



Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorthesis

Julian Bonas

Conceptual Process Integration of a Machine
Learning Applications with AutomationML and
OPC UA

Julian Bonas

Conceptual Process Integration of a Machine
Learning Applications with AutomationML and OPC
UA

Bachelorthesis eingereicht im Rahmen der Bachelorprüfung
im Studiengang Mechatronik
an der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr.Ing. Holger Gräßner
Zweitgutachter : Christian Böhlmann M.Eng.

Abgegeben am 20. Dezember 2019

Julian Bonas

Thema der Bachelorthesis

Konzeptionelle Prozessintegration einer Machine Learning Anwendung mit AutomationML und OPC UA

Stichworte

Industrie 4.0, OPC UA, AutomationML, Machine Learning, Automation , Skill-Basierte Prozesssteuerung

Kurzzusammenfassung

Die Idee einer Industrie 4.0 verfolgt die Vision von Produktionssystemen, deren Komponenten vollständig miteinander vernetzt und in ihre Umgebung integriert sind. So werden diese Systeme in die Lage versetzt, ihre Informationen und Fähigkeiten zu teilen. In dieser Arbeit soll gezeigt werden, wie diese Ideen sich auch auf reine Softwarekomponenten anwenden lassen. Es wird dafür eine Machine Learning Applikation mittels OPC UA und AutomationML beschrieben und ein Konzept für die Integration in einen Automatisierungsprozess vorgestellt.

Julian Bonas

Title of the paper

Conceptual Process Integration of a Machine Learning Applications with AutomationML and OPC UA

Keywords

Industry 4.0, OPC UA, AutomationML, Machine Learning, Automation, Skill-Base Engineering

Abstract

The idea of Industry 4.0 pursues the vision of production systems whose components are fully interconnected and integrated into their environment. This enables these systems to share their information and capabilities. In this thesis it shall be illustrated how these ideas can also be applied to pure software components. A machine learning application using OPC UA and AutomationML is described and a concept for the integration into an automation process is presented.

Table of contents

List of figures	V
List of tables	VII
Akronyme	VIII
Abbreviations	VIII
1. Introduction	1
1.1. Motivation	1
1.2. Structure of this work	3
2. Related Work	4
2.1. OpenMOS	5
3. Fundamental Principles	8
3.1. Industry 4.0	8
3.1.1. Reference Architecture Model	8
3.1.2. Asset Administration Shell	9
3.2. Information Modeling	10
3.2.1. Semantics	11
3.3. Product, Process and Resource concept	12
3.4. Skill-based Engineering	13
3.5. OPC Unified Architecture	14
3.5.1. Core Concepts	16
3.5.2. Publish Subscriber Mechanism	21
3.5.3. Programs	22
3.5.4. Notations	23
3.6. AutomationML	24
3.6.1. Instance Hierarchies	26
3.6.2. System Unit Classes	27
3.6.3. Roles	27
3.6.4. Interfaces	27
3.6.5. Communication Modeling	28

3.6.6. Data Variables	28
3.7. AutomationML and OPC UA	28
3.8. Machine Learning	29
4. System and Concept	31
4.1. Task Description	31
4.2. Riveting System Description	32
4.3. Current Implementation	35
4.3.1. Break Detection Algorithm	35
4.3.2. Possible Integration Scenarios	35
4.3.3. Parameters and Application Settings	37
4.4. Problem Description	37
4.4.1. Device Dependencies	38
4.4.2. Application Configuration	39
4.5. Implementation Objective	39
4.5.1. Modularization	40
4.5.2. Information Modeling	40
4.5.3. Uniform Access of Skills	41
4.5.4. Configuration	41
4.5.5. Proposed Integration Concept	41
5. Modeling	44
5.1. Common Semantic Model	44
5.1.1. UML Model	44
5.1.2. Skill Taxonomy	45
5.2. AutomationML Model	45
5.2.1. Role Definitions	46
5.2.2. Interface Definitions	47
5.2.3. System Unit Class Definitions	48
5.2.4. Process Instance Hierarchy	50
5.2.5. Model Conversion to OPC UA	52
5.3. OPC UA Models	53
5.3.1. Modeling Software	54
5.3.2. Type Definitions	54
5.3.3. Break Detection Application Model	59
5.3.4. Sensor Models	60
5.3.5. Model Conversion to C Code	61
6. Client Implementation	64
6.1. Resolving Skill Requirements from theAML Server	64
6.2. Resolving the Acceleration Sensing Skill	67

6.3. Test Case	68
7. Discussion and Further Work	70
References	72
A. Skill Type Definition in OPC UA	77
B. Break Detection Definition in OPC UA	78
C. Configuration Client Program	79

List of figures

2.1. openMOS architecture overview	6
3.1. Reference Architectural Model for Industrie 4.0 (RAMI)	9
3.2. Administration Shell sub models overview	10
3.3. PPR concept	12
3.4. Two excerpts in UML for example skill hierarchies	14
3.5. OPC UA Specifications overview	15
3.6. OPC UA meta model in UML	16
3.7. Example server object tree	17
3.8. Example object tree with type definitions and instances	19
3.9. OPC UA notation overview	23
3.10. Example <i>Namespace</i> in OPC UA notation	24
3.11. Parent-child and inheritance relations between classes and objects.	25
3.12. AML Editor main components and workflow	26
3.13. Example OPC UA server object tree generated from AML	29
3.14. ML pipeline by Microsoft Azure	30
4.1. Structural design of aircraft fuselages	31
4.2. Hi-Lok connection principle	32
4.3. Robotic system placing the Hi-Lok pins	33
4.4. Riveting system	34
4.5. Break detection configuration with Bosch XDK	36
4.6. Break detection configuration with UR10	37
4.7. Break detection concept	43
5.1. Semantic model in a UML class diagram	45
5.2. Skill taxonomy in UML	46
5.3. Excerpt from the role definition of data variables	47
5.4. OPC UA server definition in AML	47
5.5. AML requirement interface.	48
5.6. AML system unit classes for skills and requirements	48
5.7. AML system unit classes for modules	49
5.8. Instance hierarchy with the UR10 as an acceleration source.	50

5.9. Parameter definitions for the application requirements	51
5.10. Instance hierarchy excerpt with the Bosch XDK instead of the UR10.	52
5.11. Resulting object tree in the OPC UA server.	53
5.12. Overview SiOME	54
5.13. Module types defined in OPC UA	55
5.14. Requirement types defined in OPC UA	56
5.15. Skill state machine	57
5.16. Skill types defined in OPC UA	58
5.17. ML type in OPC UA	60
5.18. Simplified view on the break detection server	61
5.19. UR10 sensor model	62
A.1. Object tree of the skill type definition in OPC UA	77
B.1. Object tree of the skill type definition in OPC UA	78

List of tables

3.1. Namespace example	18
4.1. Parameter setting of the break detection	38

Akronyme

AAS	Asset Administration Shell
AML	Automation Modelling Language
AMQP	Advanced Message Queuing Protocol
CPPS	Cyber Physical Production System
I4.0	Industry 4.0
IPC	Industrial PC
KPI	Key Performance Indicator
ML	Machine Learning
MQTT	Message Queuing Telemetry Transport
OPC UA	Open Platform Communications Unified Architecture
openMOS	open Manufacturing Operation System
PPR	Product-Process-Resource
PubSub	Publish-Subscriber
P&P	Plug-and-Produce
RAMI	Reference Architectural Model for Industrie 4.0
RFC	Random Forest Classifier
ROS	Robot Operation System
RTDE	Real-Time Data Exchange
SiOME	Siemens OPC UA Modeling Editor
SORA	Service Oriented Robotics Architecture
SysML	Systems Modeling Language
TCP	Tool Center Point
UDP	User Datagram Protocol
UML	Unified Modeling Language
UR10	Universal Robot 10
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XML	Extensible Markup Language

1. Introduction

Since 2011 the term Industry 4.0 (I4.0) has been minted. Its vision is to enable future systems within the manufacturing domain to be fully integrated and interconnected within its environment. Thus these systems are enabled to share their information and capabilities.

In recent years, researchers have made many efforts to enable these systems. One aspect of these efforts is the aggregation and creation of information. They rely on different information modeling techniques to describe production systems, processes, and products. The main goal of this is to create models that describe and relate all parts in a production systems, so machines can interpret these, and enriched with information that can ultimately be utilized throughout the value-chain, from the supplier to manufacturers and customers, accessible through a digital representation of these models.

Another aspect is the effort to create a standardized communication that utilizes the component models and make them easily accessible. Today many communication protocols and standards are utilized in production systems and are widely varying between manufacturers. While each protocol has its advantages, most of them lack the capability of embedding information, that enables, for instance, application engineers to access the communication participants and explore their functionality. This lack results in great efforts during the integration of new components into a production system.

To provide consistency throughout these efforts, it is required, that resulting models are defined in standards. This is essential in order to provide interoperability within an I4.0 system, participating enterprises and overlapping domains. Resulting I4.0 components should become easily accessible, systems scalable, and the integration of new devices less time-consuming. Future Cyber Physical Production System (CPPS) will be created this way, that are ultimately enabled to dynamically adapt to changes, in order to fulfill quickly changing market needs and product customization requests.

1.1. Motivation

Within the automation domain, many efforts have been made, in order to describe hardware components and their capabilities, utilized in automation systems. The goal is to classify

the Product-Process-Resource (PPR) relationships and capabilities of devices. As a result, products, processes and resources should become easily interchangeable by other products, processes, and resources with the same classifications. Therefore the flexibility of the overall systems increases, as well the ability of the system to determine if all resources are available, needed to accomplish a particular process, in order to create a product.

Overall less effort has been made to describe purely software applications that are not directly related or assigned to any particular kind of hardware. For instance, the capabilities of a robot are mainly described by its ability to move inside its working area. By invoking the movement command on the robot control, the goal pose and type of movement is passed. Subsequently, the robot is moving from its current position to its goal in the given way. Thereby the movement command is directly related to the hardware configuration (kinematics) of the robot. However, while executing the movement, the robot is not aware of its surrounding. This is adequate for static environments and tasks.

Nevertheless, if a robot should be able to work in a dynamic environment, it needs to become aware of what is happening around it and adapt its movements if necessary. Consequently, additional information from sensors is needed to map the environment of the robot. To incorporate the sensor information and generate an appropriate path, additionally, a dynamical path planning program is required. Unlike the moving command, path planning is not constrained to the configuration of one robot and usually cannot be found onto a robot control. The capability of planning the path is not matched with the robot and rather described as an independent thing that is depending on the input of certain chunks of information, coming from different sensors, as well as a start and target pose. As a result, one receives several points, which can be approached by the robots movement command successively.

While the capability of the robot to move can be easily classified as a movement in general, the path planning software application has not yet been approached, to be classified or described in the context of an automation system, and so to enable the flexible integration into systems. Neither have been other pure software applications. That is why, this work will focus on describing a Machine Learning (ML) application in order to integrate it within an automation system, by means of Automation Modelling Language (AML) and Open Platform Communications Unified Architecture (OPC UA), under the considerations of an overall flexible production system.

1.2. Structure of this work

To give an overview of the topics discussed within this work, the main chapters are briefly described below.

Related Work gives an overview of related work regarding I4.0, skill-based engineering, AML, and OPC UA. These define some of the central concerns of this work. Also, the latest research regarding these topics is discussed. Furthermore, a more detailed description of the openMOS project is given, since here are most of the described concepts are already adopted.

Fundamental Principles discusses the fundamental principles of I4.0, skill-based engineering, the PPR relations, OPC UA, AML, and ML. These are fundamental for the subsequent chapters, where these principles are adopted to construct a concept for the integration of a machine learning application into an automation process.

System and Concept describes the system and its context, which will supply the use-case of this work. Also, the current implementation of the ML application is described, as well as the problems that are resulting from this implementation. Based on this explanation, a concept for the integration of the application is proposed.

Modeling here, the previously described concepts are adopted to model all required components that are essential for the integration of the application into the process. It will encompass the creation of a common semantic model. Based on this model, the process is modeled in AML, and then the application models are created based on OPC UA.

Client Implementation demonstrates how an OPC UA client can utilize the information embedded into the AML and OPC UA models. Therefore a simple client implementation in C, based on the OPC UA library open62541, is envisaged.

Discussion and Further Work recapitulates the presented models and implementations. Problems and benefits of these are discussed. Furthermore, objective for future implementations based on the proposed concepts are introduced.

2. Related Work

The current trend in the manufacturing industry is showing a strong trend to flexible component integration, away from fixed shop floor setups. In Germany, this idea is reflected in the concept of I4.0. Here one of the main participants in the group represented by the Plattform-I4.0, accounting for the problems raising with the idea of an I4.0. It is a collective of many industrial companies, societies and research institutes, backed by the German Federal Ministry of Economic Affairs and Energy as well as the Federal Ministry of Education and Research. One of the main concerns is the comprehensive standardization required to provide the flexible structure of future production systems. For the standardization process of I4.0 systems, the Reference Architectural Model for Industrie 4.0 (RAMI) has been introduced by the interdisciplinary workgroup [1]. Based on that, further concerns are directed to the Asset Administration Shell (AAS). The AAS is the basic concept to represent physical objects in the real world, in the digital world. In order to provide interoperability between objects from different vendors, the AAS, as well as the information represented within, should apply the standardization resulting from the RAMI. In order to describe the structure of the AAS universally, the Plattform-I4.0 introduced the overall structure of the AAS as reference for future implementation [2]. Further, it has been described how relationships between I4.0 components can be expressed by means of the different models and information embedded into the AAS [3].

A generic skill model has introduced a common approach of how to describe the functional aspects of I4.0 components in the AAS. While this approach is not new, it comes with challenges regarding the standardized skill models. These challenges have been shown by [4]. Among the first initiatives aiming at standardization of skills for commercial use is the Skill-Pro Project [5][6], which established the formal definition of skills as executable processes. These ideas have been progressed by [7], describing skills as a link within the description of relations between products, process, and resources. [8] created a taxonomy to describe capabilities of automation devices, considering assembly and handling operation, as well as sensory processes. Especially the necessity of standardization of the input and output parameters for invoking skills is stated. The SemAnz project [9], provided a more general description of these relations, not focusing on skills. It provides a semantic framework for modeling within taxonomies for physical resources, function descriptions, and behavior specification, where skills can be seen as elements of the functional description. While most of the approaches focusing on the skill parameter description by means of the process, [10]

introduced a product-oriented approach for matching skills by the parameters included in the product description. The objective of this approach is to reduce the required domain knowledge when describing a process since the parameters required to describe the process are already stated in the product description.

While the approaches mentioned above rely on the use of AML for the description of capabilities of devices, it is also carried unanimously that Open Platform Communications Unified Architecture (OPC UA) is the communication protocol supplying the required functionality. Nevertheless, some of them rely on the generation of OPC UA servers, based on the AML description, to ultimately implement the skills. On the device level, however, this is not compatible with the standardization effort made by the OPC Foundation to describe and access devices. [11] and [12] therefore made an effort to describe skills by means of OPC UA. Both define skills as OPC UA programs, which enable a generic way to execute skills and monitor the execution state.

2.1. OpenMOS

The open Manufacturing Operation System (openMOS) project is a research project funded by the European Commission under the Horizon 2020 program and fulfilled by a consortium of companies from different domains [13]. Its goal is to develop an accessible Plug-and-Produce (P&P) system, in order to allow quick and efficient integration of machines and automation systems within production environments as a prerequisite for the implementation of I4.0 solutions. Therefore the openMOS deliverables consolidate general known ideas for a flexible manufacturing system and assemble these to an overall view of their architecture. The general openMOS architecture is depicted in figure 2.1 and described below. With this description, an overall view of a possible manufacturing system is provided that includes the ideas of a flexible production system. This should give an idea of how the overall architecture of these systems could look like.

There are three main elements in this architecture. The Agent Cloud Platform, the Manufacturing Service Bus, and the Device Adapters. In the Agent Cloud Platform, different kinds of agents can be deployed to serve different tasks within the production. They are accountable for the overall process control and manage the production, transportation, optimization, and resources. The Agent Cloud is, therefore, segmented into three main modules. First of all, the execution layer. Here, for instance, instances with the cyber-physical internal abstraction of devices and resources are deployed as resource agents. It is activated when a new device is plugged into a system and stopped when removed. While it is plugged in, maintenance and other data can be collected and updated within its digital representation. Secondly, there is a repository layer, where historical data from the processes are aggregated for later use of optimization and maintenance prediction. Furthermore, thirdly, there is the, communication layer.

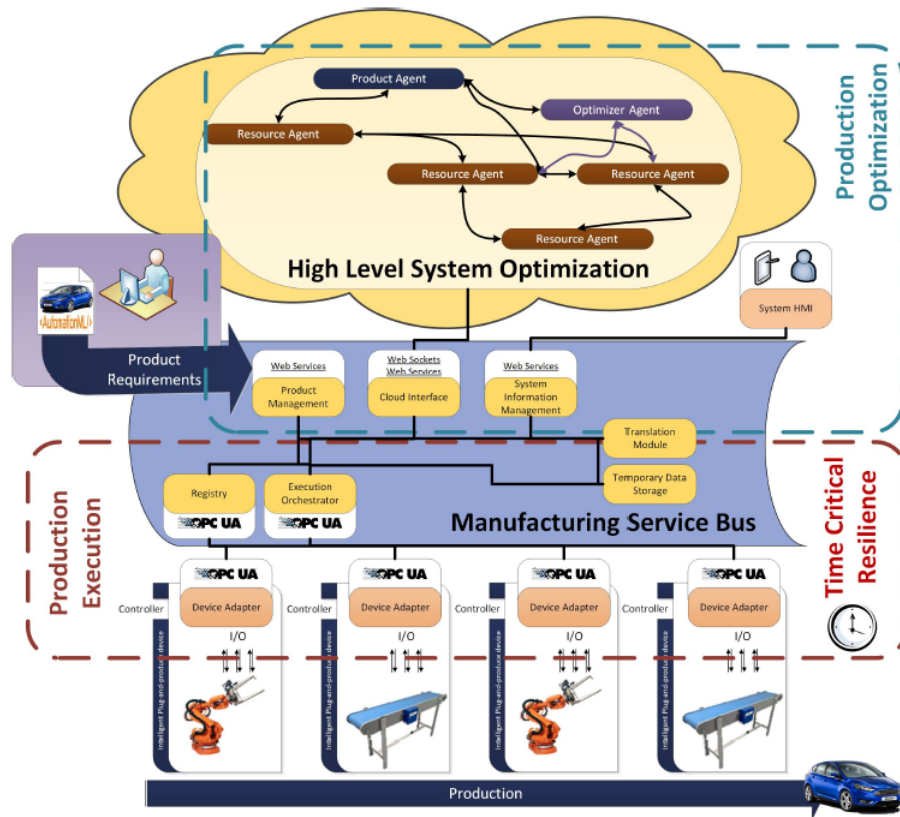


Figure 2.1.: openMOS architecture overview [14]

Within this layer communication for Human-Machine-Interfaces and Manufacturing-Service-Bus-to-Agent communication is managed.

As a link between resources and the cloud, providing the necessary functionalities for task execution and discovery serves the Manufacturing Service Bus. It is separated into three main modules as well. The first module concerns network interfaces employing the provision of required protocols, service, and device communication. The second module is providing some core functionalities for a production system. These are the recipe execution, management of the connected devices, orchestration tasks, and others. At last, there is the database interface module. Its purpose is to store data from processes temporarily, so that it can be post-processed by agents in the cloud. Also, it prevents the loss of data and ensures the continuous operation.

On the lowest level and third main module of the openMOS architecture are the Device Adapters. These wraps the functionalities of the associated devices into an OPC UA server and exposes the devices self-description to the Manufacturing Service Bus. The self-description contains the capability model and configurations, modeled in AML, and then converted to

OPC UA. During conversion, the openMOS project converts the capabilities or skills to OPC UA methods and is thus an extension to the DIN SPEC 16592, which defines the combination and transformation of AML and OPC UA. These methods are allocated in the Task Execution Table and then triggered by the Task Executor from the Manufacturing Service Bus. In order to allocate the Task Execution Table, the process is described by means of Skill Requirements resulting from the Product description. This is done in AML and includes the semantic description of processes, equipment, execution models, and products [15] [16].

3. Fundamental Principles

3.1. Industry 4.0

Industry 4.0 is a term minted by the German government, describing the aspects of industrial production regarding the fourth industrial revolution. Internet of things, digital production, and smart, adaptable processes are part of the vision, a digitized and fully interconnected value chain. New emerging technologies in machine learning and computer hardware enable this vision. However, there is still some effort required in order to determine on how these future systems will emerge.

3.1.1. Reference Architecture Model

The RAMI has been introduced in order to provide a comprehensive reference architecture for the structural description of I4.0 ideas and concepts to integrate I4.0 components [18]. A working group of more than 400 participants from industrial partners and research institutions collaborated to build this model.

The model aims to deliver an overview of the different levels that have to be considered while describing an I4.0 system. As shown in figure 3.1 the RAMI has three axes. A "Layer" axis, that represents different layers of digital representations a "Hierarchy Levels axis", that describes different levels within a fabric or plant, specified by IEC 62264 and complemented by "Product" and "Connected World"; and a "Life Cycle & Value Stream" axis, covering the life cycle of plants and products. With a fixed scale between the different levels from enterprises, automation systems to life cycle management, interoperability should be guaranteed. By interpolating the terminology and different aspects of different domains, a common language to describe things within an I4.0 system can be created. With a common language, manufacturers, customers, and suppliers are enabled to interconnect their systems beyond a companies border and maintain information regarding products and services.

A base for the common language is provided by national and international standards that are already available. These standards are not always well suited in order to describe products, components, and systems, and in some aspects, these are not complete or interfering with

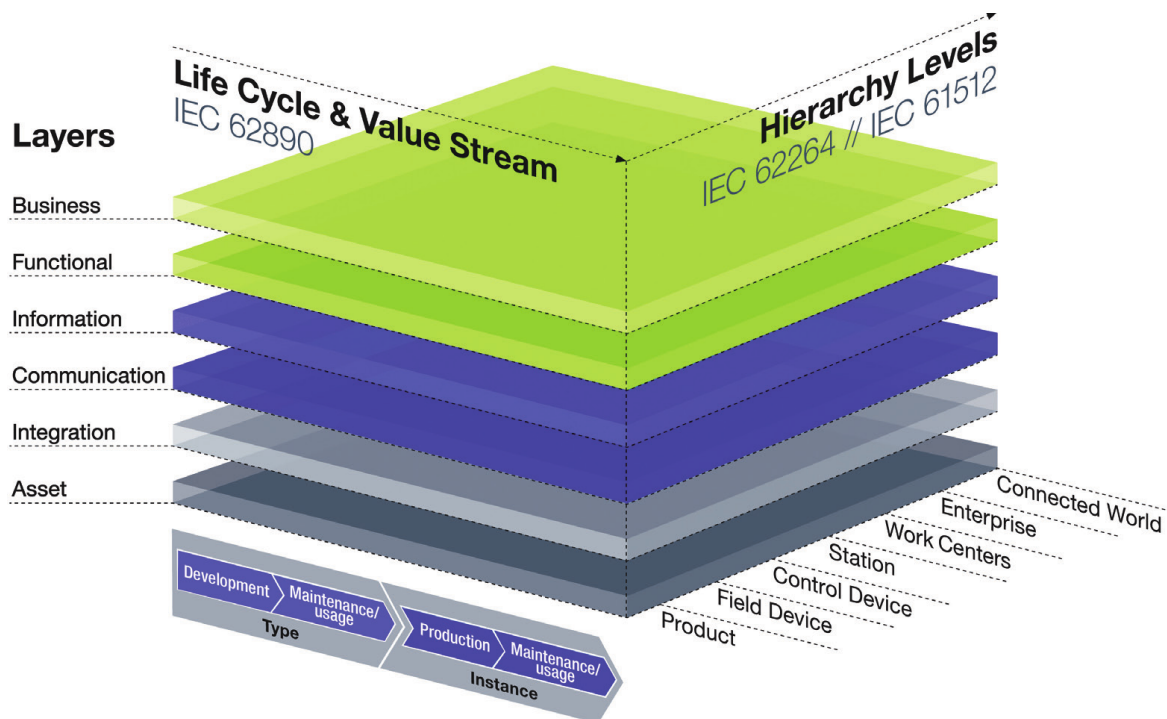


Figure 3.1.: RAMI [17]

each other on different levels. To classify these standards and check for inconsistency and interference with other standards, the RAMI model provides a fundamental base. Nevertheless, for many aspects, new standards will be created, to interpolate between available standards or to fill missing information to apply new I4.0 concepts.

