS are presented. Furthermore, a workflow and algorithms for definition and selection of software variants are introduced. Section IV describes a case study of a fill level control from industrial process engineering, which demonstrates the advantages of the approach. In Section V, we conclude this paper and describe further research potential.

II. Related Work

Variability is required to efficiently extend, adapt, or configure a software system [6]. Rabiser et al. [7] highlight in their research preview paper the lack of systematic approaches and explain why dealing with variability in software-intensive cyber-physical production systems (SiCPPS) is challenging. In industry, the handling of variability currently depends on the domain, expert knowledge and self-developed tools [8]. In many cases, these tools operate with very specific artifacts or hardware and software platforms. To manage variability of automation software such as operating modes, diagnosis or fault handling, the authors of [9] suggest an approach using family models in CPPS. Spellini et al. [10] propose a concept for component reuse in model-based systems engineering to handle the diversity of variants in production lines. A methodology is suggested to extract structure diagrams from existing AutomationML descriptions in order to design system functionalities based on them.

Complexity and changeability of CPPS can be handled by modularisation. The resulting requirements for modular plants in process technology are described in [11]. Modular plants can ensure better reusability of automation software. Reusability and reconfigurability for modular components is also the goal of Sonnleithner et al. [12], who use design patterns based on IEC61499 for this purpose. The approach of [13] also supports code reusability to efficiently enable changes to CPPS.

In [14] an AAS framework is introduced to represent relationships between PLC and other devices. Pethig et al. [15] design an information model based on IEC 61360 to automatically configure a servo motor via an AAS for a PLC. An automatic self-configuration infrastructure for components is enabled by the approach of [16] with an event-driven runtime access to the AAS.

There are several approaches for the software design of CPPS with DSL [17]. A DSL consisting of two abstraction layers is described by Vjestica et al. [18]. There is a master layer for modeling process steps and a detail layer that extends the master layer. Functions are parameterized statically. The introduction of an abstraction layer is also proposed in [3]. Therein, it is combined with a component container infrastructure based on standard system and software models of automation technology.

In [1] we propose a hierarchical DSL for collaborative design of automation software. The basic concept of the DSL is a predefined hierarchical architecture consisting of four abstraction layers, each maintained by different stakeholders such as Process Engineers, Software Architects, Application Developers and Library Designers. On each layer specific automation functionalities are encapsulated by services. Thereby, the degree of details within the layers increases towards the bottom layers and the call hierarchy of services across layers follows in a solely descending pattern. Furthermore, specific language elements for the integration of information from AAS are provided.

As already mentioned, approaches for automation software variant consideration are missing, which are tailored to changing process requirements as well as to distributed service requirements of CPPS and their execution on different process modules. The possibility to design variants of automation software with module specific information from AAS is not considered. Existing DSL approaches have no special language elements to support the consideration of variants.

III. A Novel Hierarchical DSL for Software Variant Consideration Based on AAS

In this paper, we extend our DSL concept in [1] and propose another advantage: the handling of software variants. CPPS have a high degree of individualization, which leads to a wide range of variants in automation software. The software variant consideration must match to the durability of CPPS and be able to support the many changes that occur during its evolution. This makes the development of automation software complex because many dependencies have to be taken into account [9]. Structured modeling and design methodologies are required to manage variance in software [10].

A. *Software Variant Solution using Hierachical DSL and AAS*

In Figure 1, the concept for software variant consideration based on a hierarchical DSL with abstraction layers combined with information from AAS is illustrated. Therein, a DSL program is described as an ordered set $P = \{L_4, L_3, L_2, L_1\}$, with L being a set of all defined services $S_{n,k}$ in P and $L_k \subset L$ representing the set of services on a specific abstraction layer $1 \leq k \leq 4$. In order for P to be valid, it is required that a service is exclusively assigned on a single abstraction layer and each abstraction layer contains at least one service. Thus, it holds that $|L_k| \geq 1$, $|L| \geq 4$ and $L_4 \cap L_3 = \emptyset$, $L_4 \cap L_2 = \emptyset$, $L_4 \cap L_1 = \emptyset$, $L_3 \cap L_2 = \emptyset$, $L_3 \cap L_1 = \emptyset$ and $L_2 \cap L_1 = \emptyset$.