3.1.2. Asset Administration Shell

An I4.0 component is an item that is unambiguous identified and able to communicate. It consists of an asset and an associated administration shell. An asset is an item that has a certain value for an organization and, therefore, a need for individual management. These items can be of physical or virtual nature, such as devices, software, plans, and others [17, S. 12, def. Gegenstand]. Regarding I4.0 components, assets are usually referred to physical objects, with the need of life cycle management.

The Asset Administration Shell (AAS) is the digital representation of an asset. Here all information for communication, usage, maintenance, and others is embedded. Depending on the asset, the AAS can be embedded onto the asset (e.g. device) itself or be only available in a companies cloud. Also, a separated AAS is imaginable, where functionality of assets

is provided in an AAS on a process controller in the field, while the business and life cycle information is managed in an AAS deployed into a cloud platform.

In order to describe the required information regarding an asset, the AAS is thus made up of a series of sub-models [18, S. 112]. These sub-models represent different aspects of the assets, like security, condition monitoring, or functionality. Sub-models should be standardized throughout industry domains and enterprises, as well as the overall functionality of the AAS. An overview of possible sub-models and associated standards is given in figure 3.2.

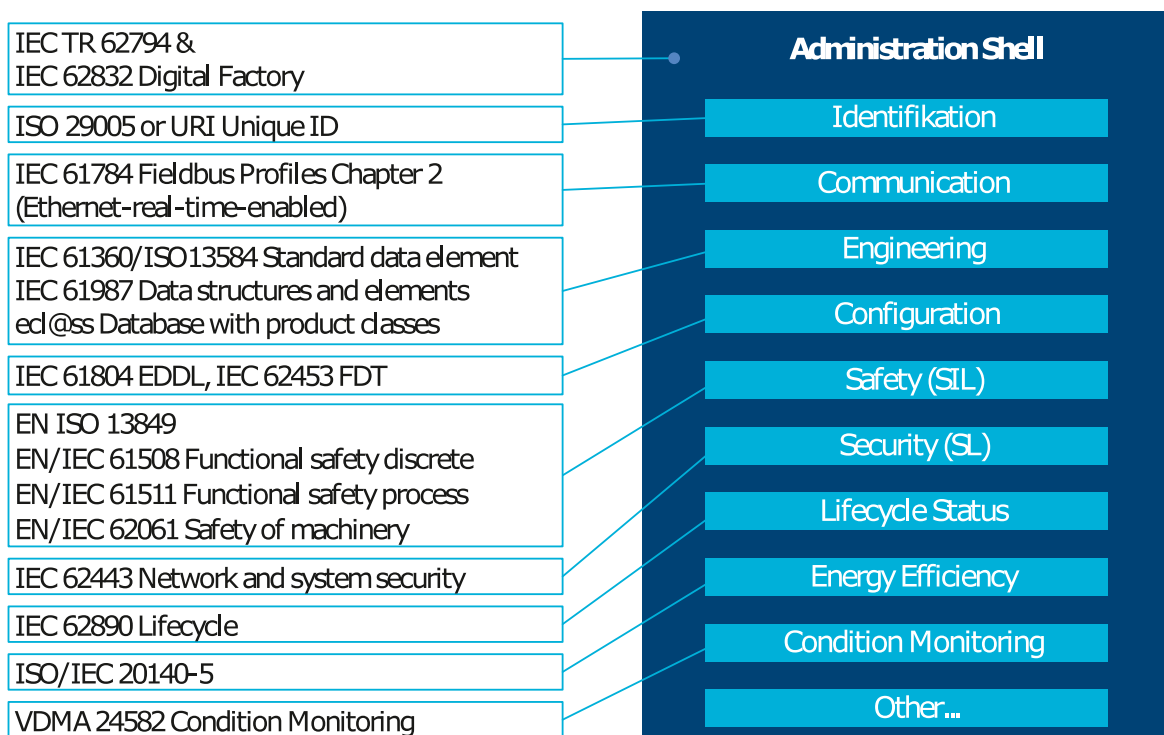


Figure 3.2.: Administration Shell sub models overview [2]

3.2. Information Modeling

An information model is a representation of concepts and the relationships, constraints, rules, and operations to specify data semantics for a chosen domain of discourse [19]. Data models are the least sophisticated way to represent a model for a particular problem. It describes different context entities as objects. Attributes add additional properties and characteristics. More complex than the data model is an information model. The difference between the data and information model is that the data model defines objects and their attributes, while an information model describes the relations between the objects in the data model

and therefore providing them a meaning. Consequently, a data model is completed by the information model. Further, on the highest level of complexity, there is knowledge. Knowledge is derived from the information model and is not only the knowledge of objects and their relations but how to utilize this information. Ultimately knowledge is the ability to reason on given information [20].

In the software domain, information models describe abstract types of objects and their relations, that enables users to utilize and interpret these types in a consistent way. To be able to create such a model, a certain degree of abstraction and structuring is required, to enable the decomposition of a problem to a more manageable size, and enable computer programs to process the described information. The decomposition of a problem usually takes place in a specific application domain. Therefore the resulting models are also called domain models. Equal named objects with similar named relations to other objects, in different domain models, can have a different meaning in one domain model than it has in another. However, widely known modeling language to create these information models are Unified Modeling Language (UML) and Systems Modeling Language (SysML).

3.2.1. Semantics

Semantic describes the meaning of signs, characters, words, and things in a given context. The semantics of a modeling language can be expressed in an informal manner or as an expression in a mathematical structure. However, in computer science, the objective is that each object or thing within a model has a meaning that is ultimately open to interpretation by machines. This enables new semantic technologies to process information more autonomously and dynamically. Semantic Web is one of these technologies, an idea of a future World Wide Web in which all information may be linked to each other, and in which machines may process all information. Today it is more or less only possible for humans to process the information on the Web and give meaning to it by interpretation since all information is embedded in natural language. Nevertheless, by enable machines to understand the content of the Web, leads for instance to more efficient search within the Web, since undesired information can be filtered out by considering the context of the search request [21].

Depending on the technology used to describe the semantics of an object within an information model, objects or things names become secondary. Instead, the meaning is given by the relations to other objects. Other objects can be classes and types of objects, but also a hierarchical relation can give an object a different meaning.

Therefore types and relations are first defined in a meta-model. Especially the meta models need to be clear and well defined so that each relationship and consequently, the meaning of an object can be described unambiguous. Models based on a given meta-model can

then be validated based on the knowledge of a meta-model to eliminate the possibility of a misinterpretation.

3.3. Product, Process and Resource concept

The PPR concept describes the relationship between products, process, and resources within a manufacturing system. An overview of the underlying relationships is shown in 3.3. Each of the parts is characterized by its feature attributes, that are then provided in the AAS. With these features, requirements for a particular product, process steps, or required resources can be derived and consequently define the relationships between these.

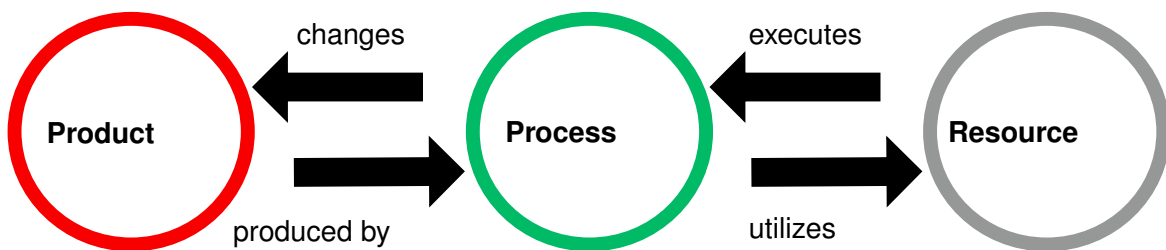


Figure 3.3.: PPR concept [10]

Utilized features should be preferably selected from the manuals or standards, like the ecl@ss catalog. In case that there is no standard to choose from yet, a standardization process should be conducted, or an industry-specific feature catalog consulted [22, S. 10]. Below the components of the PPR concept are further described.

Products Unlike may considered, products do not necessarily represent a finished product. During the production process, a starting product, like a blank steel bar, will be further processed until it reaches its final state. With each process step, it will be transformed into an intermediate product. Also, additional products, like screws and nuts, can be introduced into the process. Thru the changes throughout the process, characteristics, like geometry and other features, change and define the characteristics of the intermediate product.

Processes A process defines how a product is processed. Usually, a process consists of several process steps. It is not required that a process step changes the geometry of a product. Instead, it can be a change in position or change other features defined by the product. Sometimes several process steps can cause the required effect on a product. In this case, a consideration regarding the demanded product quality, cost and

production time has to be made. These factors belong to the Key Performance Indicators (KPIs) and can be utilized for process optimization and adjustment of customer requests. KPIs, therefore, are defined among others by features of a process.

Resources Resources represent the abilities of a process to affect a product - for instance, a drill. As a drill is a resource, it can be utilized in a drilling process, in order to drill a hole into a product. The features, defined by the resource, define how the process affects the product. In the case of the drilling process, the drill diameter would define the diameter of the hole, which is added to the product.

3.4. Skill-based Engineering

Skill-based engineering provides a fundamental approach in order to implement flexible production systems. A skill is defined as the potential of a production resource to archive an effect within a domain [4]. Ultimately skills are methods or programs on software-side or manual tasks executed by a machine operator. In a domain context, these can be typed by functionality and hierarchically ordered for each process.

Probably the most comprehensive approach to tackle this for the assembly domain has been made by the openMOS project. Here, skills are considered to be either atomic or composite. Atomic skills represent basic capabilities of resources, like the moving command of a robot or the ability of a sensor to sense. Composite Skills represent a set of atomic skills composed as one. An example is the pick and place capability of a robot. The skill is composed of a sequence of movement and grasping commands. As a result, the composite skill in the openMOS implementation has at least two skill requirements defined in order to resolve the dependencies of the required atomic skills in a composition. Skill requirements are fulfilled in skill recipes, where recipes represent the process and product skill requirements as a sequence of skills [23, S. 27]. Finally, skill requirements can then be matched against skill types. In this way, the system gains the ability to check whether it can produce a particular product or to fulfill a stipulated process step.

Even if the above described concepts are described with the terminology defined by openMOS, they are generally valid. In general, the skill model can be considered the link between the process and resources in the PPR model [7]. Skill models are usually represented in a taxonomy, where the only relation between skills is represented hierarchically. These taxonomies can be derived from an existing standard like [24] for joining and [25] for handling tasks. However, even that this concept is not new, there is still a lack of standardized skill taxonomies. Especially purely software-based skills are rarely considered.

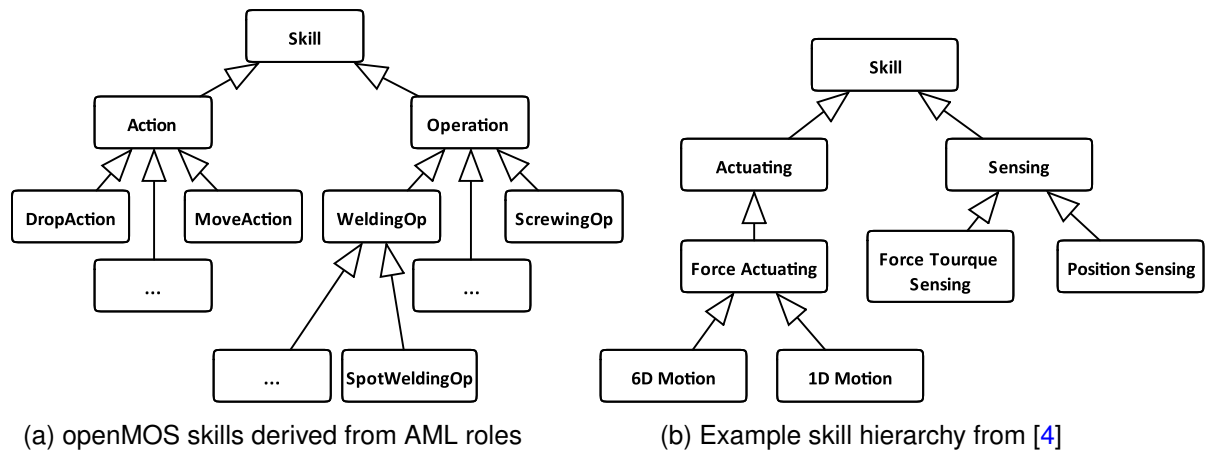


Figure 3.4.: Two excerpts in UML for example skill hierarchies

Two skill taxonomies shown in figure 3.4, represent different approaches, on how do derive a skill hierarchy. While in 3.4a), the considered skills are required to describe an assembly process, the hierarchy in 3.4b) provides a more generic way. Depending on the domain or situation, either of these descriptions could be sufficient to describe skills required in a process. Nevertheless, the more generic way of describing skills, as in 3.4b), provides more possibilities to extend it for the description of sensing and software skills.

3.5. OPC Unified Architecture

The Open Platform Communication (OPC) standards exist since 1996 and since has been used for communication in industrial automation applications. Because the OPC standard is limited to Windows platforms, the OPC Unified Architecture (OPC UA) has been developed. With the implementation of OPC UA, limitations regarding the OPC standard have been overcome. Besides, several of the OPC standards, like OPC Historical Data Access (OPC HDA), are now included within the OPC UA standard.

With OPC UA, a versatile standard for communication has been created. It offers fundamental functionality and advantages for I4.0 systems. Among others, a state of the art security model and a fault-tolerant communication protocol is included. Also, it comes with an information modeling framework that allows developers to represent data in a way that makes sense to them [26, S. 14]. Notably, the information modeling aspect enables to create system and hardware models that can easily be reused. This facilitates to describe automation systems and components with a generally valid semantic and therefore creates the possibility of uniform access to well-known components. Information presented by these systems through an OPC UA server, are contained in information models that are layered on top of

the OPC UA infrastructure. The core capabilities of this infrastructure, as well as base information models, are specified in the OPC UA Specifications IEC 62541. These Specifications are divided into 13 parts, as shown in figure 3.5.

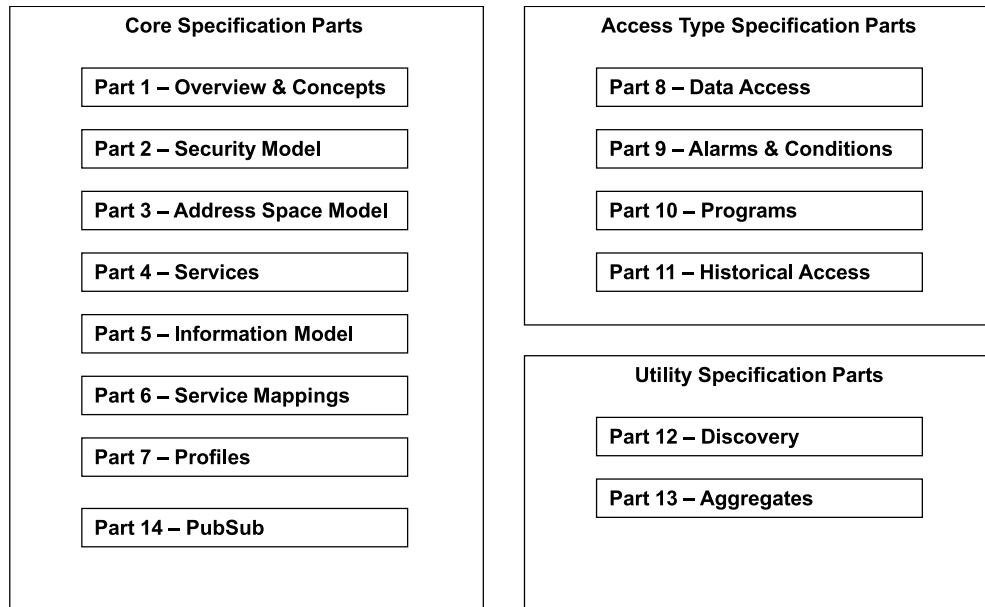


Figure 3.5.: OPC UA Specifications overview [27]

Based on core concepts, additional information models can be created and specified by a Companion Specification. The Companion Specifications addresses dedicated industry problems in order to enable interoperability at the semantic level [28]. An example is the Companion Specification for Robotics. It detains a description of common functionality and assets within a robotic system. This allows access to control functions and maintenance information with a generally valid and well known semantic.

To distribute the models created by the specification, as well as custom models, OPC UA supports to export these in an XML format. This facilitates the exchange and creation of information models since this text format can easily be edited in a simple text editor. Based on the XML representation of the information models, it is also possible to generate program code that represents the basic structure of an information model in the desired programming language. This reduces the programming effort immense. All a developer has to implement, is the servers logic, like filling OPC UA variable with values or implementing the logic of an OPC UA method.

3.5.1. Core Concepts

The OPC UA communication is based on a service orientated *Server-Client* mechanism, where a server displays information to clients in an object-orientated manner. Services provided by the server allow clients to read, write, browse, and manipulate the objects within a servers address space. Thereby a server can be accessed, by using the discovery service and resolve a server by its discovery URL or connect directly to a provided endpoint. End-points can be resolved by the endpoint URL, whereas multiple endpoints can be provided which provide communication based on different protocols, like TCP and HTTP.

In OPC UA, each object in the address space is represented as a node, which is typed according to its use and meaning, and described by its attributes and references. Attributes and references enable clients to find specific nodes within a server and to put different objects into relation. As shown in figure 3.6, the *BaseNode* has several attributes that each derived node in the address space inherits.

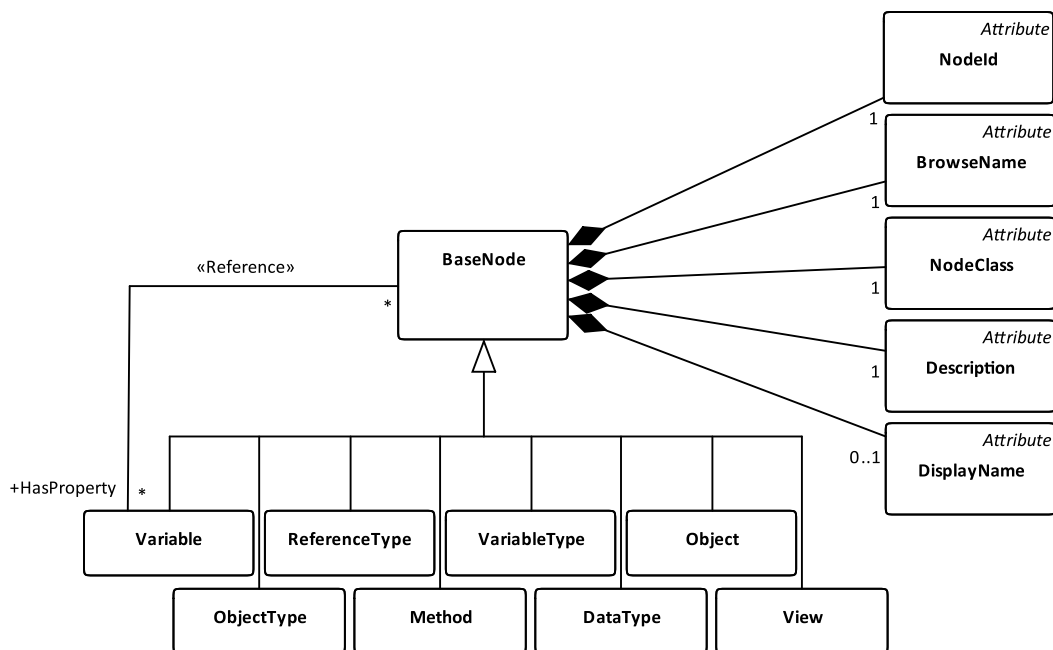


Figure 3.6.: OPC UA meta model in UML

Also shown are the base node types, fundamental node types that are derived from the *BaseNode* defining the meta-data for the OPC UA address-space. *ReferenceType*, *VariableType* and *ObjectType* define the base classes for type definitions of nodes. Custom types can be derived from this node classes. *DataType* defines a type that is associated with the value of variable nodes. The value of a variable node is included in this type of node as an additional attribute. All types known by a server are saved within the Types folder that is located in the

Root object of the servers namespace, as shown in the example object tree in figure 3.7, where custom definitions are depicted in red.

Next to the types folder is the objects folder. The server object within this folder contains the server configurations, standard methods, and other information. Each other custom defined object, representing a real-world object, will be contained in the objects folder as well. While instantiating objects and variables nodes in the objects folder, each must be associated with a type from the types folder. By typing the objects, they are ultimately provided with semantic.

Server Namespace Concept

Within a servers address-space, each node is unambiguously identified by its node ID *Attribute*. The node ID is declared as a combination of a namespace-index and an identifier, and looks like `ns=<namespaceIndex>;i=<identifier>`. Here the identifier is numeric, signified by the *i*. It can also be a string typed identifier, in which case it is stated as `s=<identifier>`. Unlike the numeric identifier, the string typed identifier has additional sub-types. For instance, the sub-type *GUID* can specify a string typed identifier in the form of a unique ID that is not just unique within a server, rather than a whole production plant or company.

The identifier identifies a node within a defined namespace, and the namespace-index defines in which namespace the node is contained. In each OPC UA server, the *BaseNodeSet* is defined by the namespace-index 0. Here all definitions stated in the core specifications are stored. Composed information models have additional namespaces where additional node types and model hierarchies are contained. Usually, each namespace represents a different model view inside the address-space information model. So, for instance, different sub-models of the AAS would be defined by different namespaces.

The namespace-index is assigned to a corresponding namespace-Uniform Resource Identifier (URI). While the numeric namespace-index is to permit more efficient transfer and processing, the namespace-URI identifies common knowledge beyond several OPC UA servers. A client, therefore, should always request the namespaces defined by a server first

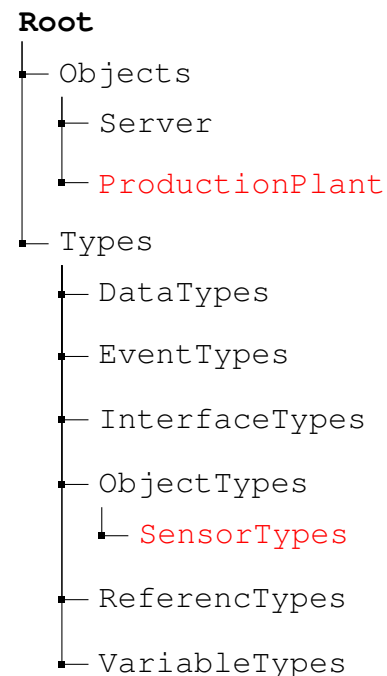


Figure 3.7.: Example server object tree

and match the required namespace-URIs in order to map the namespace-indexes. In this way, a client can ensure to access the desired information. For instance, a client needs to access the *DeviceSet* object, defined in the Companion Specification for devices. To do so, it first requests the servers namespace table and looks up the required namespace-URI defined by the Companion Specification. In table 3.1, such an example namespace table is shown, where the namespace-index for the device Specification is $ns=1$. Together with the identifier, specified for the *DeviceSet* as $i=5001$ [29] [30], the client can access the object on a server. Therefore the full node ID should be $ns=1;i=5001$ for the *DeviceSet* object. Another sever may assigned a different namespace-index to the device namespace, but thus the namespace-index was assigned by the namespace-URI required device objects can be accessed in the same way. Consequentially the namespace-index can change within the address space, while the URI ultimately identifies the namespace within explicit.

Index	URI	Description
0	http://opcfoundation.org/UA	Namespace for standard definitions of the OPC UA specifications
1	http://opcfoundation.org/UA/DI	Namespace with the definitions from the OPC UA Device Companion Specification
2	http://opcfoundation.org/UA/Robotics	Namespace with the definitions from the OPC UA Robotics Companion Specification
3	http://example.ns/SensorTypes	Namespace with custom sensor type definitions
4	http://example.ns/ProductionPlant	Example namespace, where definitions from the namespaces above are utilized

Table 3.1.: Namespace example

Browsing

Due to the fact that the node ID can have a string typed identifier, it is tempting to choose the nodes name as an identifier. But since the identifier should be conclusive within a given namespace, is it not practicable. Considering the namespace table from 3.1, if there where several sensors, monitoring environment data inside the production plant, for instance, temperature and humidity, the environment sensor type would be defined in the *SensorTypes* namespace, as shown in figure 3.8a. Using this definition and instantiate two environment sensors within the production plant would lead to an object tree, as shown in figure 3.8b.

If the node identifier is chosen the same as the names in the tree, a client accessing the data could not distinguish between the humidity nor the temperature sensor in environment sensor one and two.

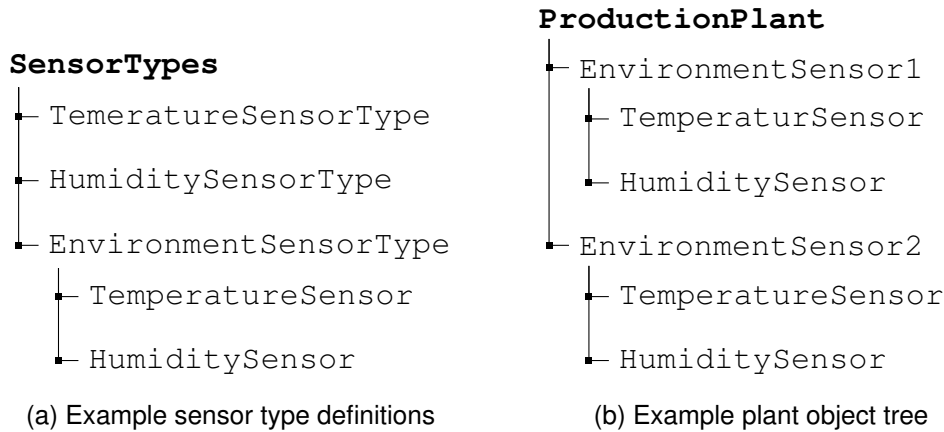


Figure 3.8.: Example object tree with type definitions and instances

So instead of using an objects name as an identifier, each *Node* has an additional *BrowseName Attribute*, with which the objects name is specified. The *BrowseName* can be used to navigate within a servers-address space. Other than the node ID it is not unambiguous. A client, looking for temperature sensors in the production plant, not given any node IDs, would use the servers browse service to browse the address-space for the temperature sensors. Thereby the given starting node for the browse request would be the production plant node. If no further restrictions are made, the request will return all nodes within the production plant, and ultimately, these can be filtered to find the two temperature sensors.

The browse service is specified in [31]. Next to the browse service, an additional service is specified, that allows to translates a given browse path to a node ID. So instead of browsing the whole namespace for a given *BrowseName*, this service enables to find the unique ID of a node by passing it its browse path. For the temperature sensor within the environment sensor two, this path would look like `ProductionPlant/EnvironmentSensor2/TemperatureSensor`.

References

A browse service request can further be constrained, so it will return only the nodes with a given *NodeClass* or following only nodes with a given reference. References relate nodes with each other, and same as *Attributes*, they are defined as fundamental components of nodes. They are either hierarchical or non-hierarchical, where hierarchical references are used to create the structure of objects and variables, while non-hierarchical are used to

create arbitrary associations. The node containing a reference is referred to as source node, and the node the reference is pointing referred to as the target node. It is not mandatory that the target node is within the same address space as the source node, it can also be located in the address space of a different OPC UA server.

Moreover, references can be symmetric or not, as well as inverse or not. If a reference symmetric, this means that the reference has the same meaning when observed from the source to target and vice versa. Otherwise it could be given a new meaning by its *InverseName* attribute.

Considering the example from figure 3.8, the *SensorTypes* object has three hierarchical references of type *HasSubtype*, since it specifies three sensor types derived from the parent sensor type object. On the other hand, the environment sensor type has two components, a temperature sensor, and a humidity sensor, and therefore, two hierarchical references of type *HasComponent*. Besides, the *ProductionPlant* object includes two *EnvironmentSensors*, its typed defined as a sub-type of the *SensorTypes* object. Thus the two *EnvironmentSensors* within the *ProductionPlant* object have a non-hierarchical reference each of type *HasTypeDefinition*.

MonitoredItems and Subscriptions

MonitoredItems are entities in the server that usually clients create by invoking the *CreateMonitoredItems* service. These items are configured to generate a notification that is ultimately transferred back to clients that subscribed to these items. Notifications are generated when a data change, event, or alarm occurred. The item to be monitored may be any node attribute. However, the server can also use local *MonitoredItems* to monitor events that are then handled by some internal server logic. In that case, the notification is not managed by *Subscription*. Instead, a callback is attached to the notification generated by a *MonitoredItem*.

A client can create a *Subscription* to any *MonitoredItem*. With *Subscriptions* clients are enabled to monitor specific values, without polling it. They are independent of the actual communication connection and, therefore, independent of a session created between the server and client. A *Subscription* can be configured to value changes of *Variables* or *Events* that are triggered by a server. To each *Subscription* a callback will be assigned. Notification messages and transmitted values can then be utilized within the callback.

Ultimately, each *MonitoredItem* is used to generate notifications, while *Subscriptions* are used to report notifications back. Every *MonitoredItem* is attached to exactly one *Subscription*, and a *Subscription* can contain many *MonitoredItems*.

Events and Alarms

Events represent specific occurrences within the server. *Subscriptions*, for instance, can be attached to a value change *Event* of *Variable*. Nevertheless, a server can create an *Event* to inform about any condition that may occur within a system. Clients can subscribe to the events to be notified if they occur.

Alarms are a special kind of *Events*, with conditions attached. A condition could be, for instance, a temperature exceeding a configured limit. Depending on the configuration of *Alarms*, the handling of an *Alarm* occurrence may differ, if the *Alarm* is set acknowledgeable. In that case, the server can be configured to take a particular state until the *Alarms* cause was handled and acknowledged by a client.

Interfaces

Interfaces are defined in an amendment of the OPC UA specification part five. It represents a generic functionality, usable by different object types or objects [29]. Interfaces are defined by modeling an interface description as a type with which a specific object structure is described. Object supporting a particular interface, define a *HasInterface* reference, pointing to the defined interface type. This genuinely means that the object with the *HasInterface* reference defines the same structure as in the interface type definition.

3.5.2. Publish Subscriber Mechanism

The Publish-Subscriber (PubSub) mechanism is an additional expansion to the service oriented Server-Client mechanism of OPC UA. PubSub communication is loosely coupled, meaning that the publisher that is providing data does not need to know about the subscribers listening to messages [32]. The number of subscribers is not affecting the performance of the publishing server. This makes it versatile for monitoring and logging data where high data rates are required. Strictly speaking, it is recommended by the OPC Foundation to use the PubSub for data rates higher than 100Hz [33].

PubSub as to OPC UA is designed to be flexible and is not bound to a particular messaging system. In the specification, three PubSub transport protocols specified that can be used alongside the Server-Client communication. Message Queuing Telemetry Transport (MQTT) and Advanced Message Queuing Protocol (AMQP) are standard messaging protocols supported. Both require an additional middleware. This middleware is also called a Broker. However, that means, next to the publishing OPC UA server, an additional server, for implementing the protocols middleware, is required. Subscribers are then enabled to connect

to this middleware and receive data from the publisher server. As a result, the messages will take a detour over the middleware before arriving by the subscribers. It is not required that the subscribers support the use of OPC UA instead utilize existing implementation of the particular protocols.

Next to the standard messaging protocols, OPC UA supports User Datagram Protocol (UDP) for publishing data. It is a core member of the internet protocol suite and can be referred to as a connectionless communication. Unlike with the Server-Client mechanism or the middleware based PubSub protocols, this protocol does not need to exchange any information with communication partners to configure the communication. In the OPC UA, PubSub communication messages send with UDP will be binary encoded. As a result, the receiver only needs to know the encoding schema. This makes it the most efficient way to transfer data with high data rates.

Which data will be published is defined by a *DataSet* object within the namespace of the server. Here already existent variables from the servers namespace can be referenced. Publishing rate and additional parameters of the PubSub mechanisms are specified by the PubSub configuration model in [34].

3.5.3. Programs

Methods defined in OPC UA enable to execute simple functions, like setting a counter or perform a simple calculation. These are stateless and limited to an execution time of ten seconds. Within this time, a *Method* should return a result. OPC UA *Programs* are utilized for more complex, stateful tasks than *Methods*, like the execution of a machine tool program or manage a file transfer. A *ProgramFiniteStateMachine* is a subtype of the *FiniteStateMachineType*, defined in the OPC UA Specifications Part 5 [27, Annex B]. The difference between OPC UA programs and state-machines is that a *Program* has a standard set of base states, transitions, and methods predefined. Sub-types of the *ProgramFiniteStateMachine* can expand this structure and add additional methods, transitions and states. Program methods cause the state to change by triggering the predefined transitions. Consequently, a client is enabled to call a servers program method and thus triggering a state transition. The cause of a state transition can also be an internal *Server* event. Transitions, on the other hand, can trigger OPC UA events, to that clients can subscribe. If the program finished its execution, a *FinalResultData* object within the program object preserves possible result data [27].

3.5.4. Notations

Because the visualization of address space models in common markup languages like UML or SysML is somewhat confusing, the OPC Foundation introduced a dedicated notation for OPC UA. With these notations, it is more convenient to display the structure and relations within an information model. An overview of these notations is given in figure 3.9.

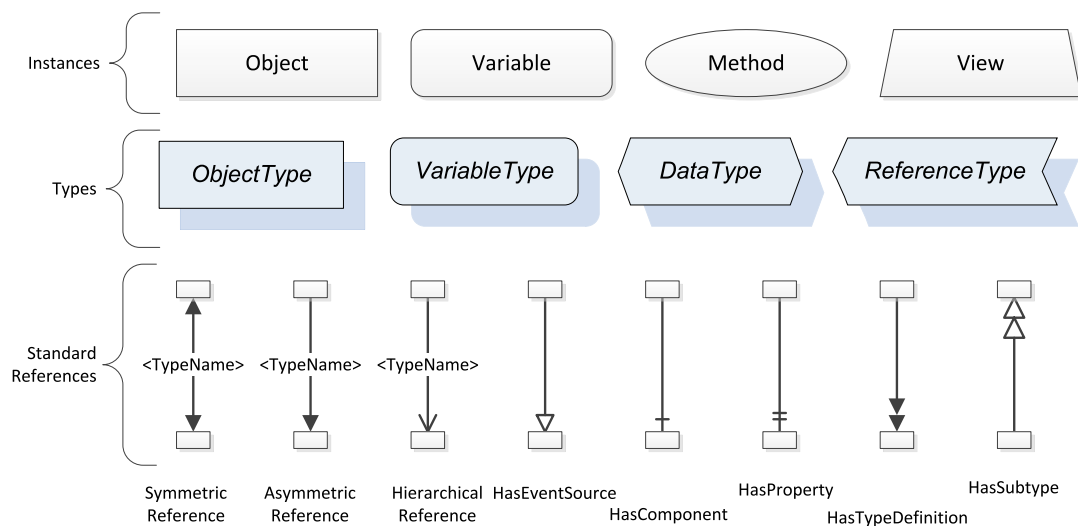
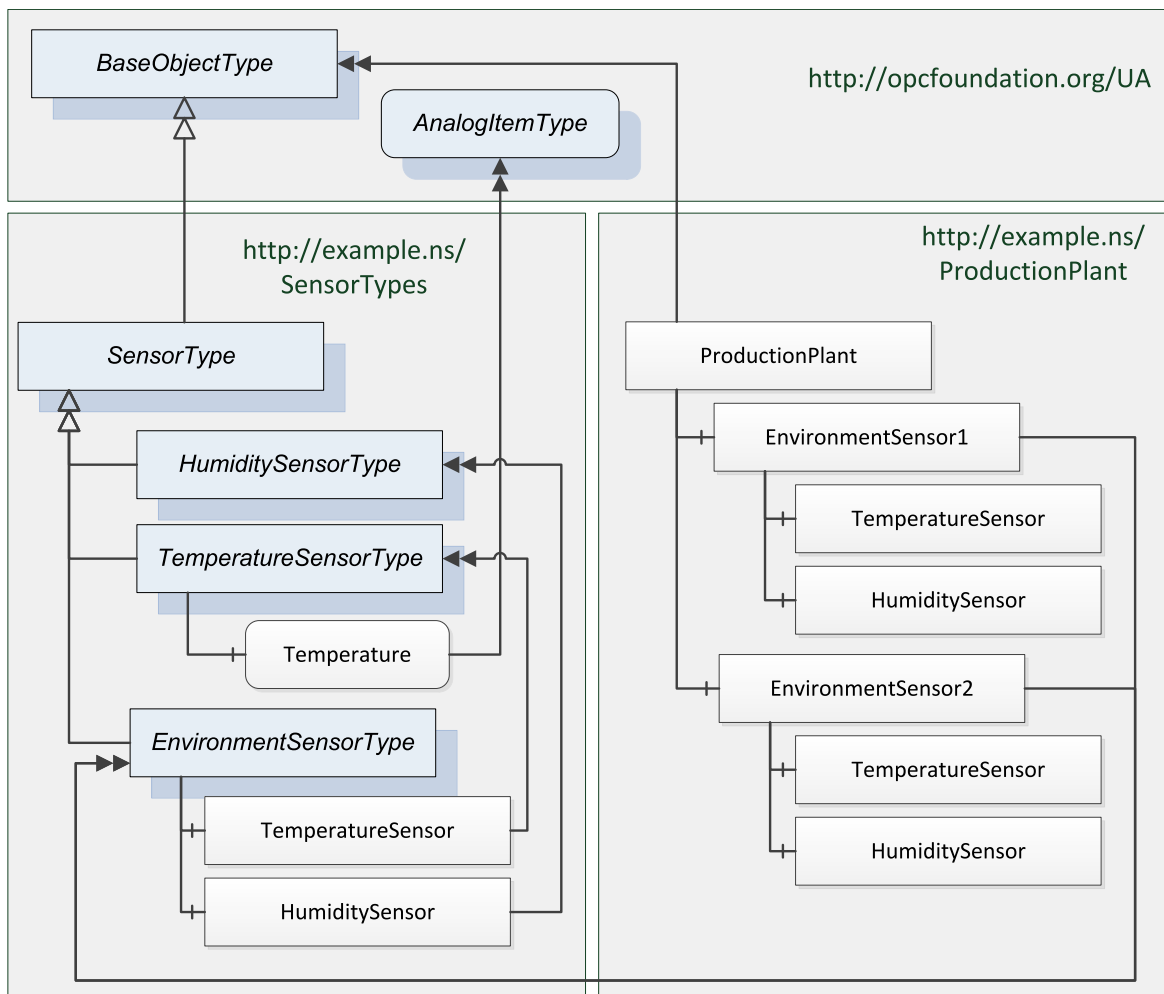


Figure 3.9.: OPC UA notation overview

Namespaces in these kinds of diagrams can be depicted by additional borders around the nodes related to a specific namespace. The namespace-index or -URI can be depicted somewhere within the border. To visualize complex server address-spaces could become confusing, especially if many namespaces are present. However, to represent model excerpts and ideas, it is quite suitable.

Considering the example from 3.5.1, the example model is visualized with the OPC UA notations, as shown in figure 3.10. Also, the type definition of the *TemperatureSensorType* has a variable defined. The variable typed is defined as and *AnalogItemType*.

Figure 3.10.: Example *Namespace* in OPC UA notation

3.6. AutomationML

Automation Modelling Language (AML) has been developed as a neutral data exchange format based on XML to gather and exchange relevant plant and production data focusing on the domain of automation engineering. It is standardized in the IEC 62714. The goal is to interconnect the heterogeneous tool landscape of modern engineering tools in different disciplines. Therefore it describes plant and system components in an object oriented manner, encapsulating different aspects. Typical aspects in plant automation comprise information on topology, geometry, kinematics, communication, and logic [35]. Resulting documents from different engineering tools can further be referenced within an object and enhanced with additional data, e.g., for configuration and documentation. Regarding the RAMI, it can be assigned to the information layer.

An AML file can be created in the AML Editor, shown in figure 3.12, where the five main components of each AML file are depicted. The AML roles in the *RoleClassLibrary* (1), interfaces in the *InterfaceClassLibrary* (2), standard system units in the *SystemUnitClassLib* (3), and the *InstanceHierarchy* (4). These will be further described below. Also, there are attributes (5), which define additional properties of objects that are not directly visible in an AML object tree. The arrows indicate the overall workflow. Primary roles and interfaces are defined or included, whereby several libraries are defined by the AML specifications and best practice recommendations. After all required roles and interfaces are available, standard system unit classes can be defined. To each class, a role can be assigned, or interfaces can be added. Finally, an instance hierarchy can be created by instantiating the predefined system unit classes. Instances are referred to as internal elements because classes are instantiated within an instance hierarchy or within other classes.

Different classes and objects can be related with each other within the models. Depending on the usage, these have different meanings and restrictions. Parent-child-relations between AML object instances are used to represent hierarchical object structures. These are primarily found in the instance hierarchy. Parent-child-relations between AML classes is when different system unit classes are encapsulated. It is no more than a hierarchical neighborhood without further semantic. However, in the AML editor this relation would look like similar, if only the arrangement of classes id considered because inheritance relations do not look different, except for the parent class definition, which can be turned of in the AML editor. Figure 3.11 shows how these relations would look like in the *SystemUnitClassLib* tab of the editor. In the *ParentChild* library folder the different parent child relations are shown, while the *inheritance* folder does only contain inheritance relations. However, it is also possible to relate instance-instance relations through the usage of interfaces and internal links.

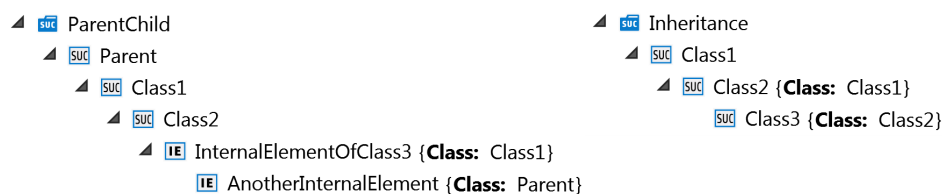


Figure 3.11.: Parent-child and inheritance relations between classes and objects.

Usually, different model views are contained in different AML files, similar to the namespaces in OPC UA. These files can be combined in so called AML containers, described in a best practice recommendation [36]. This enables to exchange the gathered information, comprising of AML and other files from different engineering tools.

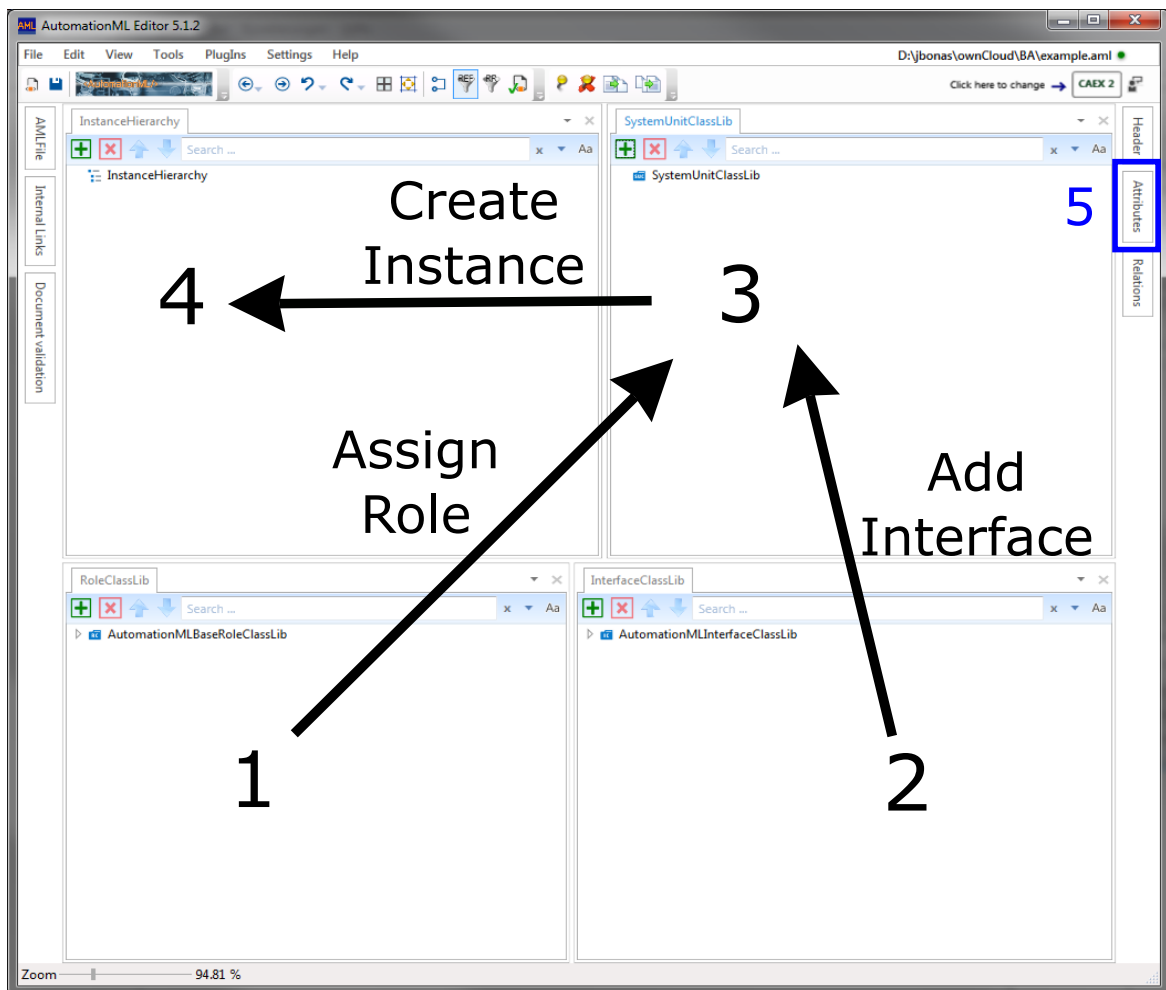


Figure 3.12.: AML Editor main components and workflow

3.6.1. Instance Hierarchies

In the Instance Hierarchy, the actual system that is to be described is represented. It is not required to define the system unit classes first and instantiate these in the Instance Hierarchy. Instead, an instance can be created and edited directly. However, the best practice is to first define reusable component classes inside a *SystemUnitClassLib* before instantiating these in the Instance Hierarchy.

3.6.2. System Unit Classes

Within the *SystemUnitClassLib*, custom objects can be created. Unlike interfaces or roles, AML does not predefined any system unit classes. System unit classes represent reusable resources, that can be combined in the instance hierarchy to build up a model representation of a production system.