A service with index $n \in \mathbb{N}_0$ on layer k is defined as ordered set $S_{n,k} = \{O, I, f_{n,k}\}$, where ordered set $O = \{o_0, ..., o_t\}, t \in \mathbb{N}_0$ represents output parameters, ordered set $I = \{i_0, ..., i_r\}, r \in \mathbb{N}_0$ represents input parameters and $f_{n,k}$ indicates the assigned

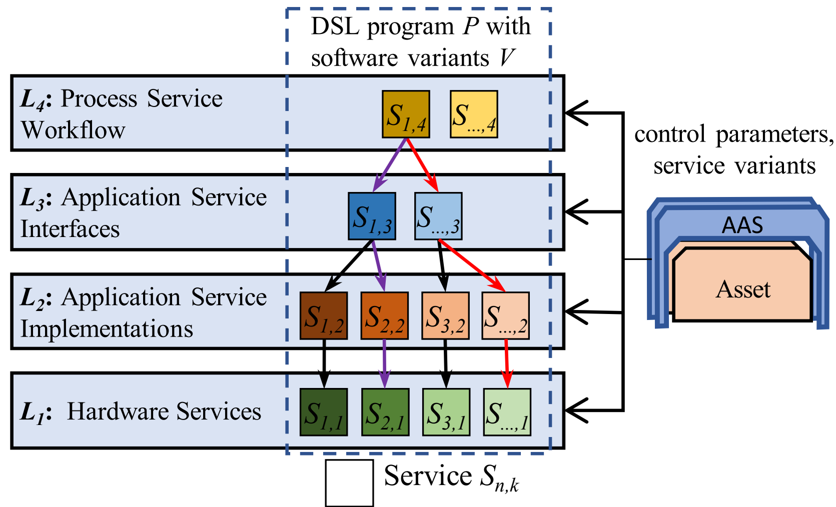

Fig. 1: Concept for software variant consideraton using hierachical DSL and AAS based on [1]

functionality category. Both single output parameter o_w and input parameter i_x are defined as an ordered set $\{d, v\}$ with d as data type and v as value.

Process modules realizing similiar process operations are interchangeable, but consist of different assembly units. The required variability in the automation software is achieved via module-specific service call variants V. In context of P, a software variant, representing a specific program flow, is defined as a dynamic service call graph $V = (L, C)$, where L is the aforementioned set of all defined services and C is a set of edges representing service calls across layers.

A specific software variant is a subgraph $V_s \subset V$ and corresponds to a path of service calls. In the AAS, the set of required services $S_u \subset L$ for each layer is stored for the respective process module instance m. This enables a module-specific service call and thus software variant consideration. The AAS provides relevant information in submodels according to the AAS metamodel [19].

B. Basic Workflow and Algorithms for Definition and Selection of Software Variants

In Figure 2, the basic workflow for generating specific intermediate code of automation software variants based on a DSL program is illustrated. Variants of automation software for a process module type can be defined based on DSL language elements and information from AAS (see Figure 2, step 1). Subsequently, the DSL program is evaluated and thereby a specific software variant is determined using information from AAS (see Figure 2, step 2). Finally, this enables the generation of process module-specific intermediate code (see Figure 2, step 3). This intermediate code is generated offline and is provided to the target platform for execution (see Figure 2, step 4). Wheras the DSL and the intermediate code generator are used on a development system, the intermediate code is executed on the embedded device of a process module.

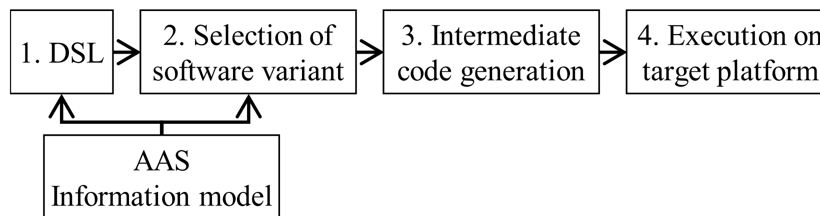

Fig. 2: Workflow for generating and executing intermediate code of specific automation software variants

In the DSL, service groups are a realization of software variant consideration according to Figure 3. This language element is used to group multiple services that provide the same type of functionality but differ in their concrete implementation depending on the process module instance. In order to use a service group in the DSL, it can be defined on an abstraction layer (see Section III-C for specific DSL language elements) and called by the superordinate layer via a service group call. With $P(S_{n,k})$ denoting a specific functionality category $f_{n,k}$ of a service $S_{n,k}$, a service group can be defined as a set $S_g = \{s_{p0}, ..., s_{pj} \mid j \in \mathbb{N}_0,\ s_{pm} \in L_k \wedge P(s_{pm})\}$. Therein, the tuple $s_{pm} = (S_{n,k}, m)$ defines a specific program flow variant based on service implementation $S_{n,k}$ within abstraction layer L_k for the corresponding process module instance m. Process module instances are derived from a common process module type. Thus, each process module instance in the AAS stores the required services for its specific software variant and a service group references all identical services for process module instances of the same type.

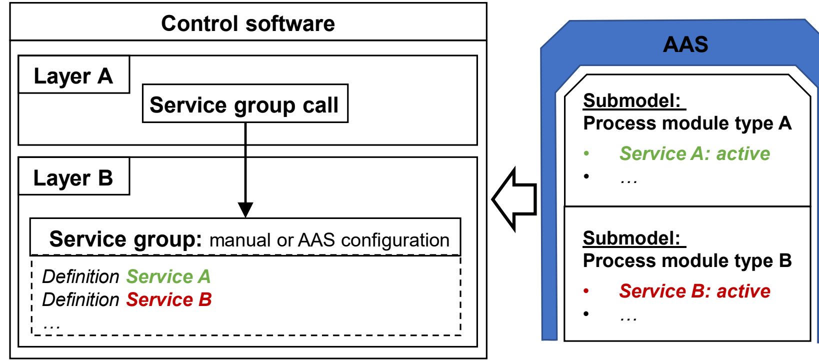

Fig. 3: Concept of service groups combined with AAS and service group calls to achieve software variant modeling

Depending on the information in the respective AAS, different services needs to be called from the superordinate layer. A service group can be called by the superordinate layer using a service group call. Inside a service group the individual service definitions can be configured from the AAS according to the required program flow of the process module type. The structure of the AAS is based on the AAS metamodel [19] and consists of several submodels (see Figure 3). Each submodel represents a process module type whose properties contain the service group configuration required for the module type under consideration. The service group configuration contains the information which services are switched active or inactive.

In scope of a DSL program P, the selection of a specific software variant using information from AAS is defined as

$$\begin{aligned} &\exists p\ (\forall s_{gc}\ (\exists S_g\ (calls(s_{gc}, S_g)\ \wedge\ \exists S_{n,k} \\ &(\exists m\ (existsInAAS(S_{n,k}, m))\ \wedge \\ &existsInProgram(S_{n,k}, P)))) \\ &\ \rightarrow\ replaceServiceGroupCall(s_{gc}, S_{n,k})) \end{aligned} \tag{1}$$

with m being the considered process module instance, $S_{n,k}$ being the determined service, S_g being the service group and s_{gc} being the service group call. Thus, given a specific process module instance, appropriate services can be determined in AAS, service group calls can be replaced with process module specific service calls and finally intermediate code for a control software variant can be generated. The algorithm for selection and intermediate code generation of a specific software variant according to Equation 1 is defined in Algorithm 1.