3.6.3. Roles

A role in AML is a class that describes an abstract functionality without defining the underlying technical implementation. By associating an AML object with a role, the object gets a semantic [35]. An object can also be associated with several roles or define supported roles, depending on its use and the model view. For instance, a robot controller in an AML file describing the system topology could be associated with the *Controller* role from the *AutomationMLCSRoleClassLib* for control equipment. In a different model view, as an example, describing the communication of a system, this controller may require a role from the AML *CommunicationRoleClassLib*, defined by the communication specification. Therefore it will be associated with the *LogicalDevice* role.

Roles are defined in the *RoleClassLibrary*. AML already defines different role classes, that can be utilized within a project. If custom roles are created, these have to be derived from roles within the *AutomationMLBaseRoleClassLib*.

3.6.4. Interfaces

Interfaces enable to relate instances within the Instance Hierarchy with each other or reference external documents. An instance-instance relation, created by connecting two Interfaces, is referred to as an internal link. Relations to external documents, however, are referred to as external links. In the AML standard interfaces are defined in the *AutomationMLInterfaceClassLib*. Same as AML roles, custom-defined interfaces have to be derived from these standard types.

Attributes can be added to the interfaces and define additional configurations. For the *Communication* interface, defined as a base interface, these parameters can be used to configure a communication, like a port or IP for Ethernet communication.

3.6.5. Communication Modeling

The AML whitepaper for communication [37] specifies different aspects for the modeling of communication networks, e.g., communication topology and configuration. Depending on the use-case, is it possible to depict hardware as well as logical communication within an AML model.

3.6.6. Data Variables

In the best practice recommendation for *DataVariables*, AML defines a concept for accessing concrete data elements inside communication networks. These networks are described by logical devices responsible for communication, e.g., OPC UA servers [38]. A *DataVariable* would describe a variable by means of the underlying communication technique. For instance, a *DataVariable* described by means of OPC UA is unambiguously identified by its node ID. The *DataSource* role, also defined in the best practice recommendation for *DataVariables*, would, therefore, define an object in AML as a logical device. For OPC UA, this device is identified by the servers discovery or endpoint URL. An additional *DataVariable* attribute would define the node ID. With the information of server URL and node ID, it is possible to access the requested information. In combination with the conversion of AML models to OPC UA servers, described below, it is possible to create an aggregated server that subscribes to desired *DataVariables* on other servers in the network and shows the live value in its namespace.

3.7. AutomationML and OPC UA

Since AML as well as OPC UA models have a XML representation, it is possible to convert AML models to OPC UA information models. The rules for this conversion are specified in DIN SPEC 16592 and also available in the OPC UA and AML specifications. This enables the possibility to model all required information in AML and transfer it to OPC UA for communication. An OPC UA object tree, resulting from the conversion AML to OPC UA is shown in figure 3.13, where the OPC UA default nodes are depicted in black, standard nodes specified by the conversion rules in blue and custom defined objects in red. Basically, AML objects and instances, as well as Interfaces, will become OPC UA objects and attributes will become OPC UA properties. If an instance has a class type definition, the resulting object will have a *HasTypeDefinition* reference pointing to the associated system unit class in the *SystemUnitClassLibs* folder in the OPC UA namespace. Roles will be depicted with a *HasAMLRoleRequirement* reference, pointing to the associated role in the *RoleClassLibs* folder.

This conversion promises to create OPC UA servers, based on may already existing information in AML and operationalise these. This could facilitate the creation of OPC UA servers for devices and systems. One other opportunity is the loss less exchange of OPC UA system configuration. By describing a communication system according to the AML specification, the systems configuration to configure servers and clients can be embedded. Making it available as a OPC UA server within the communication system, enables to configure all the communication participants based on the information, modeled in AML. This allows a automatic configuration of servers and clients in a network by accessing the configuration server.

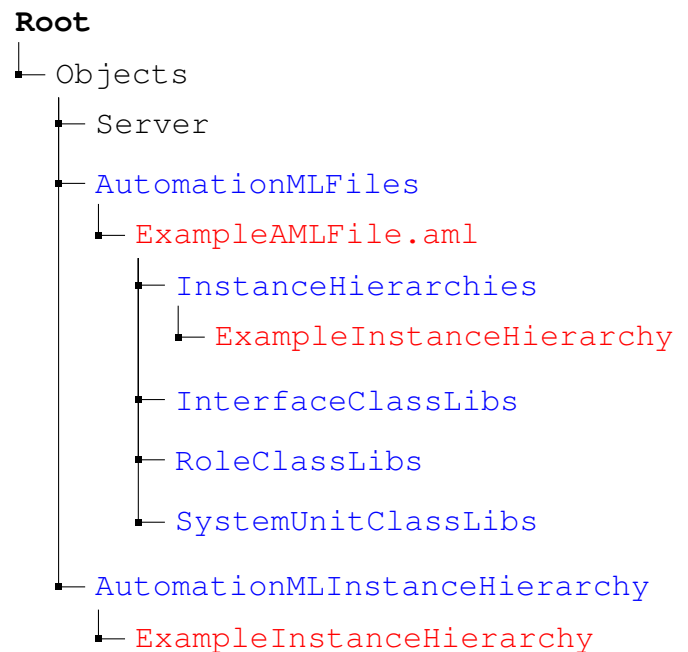


Figure 3.13.: Example OPC UA server object tree generated from AML

However, the resulting models will be compliant to the rules of the specification in DIN SPEC 16592 but not to the modeling rules by the Specifications of OPC UA. This means, that the structure of the server, as well as the types used to describe things are not unified. An entity relying solely on the OPC UA specification to access devices in a communication network, would fail, if the device server was created based on a AML model.

3.8. Machine Learning

Machine Learning (ML) is the scientific study of algorithms and statistical models that computer systems use to perform a specific task without using explicit instructions, relying on patterns and inference instead [39]. To rely on these patterns, an ML algorithm needs input data to learn these patterns from. The output generated based on the data could be a classification, regression, or even new data, like image or audio data. Classification problems are often implemented for pattern and object recognition in computer vision applications and natural language processing. Regression techniques are sometimes used in control engineering, to approximate plant functions. The overall goal is to make a prediction with available input data and therefore predict a class, number, state, word, or action.

In general, it is distinguished between three types of machine learning. Supervised learning, unsupervised learning, and reinforcement learning. Supervised learning relies on prepro-

cessed data, meaning segmented and labeled, to provide the algorithm during training with input and the desired output data. During training, the selected algorithm will adapt its internal mathematical representation, e.g., weighting factors, depending on the outcome of a prediction, to ultimately reduce the error rate on its predictions. Unsupervised learning applications, however, are used to cluster disordered data, and find noticeable, but otherwise not obvious, patterns in usually big data set. This enables to utilize unsupervised learning for the perpetration of data for supervised learning applications. By using clustering techniques it is possible to conduct an automated segmentation of the data by segmenting each data point depending on the related cluster, the data is in. Reinforcement learning techniques are concerned with how software agents ought to take action depending on its environment or current state. Therefore they are using cost and reward functions to score their actions. Depending on the score, the algorithm is en- or discouraged to repeat the action, if similar states appear again on the input.

In the context of I4.0, ML is often mentioned in context of process optimization and machine maintenance prediction [40]. These tasks require big amounts of data and rely on different supervised and reinforcement learning techniques. Because of the significant amount of data these models have to process, they are usually deployed inside a cloud system as agents or services. However, other fields of application regarding these algorithms, is to improve the cognition of robots and machines. Therefore they have to be deployed onto, e.g. a controller inside the production plant. In each case, to further improve the algorithms, new data is required, that has to be collected and segmented during operation. With each new model training, the model improvement is evaluated. Depending on the outcome, the new models can be deployed.

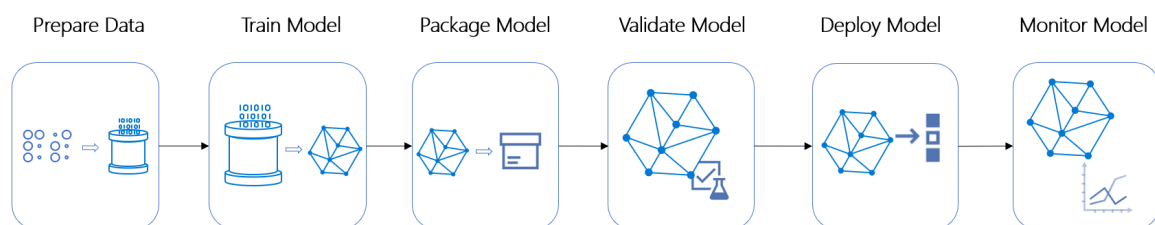


Figure 3.14.: ML pipeline by Microsoft Azure [41]

The process of gathering data, preparing data, model training, evaluation, and deployment is commonly referred to as ML pipeline. A proposed pipeline from the Microsoft Azure ML tool is depicted in figure 3.14. Pipeline does not mean, the work-flow is always one-way, from start to beginning, rather it is an iterative process, where new data is generated, the model trained and eventually its parameters adapted before it is trained again and finally deployed. Depending on the application, this pipeline sequence can variate, but all steps have to be available when deploying a model into production [42].

4. System and Concept

To provide a better understanding of the implementation goal, the current system, and application utilized, is described. Further, based on these descriptions, the problem is described, and the use case application set into context. Besides, requirements and an objective for further implementation are derived.

4.1. Task Description

The program, utilized to illustrate the integration, is part of a system, that task is to screw so-called Hi-Lok Collars, in order to perform the assembly on a circumferential joint of two aircraft fuselage parts. Different fuselage sections are shown in figure 4.1. Each section is connected at the circumferential joint. At each joint, two to three rows of Hi-Loks have to be placed.

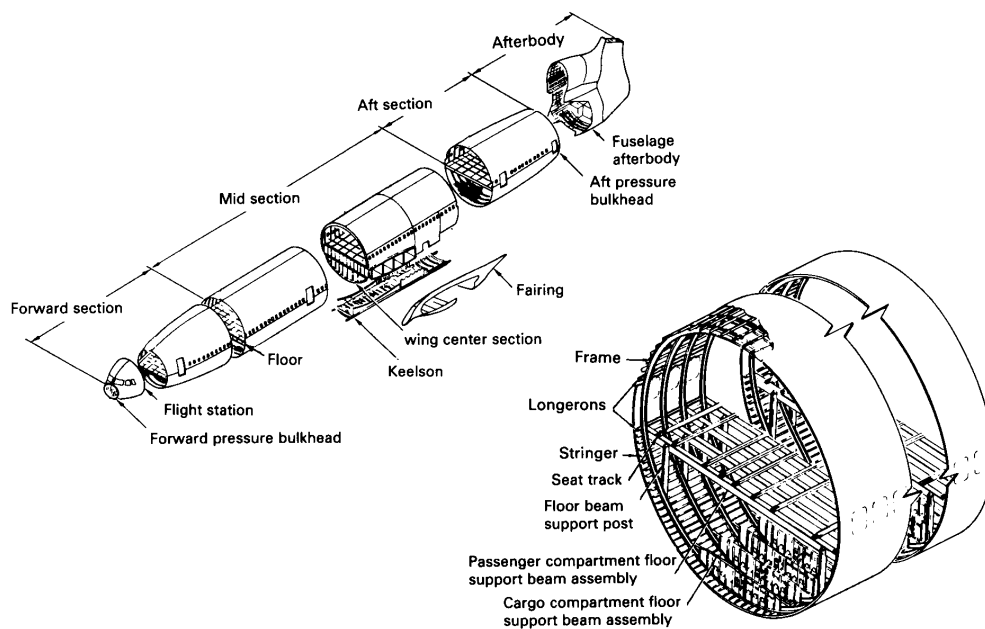


Figure 4.1.: Structural design of aircraft fuselages [43]

Hi-Lok collars are part of the rivet connection, connecting the fuselage parts. Collars have an inner thread and a nut that wrenches off after assembly. The counterpart of these connection elements is the collar pin. Collar pins have an outer thread and are placed through a hole in the fuselages that are to be joined. After the pins have been placed, Hi-Lock collars are screwed onto the pins in order to fulfill the connection. The riveting process of one rivet is considered successful if the predetermined breaking point of the collar breaks (see Figure 4.2). As a result, the collar wrenching is a remnant of the Hi-Lok that is not further needed. Through the deformation of the collar during the screwing process, it results in a positive-fit connection.

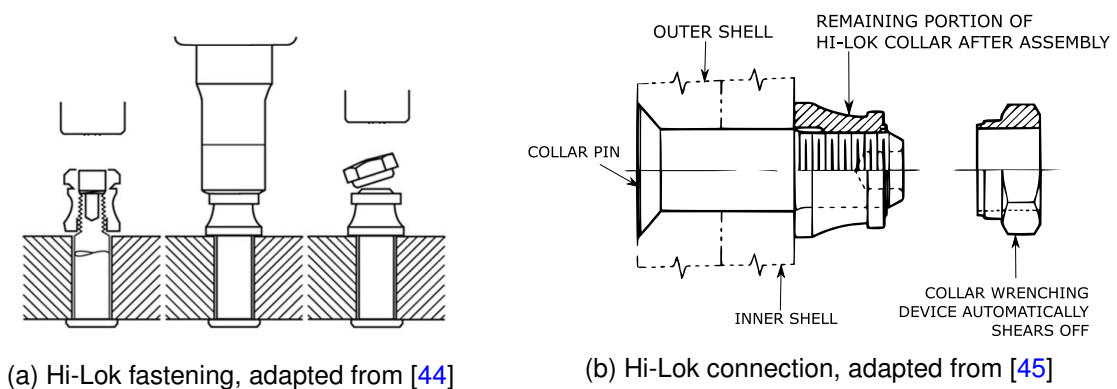


Figure 4.2.: Hi-Lok connection principle

4.2. Riveting System Description

The process of placing the pins has been automated and is described in detail in [44]. It uses non-collaborative robots that are stationed at the outside, around the fuselages on two floors, and additional linear movement systems (see figure 4.3). Equipped with an appropriate end-effector, these robots drill, place, and seal the rivets around the circumferential joint. During the drilling and pin placing operation, nobody is allowed to be inside the fuselages.

The system, providing the use case for this work, is another robotic system. Its task is to screw the collars onto the pin from the inside of the fuselages. The system is part of a research project promoted by the German Federal Ministry for Economic Affairs and Energy and is conducted by the Fraunhofer IFAM and several other participants. The project is called MFlex and a goal is to research how such a system can be made modular so that each component becomes easily exchangeable. For the riveting system, this means each device and each program is considered to be a module that can be exchanged by a module with the same type. For instance, if a modular system contains a collaborative robot, it should

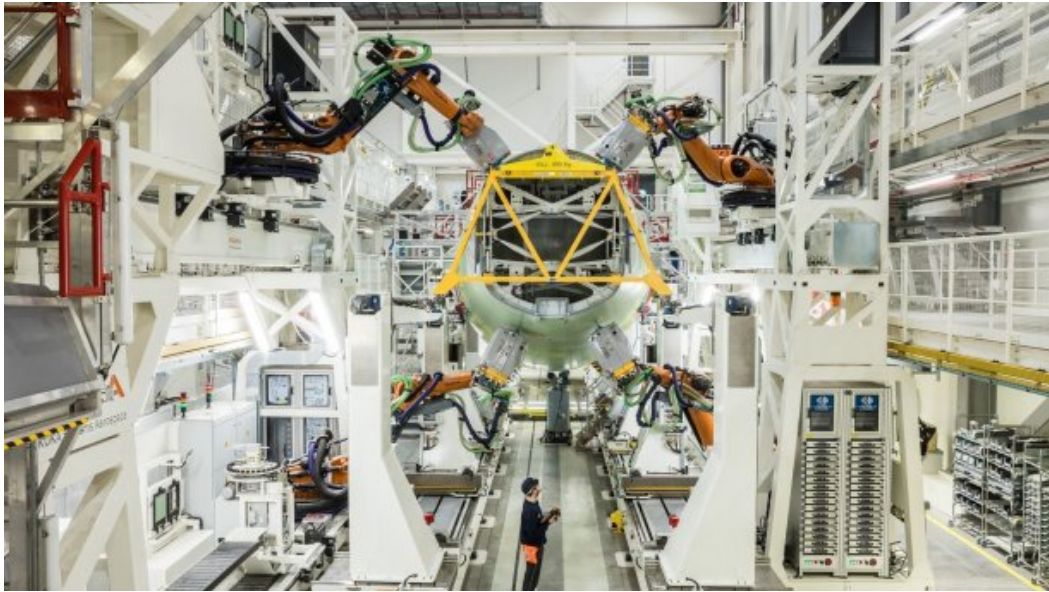


Figure 4.3.: Robotic system placing the Hi-Lok pins [46]

become exchangeable by any other thing that is typed as a collaborative robot. In the future, this should make automation systems more flexible and components in these systems better reusable. This is essentially the idea that supported by the idea of an I4.0 and P&P systems.

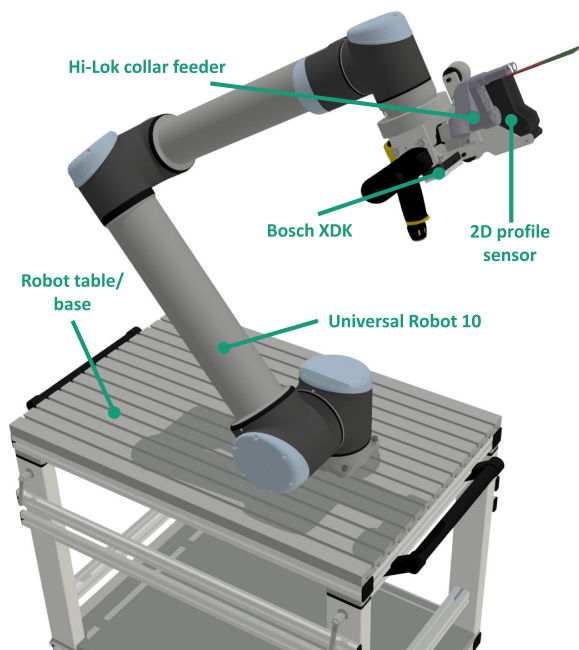
The collar screwing automation system is composed of several hardware modules that are described briefly. An Universal Robot 10 (UR10) that is placed on top of a wheeled table so that an operative can move it to its working area. A pneumatic Hi-Lok collar feeder tool is attached to the end-effector. This tool is, in fact, a handhold manual tool used by the operatives within the production. With this collar feeder, the robot can screw the collars onto the pins. The tool is controlled by switching on and off the electromagnetic pneumatic valve, the collar feeder is connected to. The collars are stored in a tube, from where they are fed by compressed air to the collar feeder. Next to the feeder, a 2D profile or laser line sensor is mounted, to scan the circumferential joint and calculate the pin poses from the resulting data. Also, a Bosch XDK, an embedded microcontroller board, is mounted at the bottom of the end-effector. The process is controlled by an Industrial PC (IPC) that is placed below the table. Robot, profile sensor and valves are connected via Ethernet to the controller. The Bosch XDK is connected via USB. The whole system is shown in figure 4.4.

The procedure of screwing the collars can be divided into five steps. First, the system is moved by an operative to its working position where compressed air and power supply are connected. Second, the robot is moved to an initial pose from where it can start its scanning task, third. The scan will take place in a section between two Stringers (see figure 4.1). After

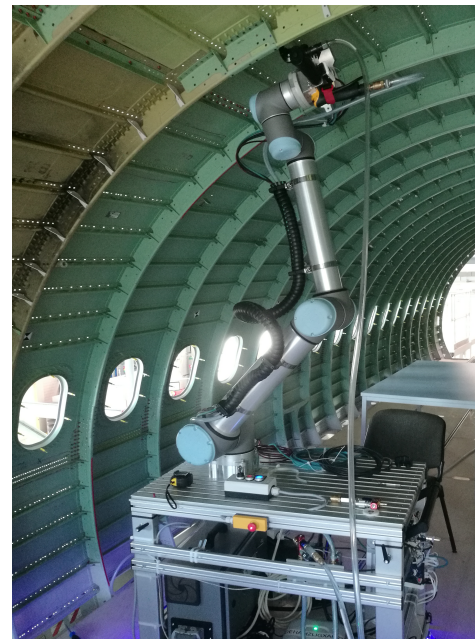
the scanning is done and the pin poses are calculated from the scanning data, the robot will start with screwing the collars by approaching the points calculated before, fourth. Because the pneumatic cannot sense the break of the collar, a predetermined time has been set that the robot will take before moving to the next pin. Fifth, the robot will move to the next section and repeating the scanning and screwing process. The fifth step will repeat until the robot is unable to reach another section.

The predetermined time for screwing the collar is not appropriate for screwing several hundred collars, because the time needed would increase massively if the collar break occurs with high variance within this time window. One way to tackle this problem would be to use an electric feeder that can sense the falling turning moment when the collar breaks. But since there was none available during the phase of development, another solution was proposed, by utilizing the acceleration sensors that are contained in the robot and the Bosch XDK.

By evaluating the acceleration data logged while screwing the collars, it was apparent that it is not possible to simply use a threshold on the data. This is why a machine learning application was developed in order to detect the break of the collar. That application is a pure software application and the central concern of this work.



(a) Riveting system overview



(b) Riveting system inside the fuselages

Figure 4.4.: Riveting system

4.3. Current Implementation

The machine learning application for the detection of the collar break, has already been implemented and tested. Its underlying algorithm and the scenarios it has been tested in, should be briefly elucidated here.

4.3.1. Break Detection Algorithm

For the implementation of the machine learning application, a statistical method, the so-called Random Forest Classifier (RFC), has been utilized. The RFC is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the data set and uses averaging to improve the predictive accuracy and control over-fitting [47]. Generally, it can be assigned to the supervised learning methods. So that the RFC can detect the break of a collar, it is fed with data from an acceleration sensor to train it. The sensor is attached close to the Tool Center Point (TCP) of the robot. It is configured to stream the acceleration with a rate of 125 Hz. That is then aggregated throughout 50 samples, in order to form a window over the acceleration signal. The acceleration window then serves as input for the classifier. As an additional input for the classifier the time since the feeder start signal occurred is utilized. It is assumed that the feeder is started at the same time the break detection is invoked. With the acceleration window and the feeding time, the resulting input dimension of the classifier is 51 data points. Furthermore, the acceleration data is normalized between zero and one. On the output, the classifier provides a binary signal that classifies whether the break event is occurring or not.

Because of the high sampling rate and the low input dimension, the output from the classifier is further processed. This is done within a state machine, checking the output of the classifier and generating a break signal, as soon as the event occurrence happened over a specified period. Otherwise, if the predefined time for screwing the collar exceeds, the state machine will reset.

4.3.2. Possible Integration Scenarios

Figure 4.5 and 4.6 show two possible configurations for the integration of the break detection application into the process. It is a simplified view in a UML component diagram. Each module is displayed as a component. Components that are stereotyped with *SoftwareModule* are programs that run independently on the *RivetProcessController*. The controller is displayed as a UML node. Devices are stereotyped with *HardwareModule*. Each module encapsulates the functionality of the module and provides it by an interface. These are either depicted by

the “lollipop” symbol or as a “flow” dependency. The “lollipop” represent complex interfaces that enable method calls, subscriptions, and others, while the “flow” dependency describes a data stream without further interaction. This modeling approach is adapted for Service Oriented Robotics Architecture (SORA) and is described in detail in [48].

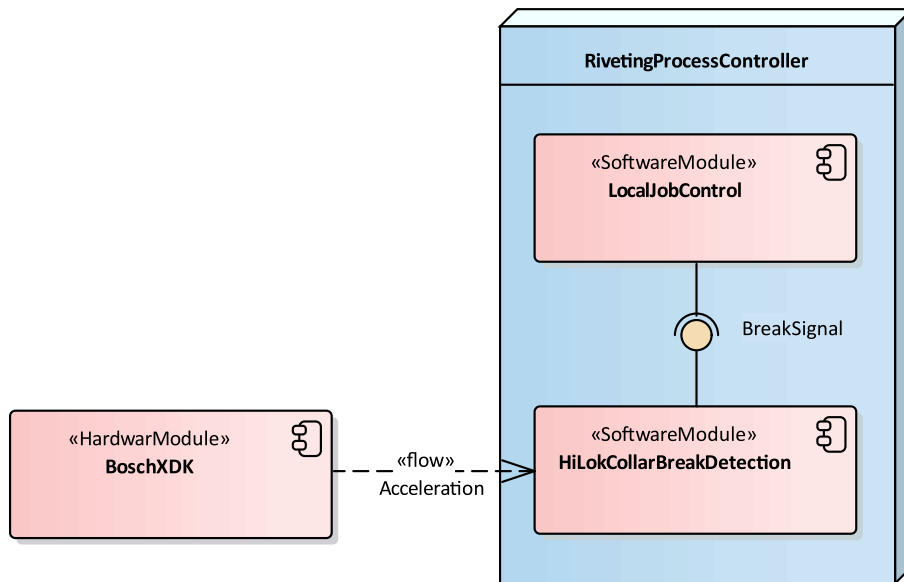


Figure 4.5.: Break detection configuration with Bosch XDK

In the first configuration, the Bosch XDK is utilized as a source for the acceleration signal. It is connected via USB to the process controller, and the application reads the signal directly from the appropriate serial port. The second configuration utilizes an acceleration sensor integrated into the UR10. This sensor can be accessed by utilizing one of the provided interfaces of the UR10. In this case, the application is utilizing the UR10s Real-Time Data Exchange (RTDE) interface.

Components depicted in the diagrams are described in the following:

LocalJobControl controls the procedure of the riveting process. Its task is to move the robot to the rivets and start the collar feeder. It also invokes the collar break detection. The break event generated by the break detection will indicate when to continue with the next collar. Next to the collar break detection, the *LocalJobControl* is to be considered a software module.

UniversalRobot10 is the robot controlled by the job controller. It has the collar feeder, the Bosch XDK, and a 2D profile sensor attached. The UR10 supports different interfaces that can be used to control the robot and receive data. The acceleration from the integrated sensor can be accessed with a sampling interval of 125Hz .

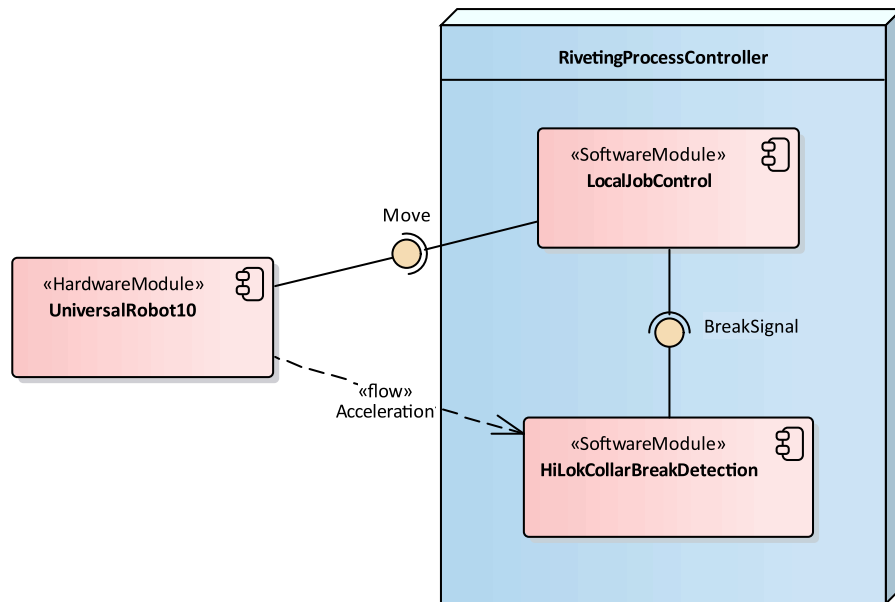


Figure 4.6.: Break detection configuration with UR10

BoschXDK is the embedded microcontroller device attached to the end-effector of the robot. It is programmed to stream the acceleration data from its included acceleration sensor over the serial interface.

4.3.3. Parameters and Application Settings

The application has to be configured to connect to the appropriate sensor and instantiate the required interface. Besides, the parameters for the classifier and the detection state machine have to be set. Default values are set within the program. Sensor dependencies can be passed by the program start or set as environmental variables. Also, a configuration file is available, where all settings can be edited. The break detection settings are described briefly in Table 4.1.

4.4. Problem Description

Based on the current implementation, several difficulties arise that complicated the integration and deployment of the application. Some of the problems are also true for other applications. These are to be solved and are described in detail below.

Table 4.1.: Parameter setting of the break detection

Parameter	Description	Default
WindowSizeAcc	Sets the number of samples that are included within the acceleration window. This value has to be adapted, if the input dimension of the classifier is change, for instance, to improve the performance of the classifier.	50
DataMax	The maximal value of the acceleration data that is used during the training of the classifier. It is later used, to normalize the acceleration data before it is fed into the classifier. This value can change, if the classifier is improved by new training data.	$\frac{m}{s^2}$
DataMin	The minimal value of the acceleration data used during the training of the classifier. Same as the <code>data_max</code> value, it is used for the normalization.	$\frac{m}{s^2}$
DataMean	The average value of the train data. It is also used for the normalization.	$\frac{m}{s^2}$
DataStdderiv	The standard derivation of the train data. It is also used for the normalization.	$\frac{m}{s^2}$
timeout	The predefined timeout value, the robot will try to screw a collar. Within this time the detection is active. Otherwise it will resume into a read state before it is invoked again by the <i>LocalJobControl</i> .	10s
Sensor	This parameter will determine if the sensor from the Bosch XDK or the UR10 is utilized for the detection.	xdk
RobotIp	The IP address of the robot, the application will connect to if the <code>sensor</code> parameter was set for the use of the UR10.	10.4.2.50
RobotPort	The port of the endpoint the robot provides its RTDE interface.	5001
SensorSerial	The path for the serial port the Bosch XDK is connected to. It is used, if the <code>sensor</code> parameter was set for the Bosch XDK.	/dev/ttyACM0

4.4.1. Device Dependencies

Mostly the acceleration sensor, required by the break detection, can be replaced by every sensor available and attached close to the TCP of the UR10. It is only required to add some additional training data from a new sensor to adjust the existing classifier. Notwithstanding, the problem of switching to a different sensor, is that with each different sensor, almost gua-

ranted, a new interface has to be implemented. This reduces the flexibility of the application since, for each sensor, the application needs adaptation to support a new sensor.

This problem does not relate only to this particular application. Exceptionally pure software applications like the break detection or the path planning application from chapter 1.1 are confronted with this problem. Even if these applications can be versatile usable, the limitation arise from incompatibly of the different interfaces that hardware components from different vendors provide.

4.4.2. Application Configuration

As described, each module interacting with the application is an independent running component communicating by some interface. Therefore, each module needs a way to be configured. In the case of the break detection, several options to configure different parameters have been offered.

The configuration is especially challenging for modular, independent, and distributed applications because they are comprised of multiple services and dependencies. Technologies like the Robot Operation System (ROS) or Docker offer the possibility to deploy modular software systems like the riveting system. With these, module configurations are usually stored in one or more files that. These, however, are stored somewhere on the controller onto the software is deployed. To find these files and navigate within is often tricky for people how are not familiar with the system.

Either way, if the applications are configured and deployed individual or in with the mentioned technologies, this process often requires advanced knowledge and reduces the ability to exchange components easily.

For ML applications, in particular, there is a need for tracking the configuration an version of the models used. Over time more and more training data is generated that is used to improve the models. Consequently, the models have to be replaced in the actual applications they are deployed in. Improvements may lead to changes in the model and, therefore, the configurations. This makes it especially hard to track if several ways of configurations and configuration files are available.

4.5. Implementation Objective

To facilitated the exchange of modules and configuration of applications, several principles should be adapted for the implementation of the break detection application. These are described here. Because some of these principles and concepts are not and will not be

implemented within the scope of this work, these account for the conceptual nature of this work.

4.5.1. Modularization

The overall modularization concept is based on the SORA and is extended to define not only different types of software modules but also hardware modules. A SORA groups software module in three different types of modules:

Hardware software modules are in charge of communication with physical sensors and actuators. Consequently, these are restricted to a particular type of hardware that functionality is implemented. Essentially, these modules map the device interface provided by manufacturers to a uniform interface based on an agreed communication protocol. This concept has already been adapted by the openMOS as *DeviceAdapter*, as shown in figure 2.1. It should facilitate the replacement of devices without adapting the interface implementation of depending modules that utilized the devices functionality. Overall the *DeviceAdapter* can be considered the functional sub-model of the AAS because it is associated of a specific asset or device and only should describing its functionality.

Software modules or services in charge of data processing and high-level algorithms for autonomy and control. These are not restricted to one particular kind of hardware. Instead, they mostly depend on a specific class or type of hardware. This means they can utilize the functionality of different hardware modules, as long as they implement the same functionality, for instance, cameras from different manufacturers, that implement the same functionality that is made available by *DeviceAdapter* modules with the same interface implementation.

Infrastructure modules that are performing tasks ranging from audible notifications over bandwidth management to the provision of common information and configurations.

Based on this concept, each software application and device is to be considered as a module. For now, it should be differentiated between hardware and software modules, while software modules should be further specialized as hardware depended modules, thus *DeviceAdapters*, and infrastructure modules.

4.5.2. Information Modeling

In an I4.0 all I4.0 components should embed information to allow more autonomous acting in changing circumstances. This means that common information model has to be created.

Information from this model is to be embedded into models of devices and applications to make their information interpretable by machines. This reduces the manual effort that is usually required to change and configure automation systems. By providing information on the nature of different modules and their capabilities, a system, as a combination of several devices and software applications, becomes aware of its overall capabilities and can decide on its own if it is able to execute a specific job.

4.5.3. Uniform Access of Skills

The skill model described in chapter 3.4, essentially classifies capabilities. This enables to resolve individual skills within a system. Together with the *DeviceAdapter* model, which embeds and provides the skills, more or less unified access of skills is possible. However, depending on the type of skill, it is possible to implement a skill as a simple method. Other skills may only provide a published sensor value, and even others may have a more complex invocation scheme, conditioned by the nature of the skill. The variety of skills implementations would induce difficulty in accessing skills. An inconsistency could lead to trouble when exchanging devices or modules, when the implementation of skills differ. To prevent this inconsistency, each skill should be implemented based on the same patterns. Also, each skill should provide information about its type and parameters that are needed to utilize it.

4.5.4. Configuration

Running systems with multiple independent services and modules requires maintaining configuration settings across different environments and updating them in production deployments. It is also essential to have the ability to quickly and easily roll back in case of an unintended or unexpected result. This is especially hard if configurations have to be edited in several places. Therefore all configurations should be provided to the modules in the same way, easily accessible and trackable. This should reduce the configuration effort and configurations better traceable.

4.5.5. Proposed Integration Concept

Based on the described principles, ideas, and concepts, the break detection application should be integrated into the process. Means, the break detection is to be considered a software module according to the modularization concept of a SORA since it provides high-level data processing for more autonomy. The information of its nature as a software module and all the information needed to integrate it into the process, should be embedded into

the module. Therefore it should contain information about its capabilities and dependencies based on a common information model that also each other module has to rely on. The information should be embedded employing OPC UA, whereas OPC UA enables to provide this information on a communication level and is considered the communication standard of choice for an I4.0. This communication standard also has to be adapted by the device adapters that provide the functionalities of connected devices to the software modules within the system.

Furthermore, the capability of detecting the Hi-Lok break should be provided by a state of the art approach introduced by [11]. This approach utilizes OPC UA programs to make capabilities of modules available in a unified way. Each capability provided this way is considered a skill and thus is to be classified as one. Therefore each module should provide its skills this way and share the information of skill types embedded into their information models.

To configure the break detection application, the configuration should be embedded into an AML model that is based on the same information as the OPC UA models and created by utilizing the AML information modeling mechanism. The AML model should also contain the configurations of all other modules deployed within the riveting system. This way it should be ensured that the system configuration can be reliably handled and tracked. Additionally, this way of managing the configurations enables the possibility to make available the configurations to all modules using OPC UA.

Figure 4.7 shows how the integration of the break detection should look like, according to the described concept. It provides its *BreakDetectionSkill* to the *LocalJobControl*. The *UniversalRobot10* provides the *AccelerationSensingSkill* to the break detection, whereby this skill could also be provided by the Bosch XDK, which is not depicted in this configuration. However, the *AccelerationSensingSkill* defines a requirement of the break detection, just as the *BreakDetectionSkill* and the *MoveSkill* define requirements of the *LocalJobControl*. For the break detection and additional requirement is introduced by the *ClassifierDownload* from the *ClassifierDatabase*, which should be considered when modeling the break detection application. The *ClassifierDatabase* is the only module not deployed within the *RivetingProcessController*, but rather in a *CloudSystem*, where also the classifier for the break detection should be trained and improved. Ultimately each module deployed on the *RivetingProcessController* should refer to the *AMLProcessModelServer* for its configuration, indicated by the named config dependency.

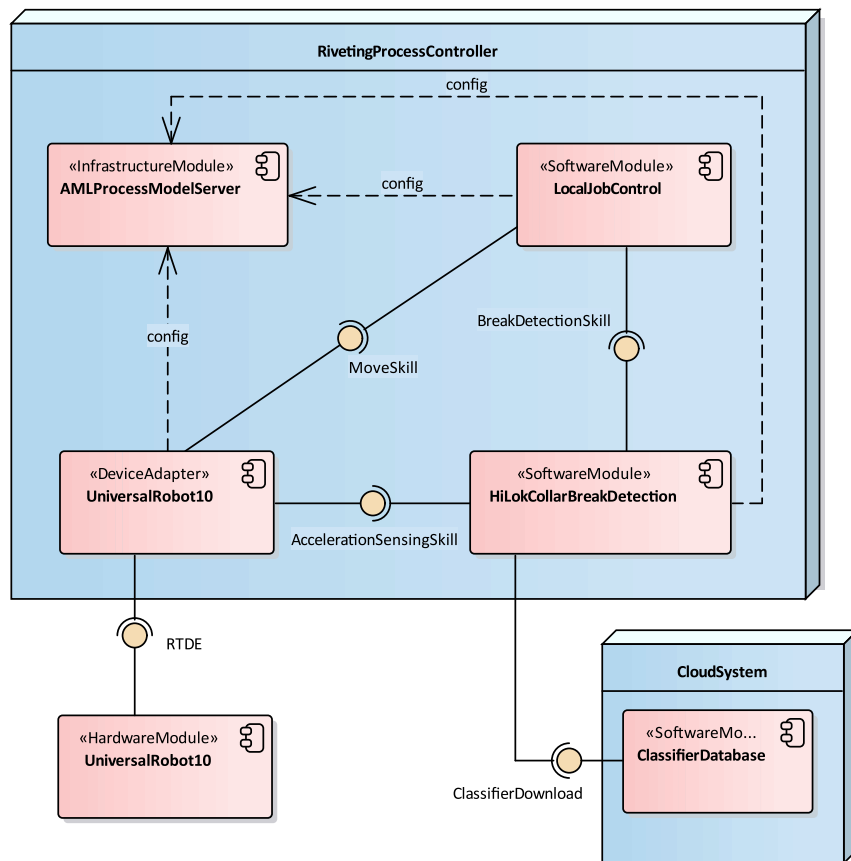


Figure 4.7.: Break detection concept

5. Modeling

The implementation has four main parts. First, a common semantic model is created. Based on this model, the AML model derived, which contains the required configuration of modules. The OPC UA model incorporates the information on a communication level. Finally, an OPC UA client is implemented to use the information stored in the models to resolve configurations and dependencies.

5.1. Common Semantic Model

To provide the information to the different modules and navigate the information, it needs a common understanding each module can share. Thus a simple, common semantic model based on the skill, PPR, and modularisation concept is created in UML. This model provides the information that each other information model, either in AML or OPC UA, should incorporate to derive the common knowledge. Based on this model, each application should be possible do derive the knowledge needed, to resolve dependency and configurations.

5.1.1. UML Model

As shown in figure 5.1, the semantic model describes modules as some kind of resource. Each module, either hardware or software module, can have several skills and requirements. Skills are specialized to be either atomic or composite. Requirements are specialized to be either a device requirement or a skill requirement. Composed skills are a composition of several atomic or also composite skills, which means that each composite skill has to have at least two skill requirements that resolve the dependency on the underlying skills. Device adapters, however, have precisely one device requirement. This requirement describes the dependency regarding the hardware module of which the functionality it implements. Besides, each requirement and skill could provide a parameter set. For a device requirement, this contains the necessary device setting, like the sampling rate of a sensor. A skill requirement, however, stores the required parameters that have to be passed, similar to method parameters in a method call. Therefore, the parameter set in the skill requirement should match the

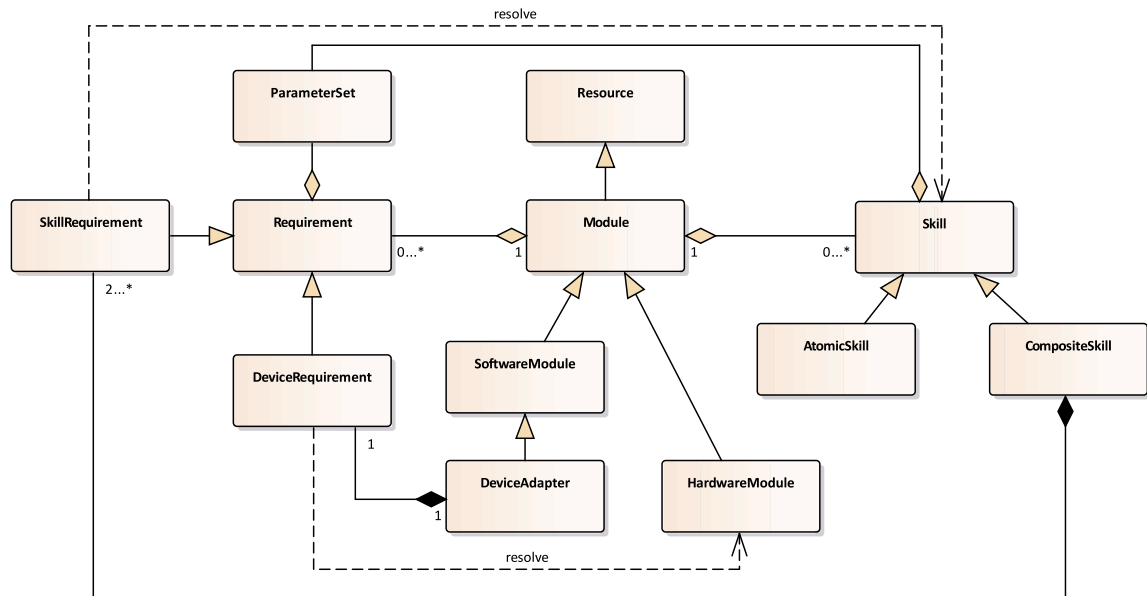


Figure 5.1.: Semantic model in a UML class diagram

parameters defined by the skill. Same for the device requirements, where the values stored in the parameter set should match the property of the associated device.

5.1.2. Skill Taxonomy

In addition to the semantic model, a skill taxonomy is defined, defining a set of skill types. The taxonomy is comparatively simple since it only covers the skill set of the break detection application. The acceleration sensing skill accommodates for the acceleration sensing, supplying the input data for the classifier. It is a specialization of a general sensing skill. As an additional example the image sensing skill is shown. For The break detection itself, a signal generating skill is defined as a specialization of logical skill. Additional logical skill can be defined if needed. For instance, into planning skills, which would accommodate a skill type for path planning as described in the example from 1.1.

5.2. AutomationML Model

The AML model should provide the required configurations of the system. References to dependency, like hardware and other, should be contained. Based on this, each application should be able to resolve its requirements and access its configurations. This is done by

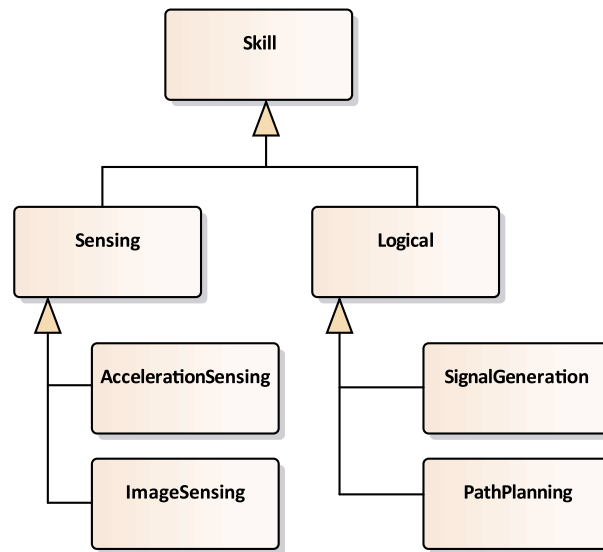


Figure 5.2.: Skill taxonomy in UML

converting the AML model to an OPC UA server. The general modeling approach is adapted and simplified from the modeling approach of the openMOS project.

5.2.1. Role Definitions

The semantics defined in the UML model has to be embedded in the AML model. This can be done, either by defining the classes of the UML model as system unit classes in AML or define appropriate AML roles. In this case, both options are used. This results in a certain degree of redundant information.

First, an appropriate set of roles is defined in an example riveting role class library, as shown in figure 5.3. Skills, modules, and requirement rules are defined. Roles provide semantics without defining the underlying technical implementation. However, while the redundancy is given, which will become evident if later on, the system unit classes are defined, the skill roles, in this case, does not define atomic or composite skills. This would indicated a technical implementation of skills. Instead, the skill role definitions define the type of the skills defined by the skill taxonomy, as shown in 5.2.

OPC UA Servers

In AML description of OPC UA servers has been presented in the best practice recommendation for data variables [38]. The OPC UA server role is defined as a specialized type of

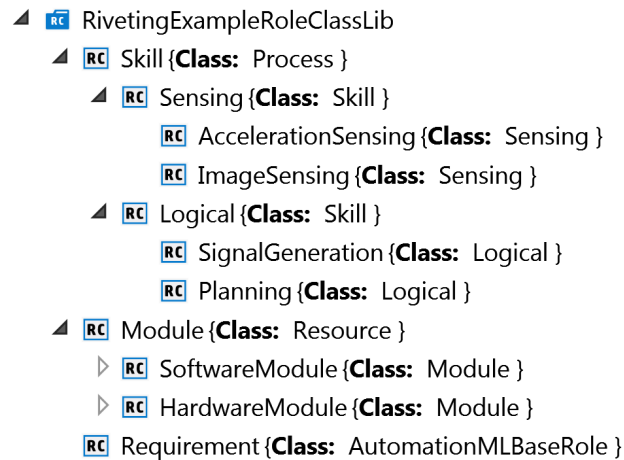


Figure 5.3.: Excerpt from the role definition of data variables

data sources. An excerpt of the data source definition is shown in figure 5.4a. It also defines a set of attributes. These attributes are shown in 5.4b and ultimately provide the required information to identify and connect to a server.