Algorithm 1 Selection and intermediate code generation for a specific automation software variant

Require: DSL program P, process module instance m
  for each ServiceGroupCall $s_{gc} \in P$ **do**
    $S_g \leftarrow getServiceGroupDefinition(s_{gc})$
    $S_{n,k} \leftarrow getSpecificServiceInAAS(S_g, m)$
    if $S_{n,k} \neq \text{NULL} \wedge S_{n,k} \in P$ **then**
      $replaceServiceGroupCall(s_{gc}, S_{n,k})$
    end if
  end for
  $c \leftarrow generateIntermediateCode(P)$
  return c

C. DSL Language Elements for Service Group and Service Call Definitions

The DSL provides language elements such as loop structures, variable declarations and capabilities to define abstraction levels, services or functions. However, the goal of module-specific variability of the automation software and its parameterization out of the AAS requires additional mechanisms and language elements. One possibility are service group calls as preprocessor statements that have the capability of conditional translation [20]. By linking these preprocessor instructions to the AAS for configuration, the execution logic of the automation software can be adapted to a module-specific variant at compile time. The basic concept of module-specific configurable services with the hierarchical DSL has two essential characteristics:

1) The complexity of software variants are reduced with the help of service groups.
2) A direct link to the AAS is provided, allowing easy access to service group configurations.

To be able to operate these characteristics with the DSL, special language elements are designed for this purpose. The concept of configurable services in the DSL intends to group thematically similar or related services with the help of declarative

annotations. This clustering can then be called up by the superordinate layer as a service group call. Inside these service groups, the corresponding services can be activated or deactivated depending on the module configuration. The service group configured in this way then affects the translation to intermediate code as a preprocessor instruction. With this concept, the developer can manage software variants to specific modules without maintaining different code bases. The conditional execution logic for configuring the service group call is declared statically and manually before execution time or is fetched from the AAS.

Table I gives an overview of selected language elements of the DSL for specifying layers and interacting with service groups or the AAS. The other language elements such as data types or process service defintions are not listed due to limited space. Each language element is described by the respective action keyword, highlighted in italics and the corresponding variable(s). A servicegroup can be defined by using the "**define** *servicegroup*" keywords, followed by the the name of the service group as well as curly brackets with the service definition. Using the call-keyword, followed by the layer name, a dot and the corresponding service group name, it is possible to call a servicegroup within a specific layer. **@inject** and the name of the AAS with curly brackets is the definition structure to fetch variables, service group configurations or other data structures of the respective AAS on demand.

TABLE I: Selection of language elements for the specification of layers and for interaction with service groups

DSL element	**Description**
Layer_ProcessServiceWorkflow { }	Specify layer 4 and the contained process service flow
define serviceGroup name() { }	Defines the individual servicegroup
call LayerName.serviceGroupName()	Calls a process servicegroup inside a specific layer
@*inject* ... { }	Fetches a variable or servicegroup on demand from the AAS via a communication protocol

In order to link a service group to an AAS the **inject** keyword can be used. The syntax of this keyword is as follows: **@inject serviceGroup** *name* **from aas-address**="...". Thereby, the address specifies an instance of an AAS (e.g. an OPC UA endpoint server address) which stores the service group information with active and inactive services. A service group call is realized using the call keyword followed by the corresponding layer and service group names: **call** *layerName.serviceGroup*().

The syntax of the declarative annotation inside a layer for manual configuration of a service group is shown in Listing 3. This configuration can be called by the layer above with the corresponding language element "**call** LayerName.serviceGroupName()" from Table I.

```
1  define serviceGroup serviceGroupName(){
2      serviceNameA / active
3      serviceNameB / inactive
4      serviceNameC / ...
5      ...
6  }
```

Listing 1: Syntax of the specific execution logic by declarative annotations for service configuration

The configuration is either done manually (Listing 1) or is automatically retrieved from the AAS using the corresponding language element "**@inject serviceGroup** *name* **from aas-address**="..."{ } like in listing 2. Both, manual service group definitions and the injects are processed by the preprocessor and thus control the intermediate code generation of a process module-specific program variant.

```
1  @inject serviceGroup serviceGroupName from aas-address = "..."{
2      serviceGroupConfDSLName = serviceGroupConfAASSubModelName
3  }
```