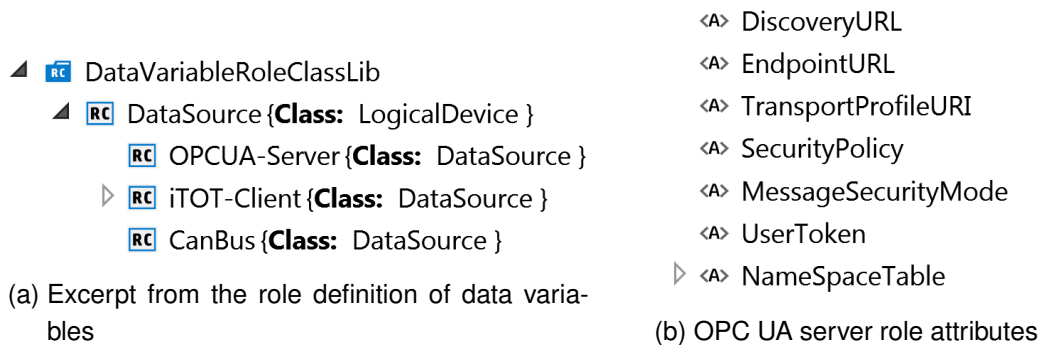


Figure 5.4.: OPC UA server definition in AML

5.2.2. Interface Definitions

For now, only one custom interface type is defined, as shown in 5.5. The requirement interface will connect requirements and their dependencies by creating internal links within the instance hierarchy. Within a requirement, it can be seen as a required interface, while in skills and others, it is instead a provided interface, requirements can connect to. In context to the UML semantic model, the interface would define the “resolve” dependencies between requirements and their targets.


- ▲  RivetingExampleInterfaceClassLib
 - RequirementConnector {**Class:** LogicalEndPoint }

Figure 5.5.: AML requirement interface.

5.2.3. System Unit Class Definitions

The system unit classes defined, define reusable classes that later can be instantiated in the instance hierarchy. Also, each class defined can be added as an internal element to another class definition. This allows, to build a library of classes that can be reused and combined for the modeling of any other process or system.

Skills and Requirements

Skills and requirements represent the modules capabilities and their dependencies. The system unit classes for skills and requirements are defined in the *RivetingExample_Skills_UnitClassLib*, as shown in 5.6. Skills are specialized as atomic and composite skills. Each of these skills will have at least one requirement connector that later allows resolving these skills.












- ▲  RivetingExample_Skills_UnitClassLib
 - ▲  Skill {**Role:** Skill}
 - ▲  Skill-Interfaces
 - RequirementConnector {**Class:** RequirementConnector }
 -  AtomicSkill {**Class:** Skill}
 -  CompositeSkill {**Class:** Skill}
 - ▲  Requirement {**Role:** Requirement}
 - ▲  Requirement-Interfaces
 - RequirementConnector {**Class:** RequirementConnector }
 -  SkillRequirement {**Class:** Requirement}
 -  DeviceRequirement {**Class:** Requirement}
 -  DatabaseRequirement {**Class:** Requirement}
 -  AdapterRequirement {**Class:** Requirement}

Figure 5.6.: AML system unit classes for skills and requirements

Requirements, however, are specialized in resolving skill, device, database, and adapter requirements. Skill requirements will be resolved by connecting to the requirement connector of the associated skill. The device and adapter requirements are complementary. A device

will define an adapter requirement, while a device adapter will contain a device requirement. This way, devices and associated adapters are connected.

Moreover, the skill and requirement classes will define a parameter set attribute, to define parameters that have to be configured to accommodate configurations provided by skills, devices, or other. Parameters will be added to the parameter set when instantiating these classes, which means when creating internal elements from the defined classes.

Modules

In figure 5.1, the model defines software and hardware modules. These will be created as system unit classes in the *RivetingExample_Modules_UnitClassLib*. In the UML model, only device adapters are defined as a specialized type of software modules. This does not mean that additional types can be derived because, ultimately, every program or application can be categorized as some kind of software module.

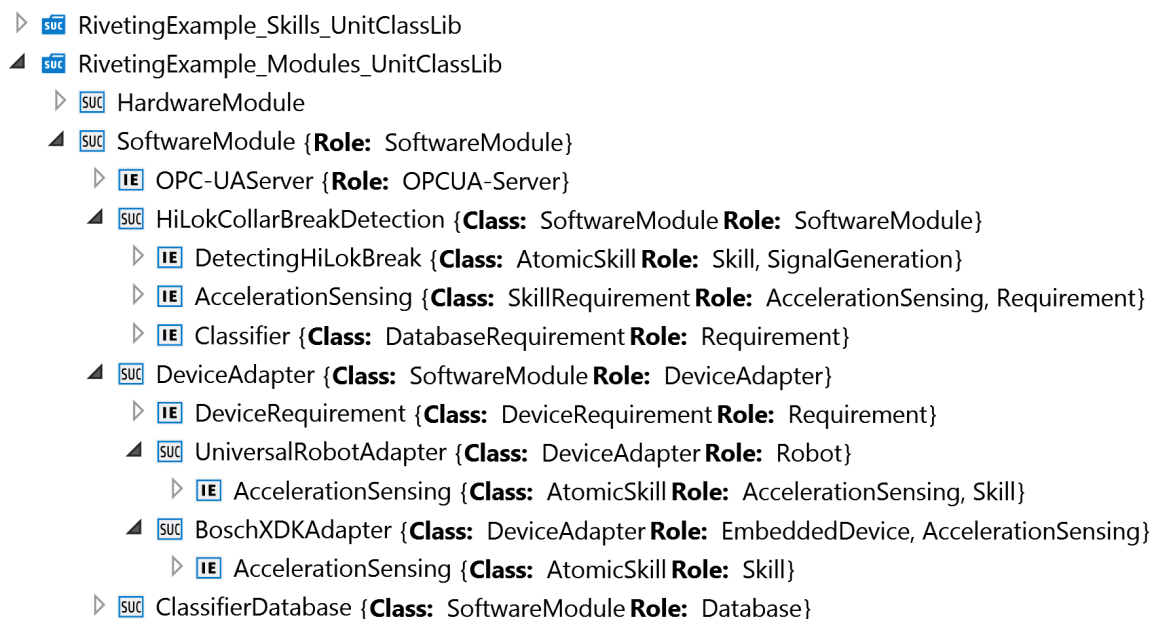


Figure 5.7.: AML system unit classes for modules

As shown in figure 5.7, the break detection application is defined as a software module. The break detection provides an atomic skill which is named *DetectingHiLokBreak* and typed by associating the internal skill element with the *SignalGeneration* role. The *AccelerationSensing* internal element is a skill requirement, that is typed by associating it with an *AccelerationSensing* role. It defines a dependency to an acceleration sensor with the acceleration sensing

skill. To enable the adaption of the classifier during run-time, the classifier *DatabaseRequirement* defines the associated database where the classifier can be downloaded.

Also defined are the device adapter software modules for the Bosch XDK and the UR10. These usually would provide more skills, but for the scope of this work, only the acceleration sensing skill is defined within each. Further, the device adapter class has a device requirement defined to accommodate for the device dependency. Devices would be categorized and defined in the hardware module class. However, these would not be considered within this model, since all dependencies are resolved by the device adapters, and are therefore unimportant for the scope of this work.

5.2.4. Process Instance Hierarchy

The defined system unit classes can now be utilized to create an instance hierarchy. For each configuration of the break detection application, one separated instance hierarchy will be constructed and saved in separated AML files.

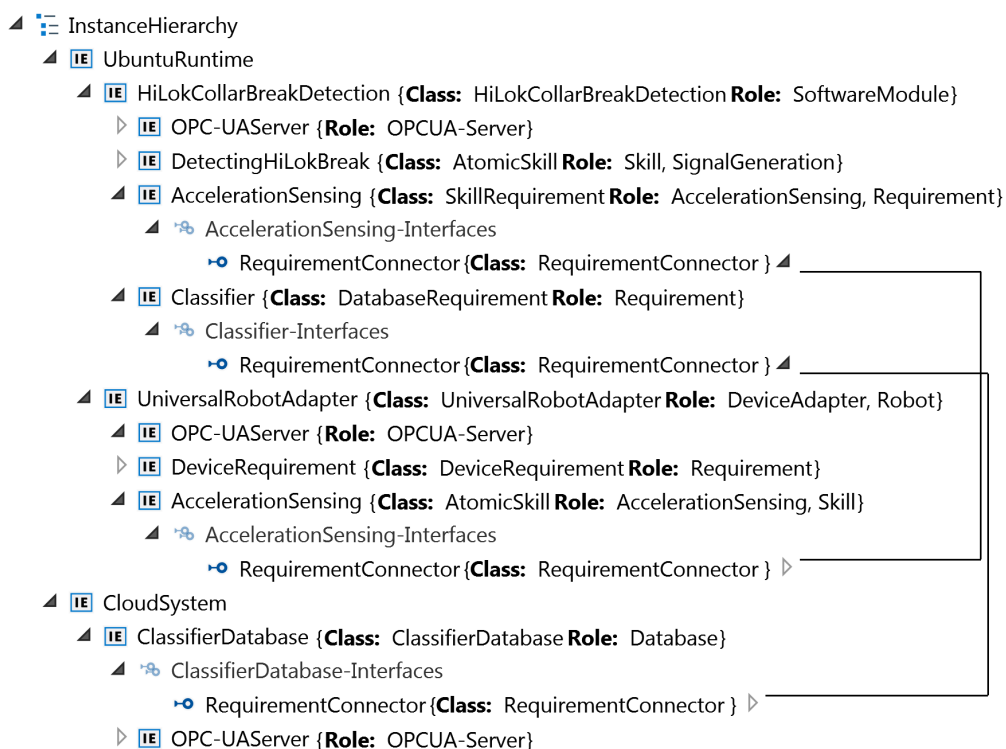


Figure 5.8.: Instance hierarchy with the UR10 as an acceleration source.

Instance Hierarchy with UR10

Since the software modules are deployed on the IPC, a runtime is defined, within the modules are deployed. In this configuration, an instance of the break detection and the UR10 device adapter are created, as shown in figure 5.8. Additionally, a cloud system is created, where the database with the classifier could be deployed. After instantiating the required modules, dependency could be resolved by creating the internal links between the requirement connectors and the targets. The acceleration sensing skill requirement of the break detection is resolved by the acceleration sensing skill of the UR10 and the database requirement is connected to the requirement connector of the classifier database.

Finally, the OPC UA server and parameter and parameter settings of the requirements and skills are edited. For each server, the discovery URL, endpoint URL, and the namespace table attributes are added. In the namespace table, only the application namespace is added with the index one. This way, each server can be resolved by a client browsing the later generated AML server. The namespace entry of the namespace table ensures a client can access the custom application objects. All other namespaces are considered standard namespaces containing type definitions and nodes that are shared and well known by each client and server in the network.

For the classifier requirement the parameters of the break detection according to the parameter definition from table 4.1 are set. The define parameters for the classifier are shown in 5.9a. These are assigned to the classifier since they affect its behavior. This is because the parameters have to be calculated during the training of the classifier. Therefore they will be contained in the classifier database, together with the classifier itself.

For the acceleration sensor dependency only the sampling time and measurement range are added. as shown in 5.9b. The acceleration sensor should ultimately provide the sensor data with a publishing rate of $125Hz$ and a range from $-32m/s^2$ to $32m/s^2$.

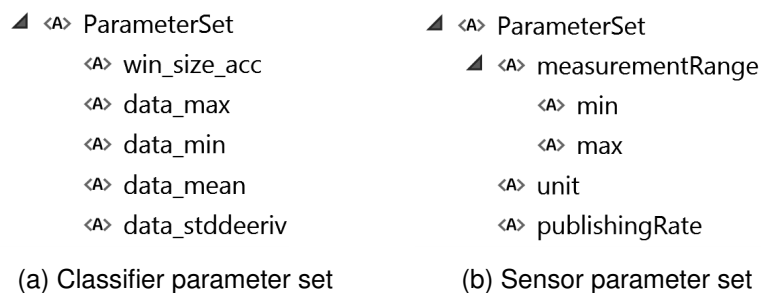


Figure 5.9.: Parameter definitions for the application requirements

Instance Hierarchy with Bosch XDK

As an alternative sensor, the device adapter of the Bosch XDK can be added from the system unit class library to the instance hierarchy, instead of the UR10. The parameter setting remains the same, only the OPC UA server parameters of the Bosch XDK device adapter server have to be added. The resulting instance hierarchy is shown in figure 5.10.

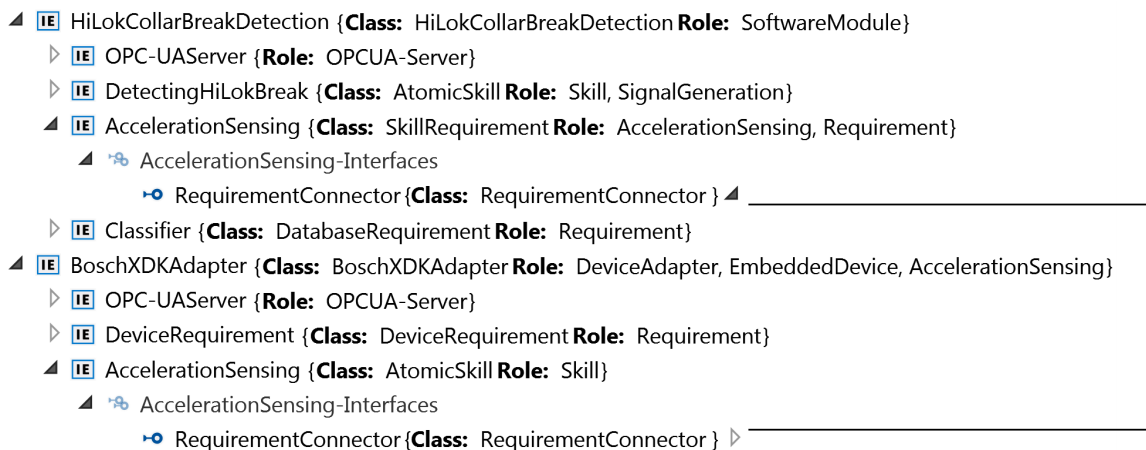


Figure 5.10.: Instance hierarchy excerpt with the Bosch XDK instead of the UR10.

5.2.5. Model Conversion to OPC UA

The resulting model can be converted to OPC UA, following the rules specified in the DIN SPEC 16592. To do so, the Fraunhofer IOSB provides a web-based tool [49] where models can be uploaded and converted to OPC UA. Resulting files can be downloaded, and for the server an executable file is included. Also, the generated code is delivered and can be edited and recompiled. Unfortunately, this tool is not based on the latest version of AML specification nor the OPC UA specification. For this application, it is acceptable, since the generated server is running as an independent instance, and the client application introduced in 6.2 can be adapted to follow either the newest specification or the outdated.

In figure 5.11a, an excerpt from the resulting address-space of the OPC UA is shown. Next to the OPC UA base name-space, the AML name-space with the type definition from the DIN SPEC 16592 and the actual model name-space will be available. Within the model name-space, the definitions from the AML file are available. It is placed in the *AutomationMIFiles* folder object. Here the instance hierarchy and the type definitions are available. Furthermore, the instance hierarchy will be available in the *AutomationMLInstanceHierarchies* folder. The instance hierarchy is placed by a *HasComponent* reference, in the *RivetingPro-*

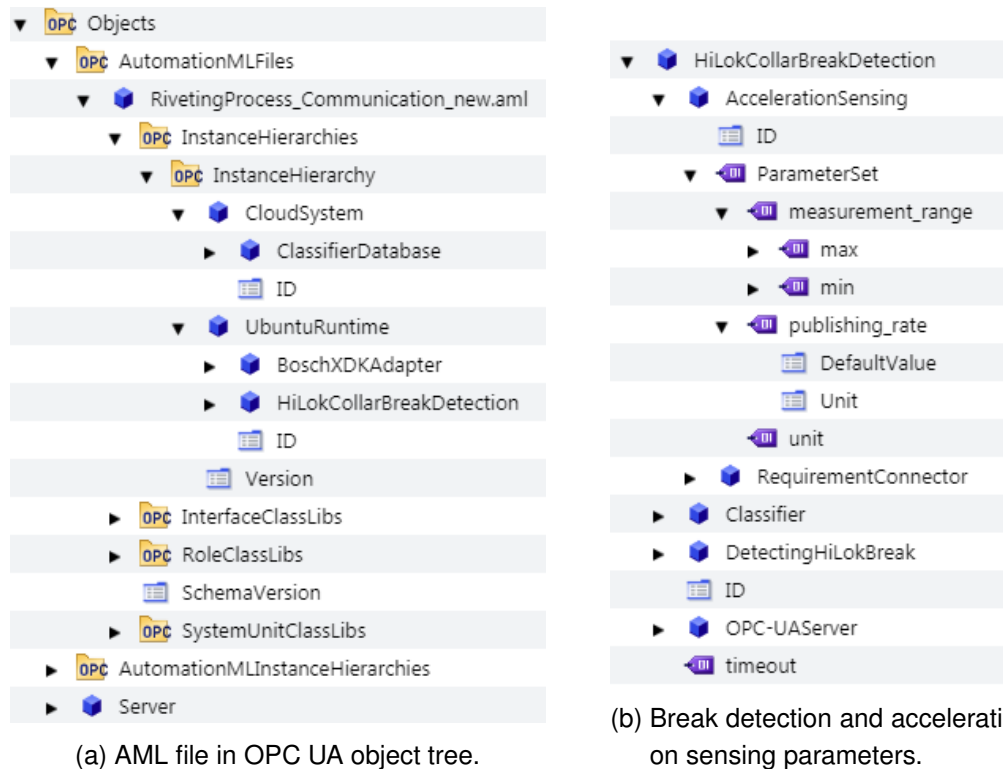


Figure 5.11.: Resulting object tree in the OPC UA server.

cess_Communication_new.aml object and referenced with an *Organizes* reference from the *AutomationMLInstanceHierarchis* folder.

Figure 5.11b shows the resulting object of the break detection application, with the expanded acceleration sensing requirement. Parameters defined in the parameter set of the requirement will be available in the OPC UA server as OPC UA variables, typed as *AMLVariableType*. The requirement connector of the acceleration sensing requirement is converted to an OPC UA object. It has a non-hierarchical reference of type *HasAMLInternalLink*, referencing the node ID of the requirement connector of the Bosch XDKs acceleration sensing skill.

5.3. OPC UA Models

The OPC UA models describe the actual, reusable modules. Here all configuration, data, and functionality are contained and accessible. First, the models, are created, and the resulting XML file is used to generate the server code. Then the generated code can be used to implement the actual server logic or just deployed.

5.3.1. Modeling Software

To model the information models of applications and devices, the Siemens OPC UA Modeling Editor (SiOME) can be utilized [50]. This editor was initially intended to create models based on the OPC UA companion specification and mapping values within the models to the logic controller of the SIMATIC family. However, the creation of OPC UA information models can be utilized without the need to connect any logic controller, and the ability to export the information models in an XML format makes it a general-purpose OPC UA modeling editor. Figure 5.12 gives an overview of the editor.

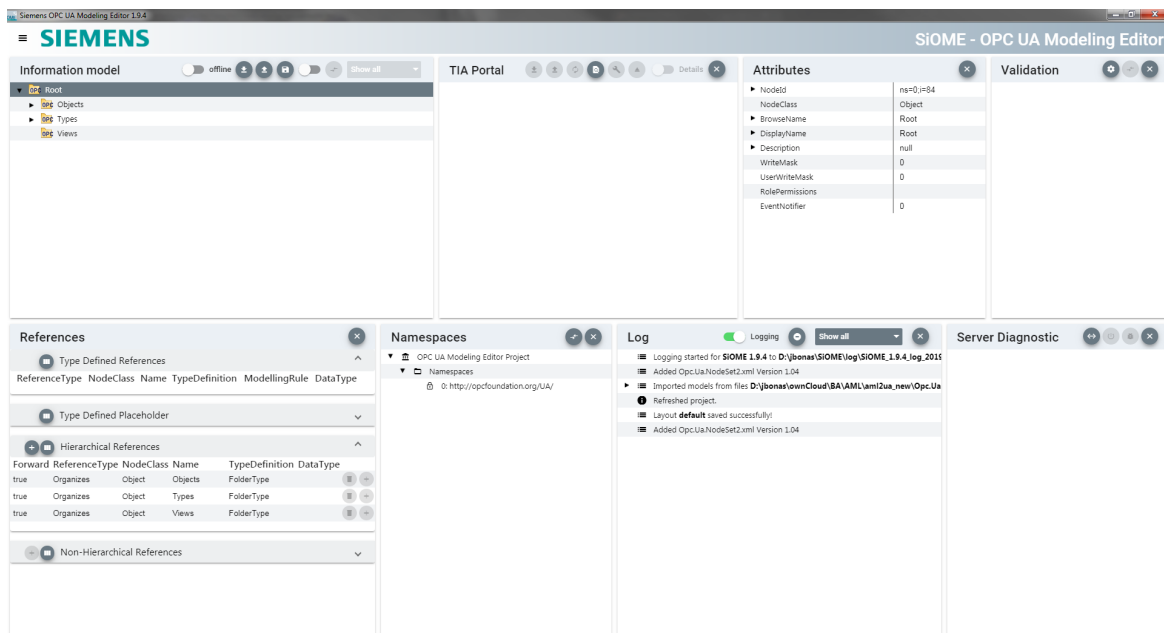


Figure 5.12.: Overview SiOME

5.3.2. Type Definitions

Before starting to model the actual information model of any application, a set of types has to be defined, based on the common semantic model.

Modules

As described in the semantic model, it is distinguished between hardware and software modules. Dependent on the module type, different module properties are represented. In the

OPC UA specification for devices, the *TopologyElementType* is defined. From it, the *ComponentType* is derived. Modules, however, could be seen as components, without further specialization. The OPC UA device specification defines *DeviceTypes* and *SoftwareTypes* as components. Each component could have an *AssetId*. The *SoftwareType* definition is shown in figure 5.13.

Nevertheless, since the semantic model defines modules as a type, it would be unappropriate to go with the different *ComponentType* definitions, since the semantic model defines the information that should be shared between models and modeling languages. Therefore the *ModuleType* is defined as a specialized type of *ComponentType*. These types are defined in the <http://ifam.fraunhofer.de/UA/ModuleTypes/> namespace.

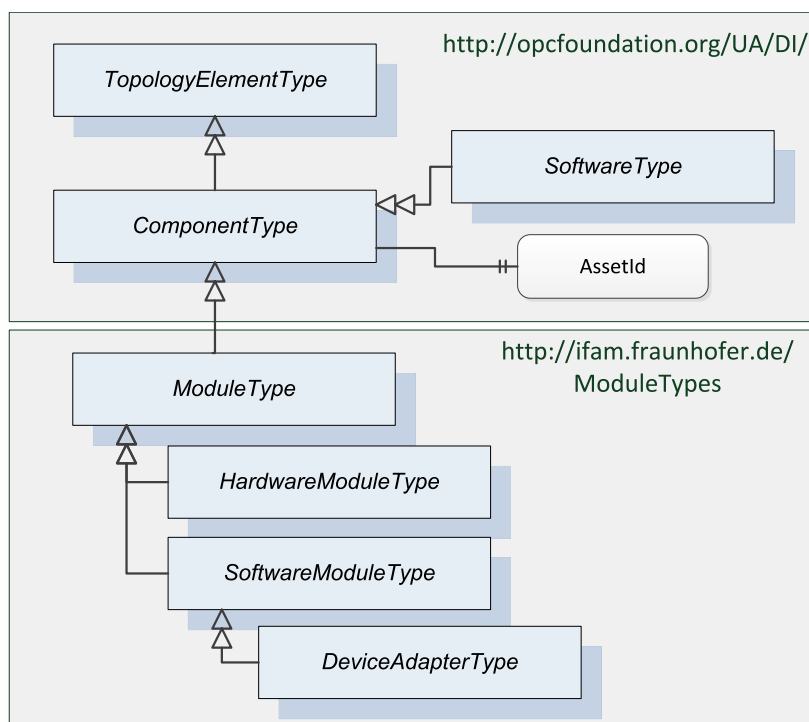


Figure 5.13.: Module types defined in OPC UA

Requirements

Requirements will be directly derived from the *BaseObjectType*, as shown in figure 5.14. These should resolve requirements that are contained in different OPC UA server. Consequently, requirements will contain a node ID and discovery URL as properties. Later on, this information can be resolved from the AML server and be used to access the requirements on the appropriate server in the network.