Listing 2: Syntax to fetch a service group configuration out of the AAS

D. Intermediate Code Generation and Execution on the Target Platform

As depicted in step 3 of the workflow in Figure 2, the automation software developed in the DSL is translated into variant specific intermediate code. This code represents the respective program flow and serves as executable model on the target platform. Thereby, it is represented by an inter-control flow graph (ICFG) [21]. Formally, an ICFG is a directed Graph $G = (V, A)$, where V is a set of vertices representing programming expressions (e.g. service calls, if-statements, while-statements

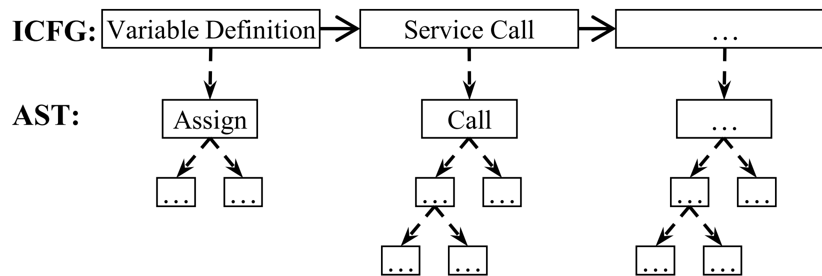

Fig. 4: Conceptional representation of an ICFG with attached AST

etc.) and A is a set of edges representing control flows. Arithmetic expressions in programming statements are represented by an abstract syntax tree (AST) [22] and are part of the corresponding vertices. Figure 4 illustrates the described structure.

The preprocessor decides which parts are generated in the ICFG depending on the service group configuration. For execution (Figure 2, step 4) of the ICFG the target platform applies a depth-first traversal [23] which is extended to evaluate and execute each programming expression of the DSL program.

IV. Case Study: Fill Level Control Service in Process Engineering

To demonstrate the benefits of the presented approach, an exemplary filling process of a reactor inside an agitation module is described. As illustrated in Figure 5, two types of agitation modules g and h are considered, each of which has a pump, a reactor, a level measurement system and an agitation motor. The basic process logic is identical, so that a common automation

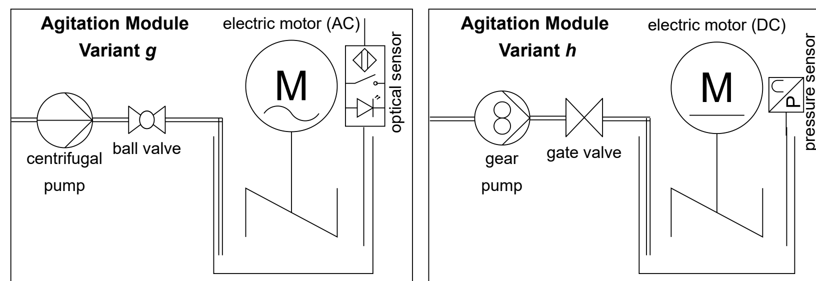

Fig. 5: P&I diagram of both agitation module variants

software architecture can be used. However, the agitation modules differ in the type of level measurement due to different level sensors. In addition, they have different pump characteristics, which is why a customized pump control is used in each case. Due to these differences, a customized control loop, sensor data preprocessing and control algorithms are required for each process module. The required services for fill level control for each process module are modeled as service group configurations in AAS. Consequently, agitation module g uses an optical measurement service and agitation module h applies a hydrostatic fill level sensor service.

The DSL program definition for the described automation software variants is illustrated in Listing 3. In line 2, layer 3 (Figure 1, Application Service Interfaces) is specified. Therein, the service *fillingControl* with the parameter *fillLevel* is defined. Furthermore, it contains, the service group call of *fillLevelControl* (line 6) on layer 2 (Figure 1, Application Service Implementations). On layer 2, in lines 15 to 18 the services for both agitation modules are defined. In order to model the individual program flow for each process module, the service group configuration *fillLevelControl* from the AAS is injected (line 13).

```
1   //Application service interfaces layer definition
2   Layer_AppServiceInterfaces moduleAppInterface {
3     //Interface service defintion
4     define interfaceService void fillingControl(declare int fillLevel) {
5     //Service group call
6     call implService appImpl.fillLevelControl (fillLevel)
7     }
8   }
9
10  //Application service implementations layer definition
11  Layer_AppServiceImplementations appImpl {
12   // Get the service group configuration from AAS
13   @inject serviceGroup fillLevelControl from aas-address = "..."
14   //Impl. service defintion 1 of service group fillLevelControl
15   define implService void fillLevelOptical(declare int fillLevel)
```

```
16      {...}
17      //Impl. service defintion 2 of service group fillLevelControl
18      define implService void fillLevelHydrostatic(declare int fillLevel)
19      {...}
20      ...
21    }
```