To uniformly access the requirements of modules, a requirement interface is defined. The interface defines a requirements object, which on the other hand, organizes the requirements of the whole server by referencing the requirements in the address-space with an *Organizes* reference.

Furthermore, the custom reference type *HasRequiredType* is defined. This reference could accommodate, for instance, for the skill type of a defined skill requirement. It would be defined by the requirement and point to the required definition of the skill type. The same was done in the AML model by adding the role of the required type to the requirement.

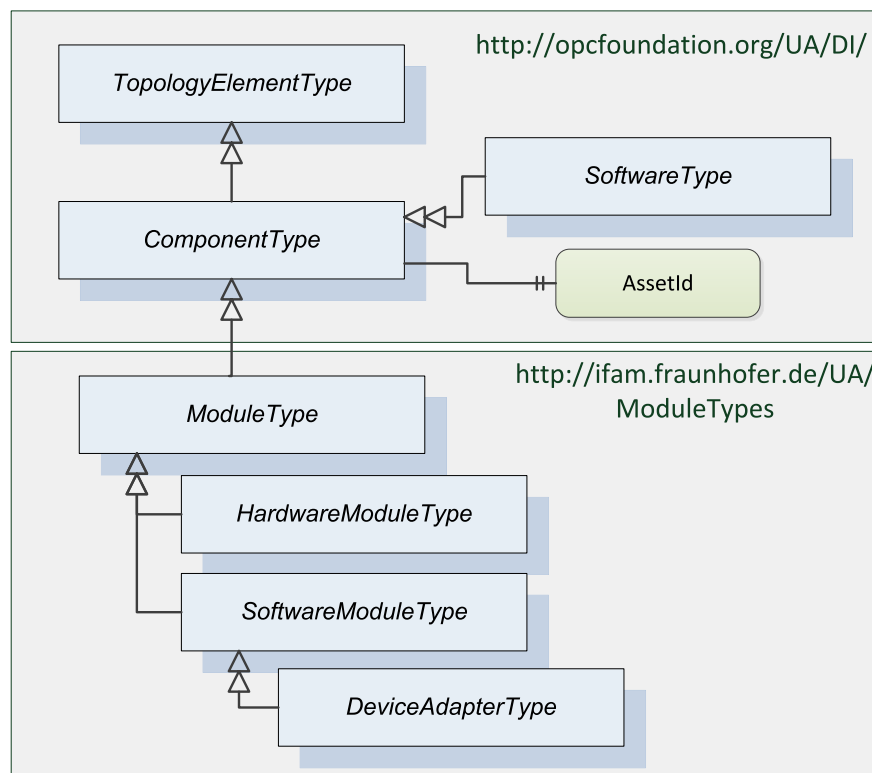


Figure 5.14.: Requirement types defined in OPC UA

Skills

Skill types are derived from OPC UA *ProgramStateMachineType*, as shown in figure 5.16. Here, only methods and states are defined. These are mandatory for every skill. Usually, the program would contain state-transitions, but these are omitted for clarity. A view of the skill definition object tree can be found in A. Transitions, however, are displayed in state machine diagram of the program in figure 5.15. The transition arrows between the states include the method name that triggers the transition and the name of the transition itself. Some of the

transitions are not triggered by any method. Instead, they are triggered by an internal server event. These are the *ReadyToHalted* and *SuspendedToReady* transitions. Moreover, each transition is an event source that will be generated if these transitions are triggered.

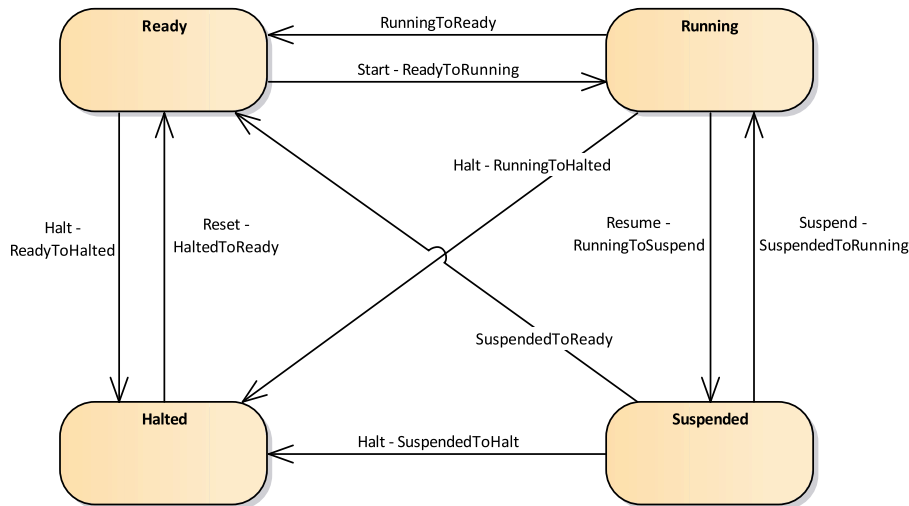


Figure 5.15.: Skill state machine

The approach of describing skills as OPC UA programs has been introduced by [11], and [12], while [4] used OPC UA state-machines. The approach of representing skills as OPC UA programs will be adapted here. With this approach, it becomes possible to access and control each skill more or less the same way and facilitates the execution of various skills that may have to be executed in parallel. First, the client who wants to invoke the skill, would, if required, write the skill parameters into the parameter set object and subscribe to the event, which is triggered by the *RunningToReady* transition. Then it can start the skill execution by invoking the start method. If the skill finished its execution, it would change the state from running to ready. Internally the associated event would be triggered. Since the client subscribed to this event before starting the skill, it would be notified that the skill execution finished. If the skill program defines a result data object, the client could now read the result data or continue with its task.

For the sensing skill, the program has to be customized to include an object *SensorValueDataSet* of type *PublishedDataItemsType*. This object will ultimately contain the sensor data that is to be published by the PubSub mechanism. In a way, this approach has been proposed by [4] who added the ability to publish skill execution information to provide better real-time performance for skill execution and monitoring. However, instead of adopting this approach for the whole state machine, in this case, it is only relevant for the *SensorValueDataSet*. Together with the program state machine, it should be possible to control the publisher by the defined methods of the program. Since the sensing skills would never trigger the *Run-*

ningToReady event, it would continue publishing data until a client terminates or pauses the skill by invoking the halt or suspend method. So instead of subscribing to the *RunningToReady* event, the client would subscribe to the published sensor value defined within the skill program.

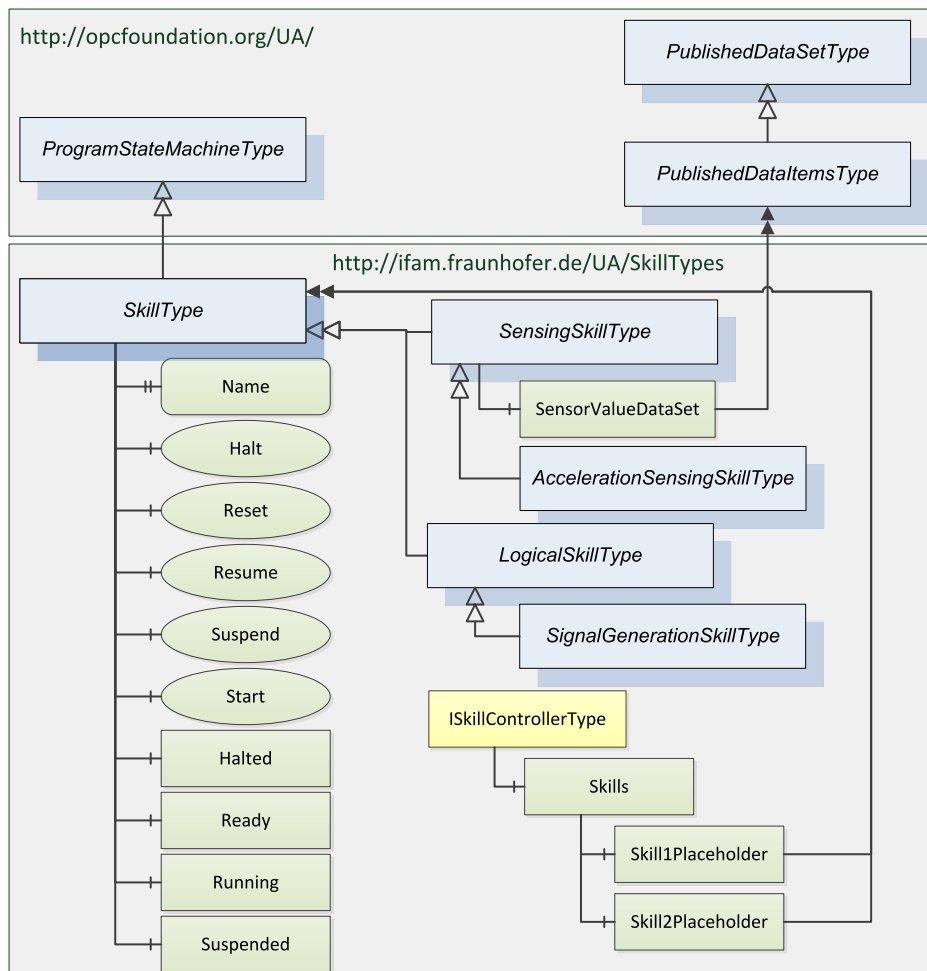


Figure 5.16.: Skill types defined in OPC UA

To facilitate the access of skills, the same way as for requirements, an *ISkillControllerType* interface is defined, as shown in figure 5.16. This is an OPC UA interface that each module that offers skills should define to make skills easy accessible. It will be indicated by the containing module typed object by a *HasInterface* reference, the same way as it has been done for requirements. The skill object is defined by the controller interface and will organize all skills that are available in the servers address-space.

Skill types defined here, have neither a parameter set or a result data object defined. These are optional, depending on the skill, and can be added after instantiating a skill or while

defining a new skill type. Skill types defined here accommodate for the break detection application.

Furthermore, there is again no distinction made between atomic and composite in the type definitions of skills. However, a composite skill would be a skill, according to the common semantic model, that has at least two skill requirement defined. Therefore there is no need to distinguish it further by type.

Machine Learning Classifier Type

Finally, an ML type is defined. More precisely, an ML classifier. This type is defined in the application namespace. It defines the structure of a classifier that each classifier could receive. The classifier model variable contains the classifier as a byte string. Depending on the programming language the server is implemented in, it is a byte string of the classifier object that is serialized or a file of a dynamic library. In the case of the break detection, the classifier would be received as a dynamic library, since the application will be implemented in C/C++.

The predict method represents the method that is ultimately used to make a prediction based on the given input data. By representing it within the server, it also could be utilized by a client. However, it will not be used by the server implementation.

The parameter set is defined within the classifier too. For the break detection application it contains the attributes defined in the AML model as AML variables. These have to be resolved from the classifier database.

5.3.3. Break Detection Application Model

With the defined types, all information is available to create the information models of the actual applications. Figure 5.18 shows a simplified representation of the break detection application. It consists of the ML classifier and the signal generating skill program, detecting the Hi-Lok collar break. It utilizes the type definitions from the previously defined namespaces. The break detection program *DetectingHiLokBreak* is typed as a *SignalGenerationSkillType* while the *HiLokBreakDetection* application object is typed as a *SoftwareModuleType* and the *Classifier* object as a *ClassifierType*. A detailed representation, with child variable and objects, like states and program methods, is omitted for clarity. A full overview of the address-space, however, can be found in B.

In addition to the underlying functional object, the required dependencies are defined. These are placed within the associated objects. Therefore the *DetectingHiLokBreak* skill program

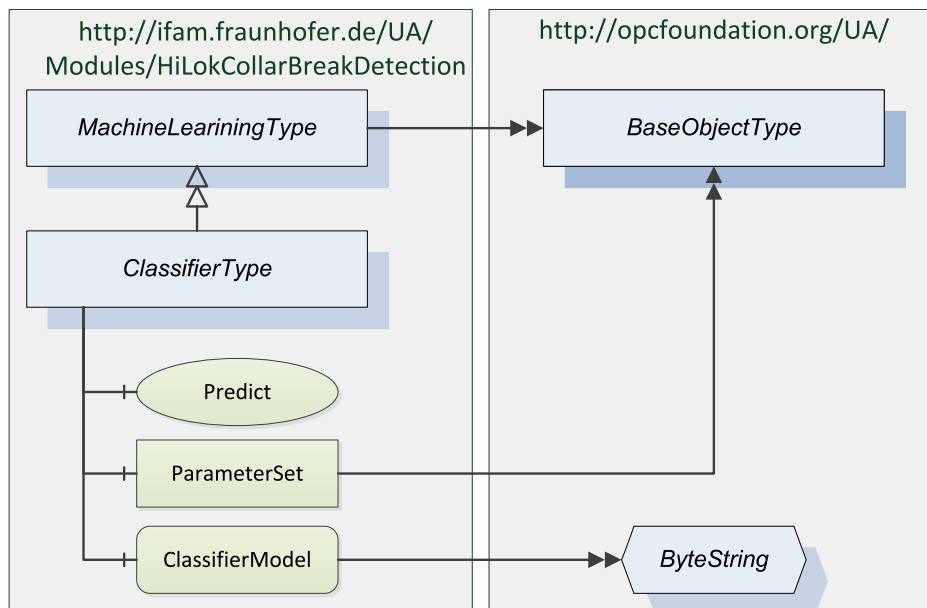


Figure 5.17.: ML type in OPC UA

defines the *AccelerationSensing* object as a *SkillRequirement*, resolving the required skill type by the custom-defined *HasRequiredType* reference. In the *Classifier*, the *ClassifierDatabase* is defined as a *DatabaseRequirement*, resolving the classifier source.

Another part of the application, the *DetectedHiLokBreaks* variable, is defined, to allow clients to monitor the performance of the application.

The shown model in figure 5.18 does not display the entirety of the application. For clarity, most of the objects, variables, and properties are omitted. However, the *Skills* object contains one hierarchical *Organizes* reference to the *DetectingHiLokBreak* skill program, since this is the only skill defined by the application. The *Requirements* contains two *Organizes* references to the *ClassifierDatabase* and *AccelerationSensing* requirements. Furthermore, the *HiLokCollarBreakDetection* module object defines two *HasInterface* references. One for the *ISkillControllerType* and one to *IRequirementsTypes*.

5.3.4. Sensor Models

The sensor model is an excerpt of the appropriate device adapter. In figure 5.19, it is shown for the UR10. Same as the break detection application, it uses the previously defined types. Usually, the device adapter would contain additional skills and variables to control and

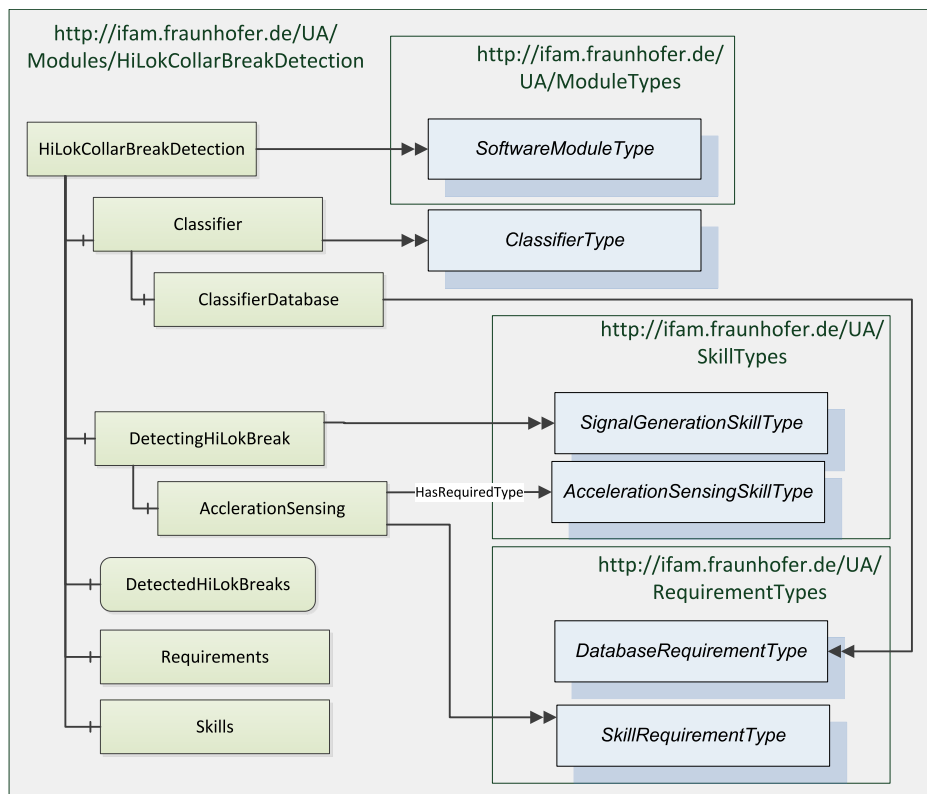


Figure 5.18.: Simplified view on the break detection server

monitor of the robot. However, for the integration of the break detection, only the *AccelerationSensing* skill is required. It is defined as a child of the *UR10Adapter* and typed as the *AccelerationSensingSkillType*.

The skill contains the same parameter set as previously defined in the AML model and by the acceleration sensing skill requirement of the break detection program. Also defined are the skills and requirements interfaces, which references are also omitted in the representation of the model. Therefore, different than previously in the representation of the break detection, the *Skills* object has the *Organizes* reference defined, targeting the *AccelerationSensing* skill.

5.3.5. Model Conversion to C Code

The created model can be exported in an XML format. Based on these files, it is possible to generate the program code. Depending on the OPC UA implementation and programming language used, there are several options available. For this work, the open-source open62541 [51] library will be utilized. It is an OPC UA C library that also provides Python

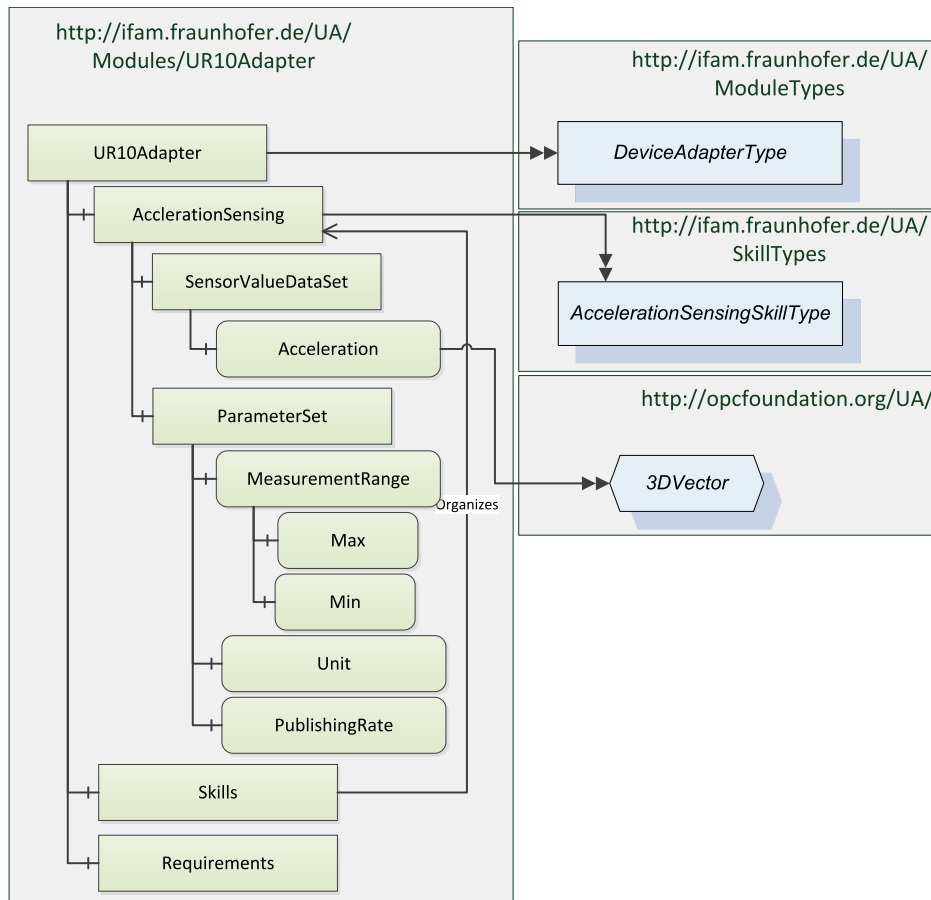


Figure 5.19.: UR10 sensor model

scripts to generate the servers C code. This Python application is also called a nodeset compiler.

Nevertheless, since the library is open-source and some the of OPC UA specifications have been released only in the last month, not all of the specifications have been realized within the library. Consequently resulting OPC UA programs, for instance, will define the structure of the address-space, but not the actual mechanisms that should be given by the program, like the triggering of events or transitions when calling a method.

However, when exporting the XML files from the SiOME editor, created namespaces could be exported apart to separated files or all into one file. While the export of separated files facilitates the distribution of common types and creation of new server models, the export of one fused file facilitates the work with the nodeset compiler. Usually, each existing nodeset needs to be included during the compilation of the nodeset by the `--existing` option. Although, since all definitions are included within one file, the command for compiling the nodeset simplifies to the command shown below.

Listing 5.1: Compiling the nodeset

```
python ./nodeset_compiler.py --types-array=UA_TYPES \<\  
    --existing Opc.Ua.NodeSet2.xml \<\  
    --xml ifam.breakdetection.xml breakdetection
```

This will create a `breakdetection.c` and `breakdetection.h` file, containing the code that ultimately creates the structure of the servers address-space. Therefore all is left to do, is to create a server struct and pass it to the generated `breakdetection` function. This will add all previously defined nodes to the server, as shown in listing C.1.

Listing 5.2: Server creation from generated code

```
#include <signal.h>  
#include <stdio.h>  
#include <open62541.h>  
// Including the generated header  
#include "breakdetection.h"  
  
UA_Boolean running = true;  
static void stopHandler(int sign) {  
    running = false;  
}  
  
int main(int argc, char **argv) {  
    // Creating the server configurations  
    UA_ServerConfig *config = UA_ServerConfig_new_default();  
    // Creating a default server  
    UA_Server *server = UA_Server_new(config);  
    // Adding the nodeset to the server  
    UA_StatusCode retval = breakdetection(server);  
    if(retval != UA_STATUSCODE_GOOD) {  
        UA_LOG_ERROR(UA_Log_Stdout, UA_LOGCATEGORY_SERVER,  
            "Unable to add the nodeset.");  
        retval = UA_STATUSCODE_BADUNEXPECTEDERROR;  
    } else {  
        retval = UA_Server_run(server, &running);  
    }  
    UA_Server_delete(server);  
    UA_ServerConfig_delete(config);  
    return (int) retval;  
}
```

6. Client Implementation

With the information embedded into the models, a simple OPC UA client application should show how to utilize the information to configure the application and resolve its dependency. Therefore several browse requests are defined to browse the AML server. With the browse results, it is then possible to connect to the appropriate server and access skills, databases, and other requirements. How an OPC UA client can be utilized to resolve requirements is shown here only for skills. In particular for the acceleration sensing skill of the break detection application.

To hold all information needed, to connect to skill, a *SkillClient* struct is defined. This struct is to be filled with the information from the defined information models by using the browse functionality of OPC UA. The *SkillClient* struct is shown in listing 6.1.