Listing 3: Example of a fill level control with a service group configuration linked to an AAS and a respective service group call

Based on the described DSL program and information from AAS, the required process module-specific services are configured and thus influence the generation of individual intermediate code. By the module type specific service group instruction only the active services are transferred to the intermediate code of the ICFG and executed on the target platform. The intermediate code is an inter-control flow graph (ICFG). The generated ICFG for each process module is illustrated in Figure 6. As depicted, the program flow differs depending on the type of agitation module. Basically, this model is equivalent to an adjacent list. The process module now interprets the different types of vertices, which represent individual programming expressions of the DSL. The AST attached to the vertices is resolved by the target platform with depth first traversal.

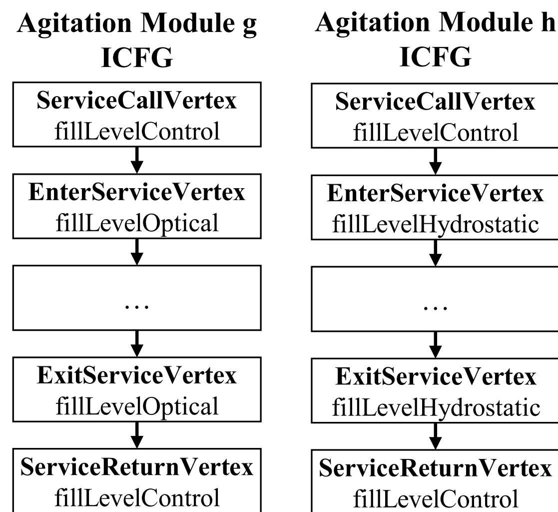

Fig. 6: Generated ICFG depending on the agitation module type

The Eclipse IDE is used for the implementation. The DSL grammar is defined in Xtext, the validator and generator in Xtend. The Modeling Workflow Engine 2 and the Eclipse Modeling Framework are used in the background [24]. The intermediate code is converted to an XML data structure and then parsed and interpreted. This occurs on the target platform, the agitation module, which is developed using the c programming language.

V. Conclusion and Outlook

The development of automation software for CPPS requires the support of process module-specific software variants allowing a variable system architecture. For this purpose, software design methods are necessary that support the handling of automation software variants. In addition to the service oriented structure of CPPS, process module-specific properties must also be taken into account. The AAS contains process module-specific information which can be used for parameterization and customization of the software variants. In order to achieve this, we propose the usage of a hierarchical DSL with special language elements for variant consideration integrating the AAS. Compared to other related work, we provide a DSL for the systematic design of software variants in CPPS, which is not limited to single domains or reliant on expert knowledge. With this concept, the software designer can customize the software to specific process modules without having to maintain different code bases. The applicability is demonstrated by using a filling control application from the process industry.

The application of a DSL, which integrates the AAS for variant consideration, can be seen as a further essential development component for CPPS.

The applicability of the approach is shown on a module from process engineering, but can also be applied to modules from manufacturing engineering. The current approach focuses on a textual DSL. For an intuitive graphical programming of the DSL, a graphical representation concept needs to be determined. In addition, further research involves time decoupling of layers and semantic programming of parallel services. These aspects are not part of our contribution.

Acknowledgment

This work is funded by the Federal Ministry of Education and Research Germany as part of the project IMPACT ”Innovative Methods for Programming of Automation Control Technology” under grant number 01IS19031B.

References

[1] C. Lehnert, G. Engel, H. Steininger, R. Drath, and T. Greiner, "A hierarchical domain-specific language for cyber-physical production systems integrating asset administration shells," in *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2021, pp. 01–04.

[2] K. Meixner, J. Decker, H. Marcher, A. Lüder, and S. Biffl, "Towards a Domain-Specific Language for Product-Process-Resource Constraints," in *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1, 2020, pp. 1405–1408.

[3] G. Music, B. Heinzl, and W. Kastner, "Ava: A component-oriented abstraction layer for virtual plug&produce automation systems engineering," *Journal of Industrial Information Integration*, vol. 26, p. 100251, 2022.

[4] "Automation requirements relating to modularisation of process plants," *NAMUR NE 148*, 2013.

[5] "Iec 61131 - programmable controllers - all parts," International Electrotechnical Commission, Standard, 2022.

[6] K.-C. K. Rafael Capilla, Jan Bosch, *Systems and Software Variability Management*. Springer, 2013.

[7] R. Rabiser and A. Zoitl, "Towards mastering variability in software-intensive cyber-physical production systems," *Procedia Computer Science*, vol. 180, pp. 50–59, 2021.

[8] A. Gutierrez, L. Sonnleithner, A. Zoitl, and R. Rabiser, "Approaches to mastering variability in software-intensive cyber-physical production systems," in *e & i Elektrotechnik und Informationstechnik volume 138*, 2021, pp. 321–329.

[9] B. Vogel-Heuser, J. Fischer, D. Hess, E.-M. Neumann, and M. Würr, "Managing variability and reuse of extra-functional control software in cpps," in *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2021, pp. 755–760.

[10] S. Spellini, S. Gaiardelli, M. Lora, and F. Fummi, "Enabling component reuse in model-based system engineering of cyber-physical production systems," in *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2021, pp. 1–8.

[11] "Automation engineering of modular systems in the process industry - general concept and interfaces," *VDI/VDE/NAMUR 2658 Blatt 1*, 2022.

[12] L. Sonnleithner, B. Wiesmayr, V. Ashiwal, and A. Zoitl, "Iec 61499 distributed design patterns," in *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2021, pp. 1–8.

[13] B. Vogel-Heuser, E. Trunzer, D. Hujo, and M. Sollfrank, "(Re)deployment of Smart Algorithms in Cyber-Physical Production Systems Using DSL4hDNCS," *Proceedings of the IEEE*, vol. PP, pp. 1–14, 01 2021.

[14] S. Cavalieri and M. G. Salafia, "Asset Administration Shell for PLC Representation Based on IEC 61131–3," *IEEE Access*, vol. 8, pp. 142 606–142 621, 2020.

[15] F. Pethig, O. Niggemann, and A. Walter, "Towards Industrie 4.0 compliant configuration of condition monitoring services," in *2017 ieee 15th international conference on industrial informatics (indin)*. IEEE, 2017, pp. 271–276.

[16] M. Wenger, A. Zoitl, and T. Müller, "Connecting PLCs With Their Asset Administration Shell For Automatic Device Configuration," in *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*, 2018, pp. 74–79.

[17] A. Wortmann, O. Barais, B. Combemale, and M. Wimmer, "Modeling languages in Industry 4.0: an extended systematic mapping study," *Software and Systems Modeling*, vol. 19, no. 1, pp. 67–94, 2020.

[18] M. Vještica, V. Dimitrieski, M. Pisarić, S. Kordić, S. Ristić, and I. Luković, "Multi-level production process modeling language," *Journal of Computer Languages*, vol. 66, p. 101053, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2590118421000320

[19] "Examples of the asset administration shell for industrie 4.0 components – basic part - continuing development of the reference model for industrie 4.0 components," *ZVEI*, 2017.

[20] B. S. Institution, *The C Standard: Incorporating Technical Corrigendum 1*. Wiley, 2003.

[21] J. Knoop, *Optimal Interprocedural Program Optimization: A New Framework and Its Application*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003.

[22] R. Harper, *Practical Foundations for Programming Languages*. Cambridge University Press, 2016.

[23] M. Goodrich, R. Tamassia, and M. Goldwasser, *Data Structures and Algorithms in Java*. Wiley, 2014.

[24] L. Bettini, *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.