Listing 6.1: SkillClient struct

```
typedef struct SkillClient{
    UA_Client *client;
    UA_String endpointUrl;
    UA_ExpandedNodeId skillId;
    UA_String skillType;
}SkillClient;
```

The struct holds the client that will be connected to the server, providing the skill. To connect the client to the right server, the endpoint URL of the server is included. To resolve the skill within the server, the node ID is needed. It is saved as an expanded node ID. Expanded node IDs do hold not only the namespace index and ID but also the associated namespace URI. Endpoint *URL* and namespace *URI* have been included into the AML model. The node ID has to be resolved from the server that is providing the skill, and the skill type is provided by the server, which requires the skill.

6.1. Resolving Skill Requirements from the AML Server

To resolve the requirements from the AML server, an OPC UA client is created, that is will then connect to and browse the AML server. All information that is ultimately needed to

resolve dependencies is the AML server endpoint URL and the discovery URL of the module server whose dependencies are to be resolved.

In order to navigate the server, a number of browse requests are defined. The functions are described in the following:

getUAServerIdByDiscoveryUrl(*client, url, *serverId) resolves the OPC UA server by its given discovery URL. This function follows the inverse references of the OPC UA-Server role type definition. For each result, it reads the discovery URL variable and compares it with the passed discovery URL. If both match the servers node ID is saved in passed node ID reference.

readUAServerDiscoveryUrl(*client, serverId) reads and returns the discovery URL of the given OPC UA server node in the AML server.

readUaServerEndpointUrl(*client, id) reads and return the endpoint URL of the given OPC UA server node in the AML server.

isOpcUaServer(*client, id) checks if the given node ID belongs to an OPC UA server node object, based on the OPC UA server role type definition from AML .

getModuleUaServerId(*client, id, *result) finds the associated OPC UA server of a module in the AML server by checking recursive for each object within the module is a server.

isSkillRequirement(*client, id) checks if the node with the given ID is a skill requirement, based on the type definition from the AML system unit class library.

isRequirement(*client, id) checks if the node with the given ID is a requirement, based on the type definition role definition for requirements from AML role class library.

isObjectSubType(*client, type, node) checks if the given node is a sub-type of the type ID passed.

isSoftwareModule(*client, id) check if the node with the given ID is a software module, based on the role type definition for software modules from the AML role class.

getContainingModule(*client, id, *result) gets the node ID of the module that contains the node with the given ID. The result is saved in the result reference.

getModuleSkillRequirements(*client, id, *requirements, size) finds all skill requirements of a module and saves them in the requirements reference.

resolveAmlInternalLink(*client, id, *result) resolves the node the internal link reference is pointing to. If the requirement is a skill requirement, the resulting ID of the target node would be the ID of a skill type node.

`resolveRequirement(*client, id, *result)` finds the requirement connector of the node with the given ID. If the requirement connector has a *hasAMLInternalLink* reference, it uses the `resolveAmlInternalLink` function to get the node ID of the target node.

The browse request for the `resolveRequirements` function is shown in detail in Listing 6.2. It shows the basic concept of how to browse to utilize the browse functionality. First, the browse request is created. Afterward, the service function `UA_Client_browse` is used to make the service request at the connected AML server. Finally, the references returned by the browse request can be used in the for loops to check if the returned reference satisfies the required condition. If so, the referenced node ID required is returned.

Listing 6.2: Resolving module requirements

```

UA_StatusCode resolveRequirement(UA_Client *client, UA_NodeId *id,
                               UA_ExpandedNodeId *result){

    // The node ID of the requirement connector type
    UA_NodeId requirement_connector = UA_NODEID_STRING(process_idx,
                                                       AML_CLASSTYPEDEFINITION_REQUIREMENTCONNECTOR);

    // Creating the browse request
    UA_BrowseRequest req;
    UA_BrowseRequest_init(&req);
    req.nodesToBrowse = UA_BrowseDescription_new();
    req.requestedMaxReferencesPerNode = 0;
    req.nodesToBrowseSize = 1;
    req.nodesToBrowse[0].browseDirection = UA_BROWSEDIRECTION_FORWARD;
    req.nodesToBrowse[0].nodeId = *id;
    req.nodesToBrowse[0].resultMask = UA_BROWSERESULTMASK_ALL;
    req.nodesToBrowse[0].includeSubtypes = true;
    req.nodesToBrowse[0].referenceTypeId = UA_NODEID_NUMERIC(0,
                                                            UA_NS0ID_HASCOMPONENT);

    // Using the server browse service to answer the browse request
    UA_BrowseResponse resp = UA_Client_Service_browse(client, req);
    for (size_t i=0; i<resp.resultsSize; ++i){
        for (size_t j=0; j<resp.results[i].referencesSize; ++j){
            UA_ReferenceDescription *ref =
                &(resp.results[i].references[j]);

            // If the returned node reference is typed as a requirement
            // connector, the requirements internal link is resolved.
            if(UA_NodeId_equal(&requirement_connector,
                              &ref->typeDefinition.nodeId)){
                resolveAmlInternalLink(client,
                                       &ref->nodeId.nodeId, result);
            }
        }
    }
}

```



```

        return UA_STATUSCODE_GOOD;
    }
}

return UA_STATUSCODE_BADNOMATCH;
}

```

These browse requests can then be utilized to get the discovery, endpoint URL, and application namespace URI for the module server that provides the required dependency. In Listing 6.3, a sequence of the previous defined browse request is shown. At the end of this sequence the endpoint URL and namespace URI of the server providing the acceleration sensing skill to the break detection are recovered.

Listing 6.3: Finding the OPC UA servers with the break detection dependencies

```

// Find the server of the break detection application in the AML server
getUAServerIdByDiscoveryUrl(client,
    UA_STRING(
        "opc.tcp://0.0.0.0:5003/RivetingProcess/HiLokBreakDetection"),
    &serverId);
// Get the node ID of the break detection module in the AML model
getContainingModule(client, serverId, &moduleId);
// Get all skill requirements of the break detection module
getModuleSkillRequirements(client, &moduleId, requirements, 1);
// Get the requirement connector node ID of the targeted skill
resolveRequirement(client, &requirements[0], &dependencyId);
// Get the node ID of the module with the targeted dependency
getContainingModule(client, skillId.nodeId, &targetModuleId);
// Get the node ID of the OPC UA server object of the module
getModuleUaServerId(client, targetModuleId, &targetServerId);
// Get the server discovery URL
UA_String targetUrl = readUaServerDiscoveryUrl(client, targetServerId);
// Get the namespace URI of the target server application
UA_String targetUri = readUaServerNamespaceUri(client, targetServerId);

```

6.2. Resolving the Acceleration Sensing Skill

With the known endpoint *URL*, application namespace, and skill type, it is possible to connect to the server and resolve the skill within the targeted server. This is done by connecting to the server and browse it in order to find all objects that have the *HasInterface* reference that is pointing to the *ISkillControllerType*. Since the *ISkillControllerType* defines the interface by defining a *Skills* object within the object that is referencing the interface, all known skills

should be contained within the *Skills* object. Therefore, it is possible to browse the *Skills* object that is organizing the skills and find the skill with the required type.

Because it is not possible with the current implementation of the AML to OPC UA converter to export the defined library in a separated file, it is therefore not possible to share type definitions between the AML generated OPC UA server and the custom OPC UA server. This is why, for now, the type of skills required is checked by the names of the skill types. By comparing the defined AML role skill types with the OPC UA skill program name types, the types can be matched. It is not the optimal way to do so. Superior would be to share type definitions of skills between the AML server and the custom module servers, associate the skills with these types, and ultimately match skill type by their node IDs.

However, to find the right skill within the target server, an additional browse functions are defined, that utilizes the skills type name as a string to find the targeted skill in the device adapters. These functions are described in the following.

`getSkillByTypeName(*client, type, *result)` will resolve the skill by the skill type provided as a string. The delivered client is the *SkillClient* that is already connected to the target server. It is used to search for the objects reference the *ISkillControllerType* interface by following nodes with the *HasInterface*. If a node has this interface, it will search the *Skills* object for skills with the type definition of the type provided to the function.

`findSkillsObject(*client, *result)` finds the *Skills* interface object, organizing the servers skills and returns saves its node ID in the provided result reference.

6.3. Test Case

To test the resolution of the acceleration sensing skill of the break detection application, the AML, generated OPC UA server and the appropriate device adapter is deployed. First of all, the AML server created from the model in chapter 5.2.4, containing the break detection application, resolving its acceleration sensing skill requirement to the UR10, and the device adapter, created according to the OPC UA model from chapter 5.3.4, are deployed. Then the same is done for the Bosch XDK AML and OPC UA model.

In both cases, a configuration client program is started. The code of this program can be found in C. It will first create an AML client that connects to the AML server and resolves the acceleration sensing skill, using the functions described in 6.1. With the results from the AML client, a second client is created that connects to the server with the resolved endpoint URL. It will then resolve the acceleration sensing skill in the appropriate device adapter server,

using the functions defined before in chapter 6.2. All information is saved in the *SkillClient* struct.

In either case, the acceleration sensing skill is resolved, without changing the configuration client program. It shows that by embedding the right information into the AML and OPC UA models, software modules become more autonomous. The configuration client can potentially be used in every software module that is to be configured by accessing an AML configuration server without any need to change the module nor the configuration client implementation. All that is needed, is the modules own discovery URL and the endpoint of the AML server.

Furthermore, in combination with the skills implemented as a OPC UA program, the usage of sensors and other device functionality enables more flexibility when changing the configuration of a production system.

7. Discussion and Further Work

This work demonstrated a concept on how a pure software application represented by a machine learning application, for the detection of the break of Hi-Lok collars, can be integrated into an automation process by utilizing AML and OPC UA. Therefore the application and dependent components have been modeled with the information modeling techniques by both AML and OPC UA. A common semantic model, defined in UML, provided the common knowledge that is shared between different models. The common semantic model was based on different principles, which have been adapted. A modularization concept based on the SO-RA principle was used to describe the machine learning application and other components as independent modules. Also, the idea of skill-based engineering has been adopted. Based on this, each module was modeled to provide its capabilities as skills. Therefore, skills have been considered to be OPC UA programs, that have been typed according to a previously defined skill taxonomy. Furthermore, the application dependencies have been included in the module models in the form of requirements.

The information model in AML has been adapted to provide the application configuration and resolve its requirements to other modules. This model has been converted to an OPC UA server and therefore been made available for the different modules in the system to communicate. By the implementation of an OPC UA client, it has been demonstrated how the information embedded into modules, can be utilized to navigate and resolve server configurations and dependencies in an autonomous way.

However, difficulties arise when relying on information that is modeled not considering any standards. This is especially the case for applications as discussed within this work, where pure software modules, machine learning application and domain-specific tasks are discussed. For instance, skills have been discussed in research for over a decade, but there is no standard to standard available to classify or describe even non-domain-specific skills. The problem that is arising with that, that models from different vendors and developers will consequently be inconsistent with their information description.

Moreover, the exchange and share of common information between different modeling frameworks is an obstacle. In the case of AML and OPC UA, it is not possible to rely on type definition from OPC UA when modeling in AML and vice versa. That is why [52] introduced an ontology, in which information is aggregated and then accessed from OPC UA and AML the same way. This would create a common knowledge base that both modeling frameworks

could rely on and therefore provide more consistency when modeling the same modules in both frameworks.

Therefore in future work, a common base for information has to be created based on which common data each model can rely on. Furthermore, the configuration client has to be adapted and integrated into modules. Also, additional clients have to be implemented that are then provided for the development of new modules. These should enable to access skills and other requirements in a more generic way. Of course, the base for all of this is common and standardized information that is shared and adapted by manufacturers and developers. This will may result from the efforts made around the RAMI.

References

- [1] DIN. Din spec 91345 - reference architecture model industrie 4.0, 2016.
- [2] Federal Ministry for Economic Affairs and Energy, editor. *Structure of the Administration Shell: Continuation of the Development of the Reference Model for Industrie 4.0 Component*. Plattform Industrie 4.0, Berlin, 2016.
- [3] Bundesministerium für Wirtschaft. Relationships between i4.0 components: Composite components and smart production. 2019.
- [4] Somayeh Malakuti, Patrick Zimmermann, Julian Grothoff, Jurgen Bock, Mathias Wiegand, and Andreas Bayha, editors. *Challenges in Skill-based Engineering of Industrial Automation Systems*. IEEE, 2018.
- [5] Julius Pfrommer, Denis Stogl, Kiril Aleksandrov, Victor Schubert, and Björn Hein. Modelling and orchestration of service-based manufacturing system via skills. 2014.
- [6] Miriam Schleipen, Julius Pfrommer, Björn Hein, Andre Mankowski, Denis Stogl, Stefan Navarro, and Jürgen Beyerer. Automationml to describe skills of production plants based on the ppr concept. 2014.
- [7] J. Backhaus and G. Reinhart. Digital description of products, processes and resources for task-oriented programming of assembly systems. *Journal of Intelligent Manufacturing*, 28(8):1787–1800, 2017.
- [8] Veit Hammerstingl and Gunther Reinhart. Fähigkeiten in der montage. 2017.
- [9] Alexander Fay, Xuan Luu Hoang, Christian Diedrich, Martin Dubovy, Christian Eck, Constantin Hildebrandt, André Scholz, Tizian Schröder, and Ralf Wiegand. *Abschlussbericht – SemAnz40: Vorhandene Standards als semantische Basis für die Anwendung von Industrie 4.0*. Helmut-Schmidt-Universität / Universität der Bundeswehr Hamburg, Hamburg, 2018.
- [10] Xuan-Luu Hoang, Constantin Hildebrandt, and Alexander Fay. Product-oriented description of manufacturing resource skills. *IFAC-PapersOnLine*, 51(11):90–95, 2018.
- [11] Stefan Profanter, Ari Breitzkreuz, Markus Rickert, and Alois Knoll. A hardware-agnostic opc ua skill model for robot manipulators and tools. 2019.

-
- [12] Patrick Zimmermann, Etienne Axmann, Benjamin Brandenbourger, Kirill Dorofeev, Andre Mankowski, and Paulo Zanini. Skill-based engineering and control on field-device-level with opc ua. 2019.
- [13] openMOS consortium. openMOS Homepage. <https://www.openmos.eu/>, 2019. [Online; accessed 17-09-2019].
- [14] fortiss. White paper: openmos development platform for plug&produce automation components. 2019.
- [15] P. Danny. An automationml model for plug-and-produce assembly systems. *IEEE*, pages 849–854, 2017.
- [16] fortiss, editor. *openMOS Deliverable: D3.3: Assessment of the current semantic model technologies*. 2017.
- [17] Bundesministerium für Wirtschaft und Energie, editor. *Verwaltungsschale in der konkreten Praxis: Wie definiere ich Teilmodell, beispielhafte Teilmodelle und Interaction zwischen Verwaltungsschalen*. Plattform Industrie 4.0, Berlin, 2019.
- [18] Federal Ministry for Economic Affairs, editor. *Details of the Asset Administration Shell: The exchange of information between partners in the value chain of Industrie 4.0*. Plattform Industrie 4.0, Berlin, 2018.
- [19] Wikipedia contributors. Information model. https://en.wikipedia.org/wiki/Information_model, 2019. [Online; accessed 30-10-2019].
- [20] Anreas Gadatsch. *Datenmodellierung: Einführung in die Entity-Relationship-Modellierung und das Relationenmodell*. Springer Vieweg, 2017.
- [21] Wim Pessemier. Why Semantics Matter. <https://opcconnect.opcfoundation.org/2015/12/why-semantics-matter/>, 2015. [Online; accessed 5-11-2019].
- [22] Bundesministerium für Wirtschaft, editor. *Diskussionspapier I4.0-Sprache: Vokabular, Nachrichtenstruktur und semantische Interaktionsprotokolle der I4.0-Sprache*. Plattform Industrie 4.0, 2018.
- [23] fortiss, editor. *openMos Deliverable: D1.3: Detailed Review and Evaluation of Relevant Standardisation Activities Report*. 2016.
- [24] DIN. Manufacturing processes joint - part 8: Joining by means of adhesives.
- [25] Assembly and handling - handling functions, handling units, terminology, definitions and symbols, 1990.

- [26] OPC Foundation, editor. *OPC UA Amendment 7: Interfaces and AddIns*. OPC UA Specification. 1.04 edition, 2019.
- [27] OPC Foundation, editor. *OPC Unified Architecture Specification Part 1: Overview and Concepts*, volume 1. 1.04 edition, 2017.
- [28] OPC Foundation. UA Companion Specifications. <https://opcfoundation.org/about/opc-technologies/opc-ua/ua-companion-specifications/>, 2019. [Online; accessed 8-09-2019].
- [29] OPC Foundation, editor. *OPC UA for MTConnect Part 1: Device Model*, volume 1 of *OPC UA Companion Specification for MTConnect*. 2.00.00 edition, 2019.
- [30] OPC Foundation. OPC UA Nodeset for Devices. <http://opcfoundation.org/UA/DI/>, 2019.
- [31] OPC Foundation, editor. *OPC Unified Architecture Specification Part 10: Programs*, volume 10 of *OPC UA Specification*. 1.04 edition, 2017.
- [32] OPC Foundation, editor. *OPC Unified Architecture Part 12: Discovery and Global Services*, volume 12 of *OPC UA Specification*. Release 1.04 edition, 2018.
- [33] Darek Kominek. Should I Use OPC UA or MQTT or AMQP? <https://opconnect.opcfoundation.org/2017/10/should-i-use-opc-ua-mqtt-amqp/>, 2017. [Online; accessed 5-11-2019].
- [34] OPC Foundation, editor. *OPC Unified Architecture Specification Part 14: PubSub*, volume 14 of *OPC UA Specification*. 1.04 edition, 2018.
- [35] AutomationML consortium, editor. *Whitepaper AutomationML Part 1: Architecture and general requirements*. 2016.
- [36] AutomationML consortium, editor. *Best Practice Recommendation: AutomationML Container*. AutomationML Best Practice Recommendations. 1.0.0 edition, 2017.
- [37] AutomationML consortium, editor. *Whitepaper AutomationML Communication*. Whitepaper AutomationML. 1.0.0 edition, 2014.
- [38] AutomationML consortium, editor. *AutomationML Best Practice Recommendations: DataVariable*. AutomationML Best Practice Recommendations. 1.0.0 edition, 2017.
- [39] Wikipedia contributors. Machine learning. https://en.wikipedia.org/wiki/Machine_learning, 2019. [Online; accessed 30-10-2019].
- [40] Bundesministerium für Wirtschaft und Energie (BMWi) Industrie. Künstliche Intelligenz und recht im Kontext von Industrie 4.0. *Plattform Industrie 4.0*, 2019.

- [41] Microsoft. What are Azure Machine Learning pipelines? <https://docs.microsoft.com/en-us/azure/machine-learning/service/concept-ml-pipelines>, 2019. [Online; accessed 6-11-2019].
- [42] Julien Kervizic. Overview of the different approaches to putting Machine Learning (ML) models in production. <https://medium.com/analytics-and-data/overview-of-the-different-approaches-to-putting-machinelearning-ml-models-in-production-c699b34abf86>, 2019. [Online; accessed 10-08-2019].
- [43] Michael Chun-Yung Niu. AD Adaso/Adastra Engineering LLC, 1999.
- [44] Katrin Jaacks. Automated circumferential joint assembly in aircraft production: Development and assesment of a production process: Master's thesis in production engineering. 2016.
- [45] Jet-Tek. Hi-Lok Fasteners. <http://www.jet-tek.com/hi-lok-pins/hl939.php>, 2019.
- [46] Volker Briegleb. Roboter im Flugzeugbau: Airbus automatisiert A320-Produktion . <https://www.heise.de/newsticker/meldung/Roboter-im-Flugzeugbau-Airbus-automatisiert-A320-Produktion-4544052.html>, 2019.
- [47] Scikit learn. 3.2.4.3.1. `sklearn.ensemble.RandomForestClassifier`. <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html>, 2019. [Online; accessed 26-07-2019].
- [48] Lorenzo Flückiger and Hans Utz, editors. *Field Tested Service Oriented Robotic Architecture: Case Study*. International Symposium on Artificial Intelligence, Robotics, and Automation in Space (ISAIRAS). 2012.
- [49] Fraunhofer IOSB. AutomationMI - OPC UA Converter. <https://aml2ua.iosb.fraunhofer.de/servlet/is/2112/>, 2017.
- [50] Siemens. Siemens OPC UA Modeling Editor (SiOME) zur Umsetzung von OPC UA Companion Spezifikationen. [https://support.industry.siemens.com/cs/document/109755133/siemens-opc-ua-modeling-editor-\(siome\)-zur-umsetzung-von-opc-ua-companion-spezifikationen?dti=0&lc=de-WW](https://support.industry.siemens.com/cs/document/109755133/siemens-opc-ua-modeling-editor-(siome)-zur-umsetzung-von-opc-ua-companion-spezifikationen?dti=0&lc=de-WW), 2019. [Online; accessed 10-12-2019].
- [51] open62541. open62541. <https://open62541.org/>, 2019. [Online; accessed 11-12-2019].

-
- [52] Andreas Bunte, Oliver Nigemann, and Benno Stein. Integrating owl ontologies for smart services into automationml and opc ua. *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2018.

A. Skill Type Definition in OPC UA

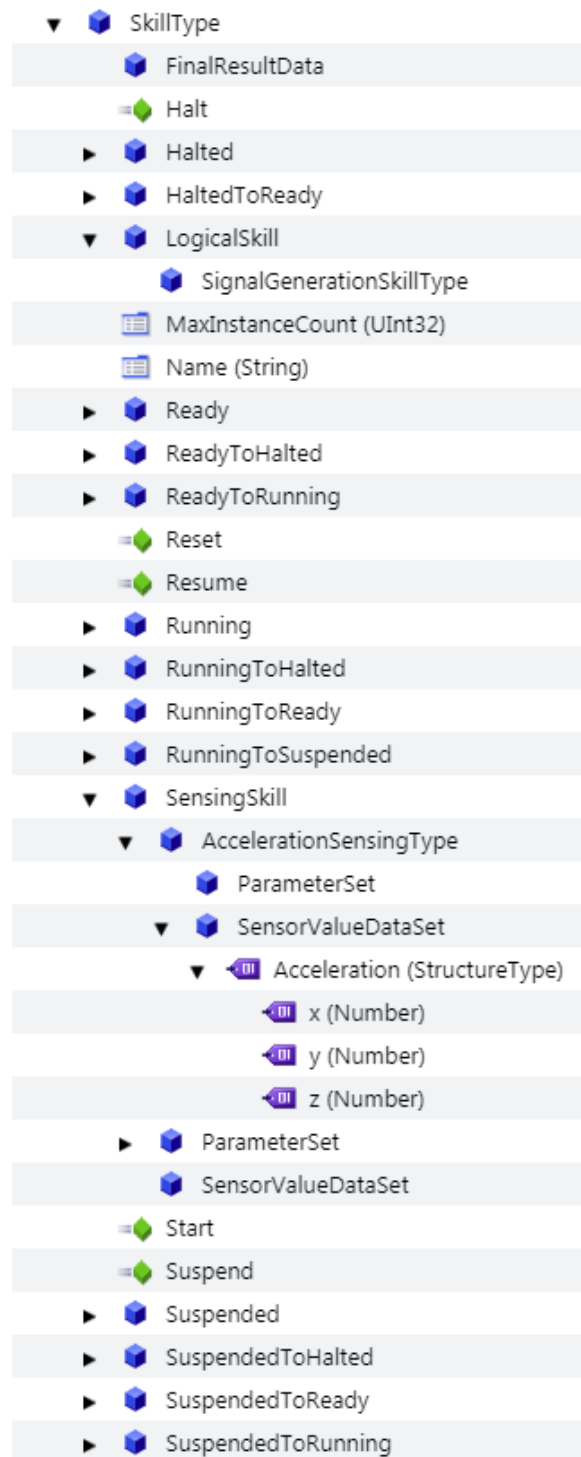


Figure A.1.: Object tree of the skill type definition in OPC UA

B. Break Detection Definition in OPC UA

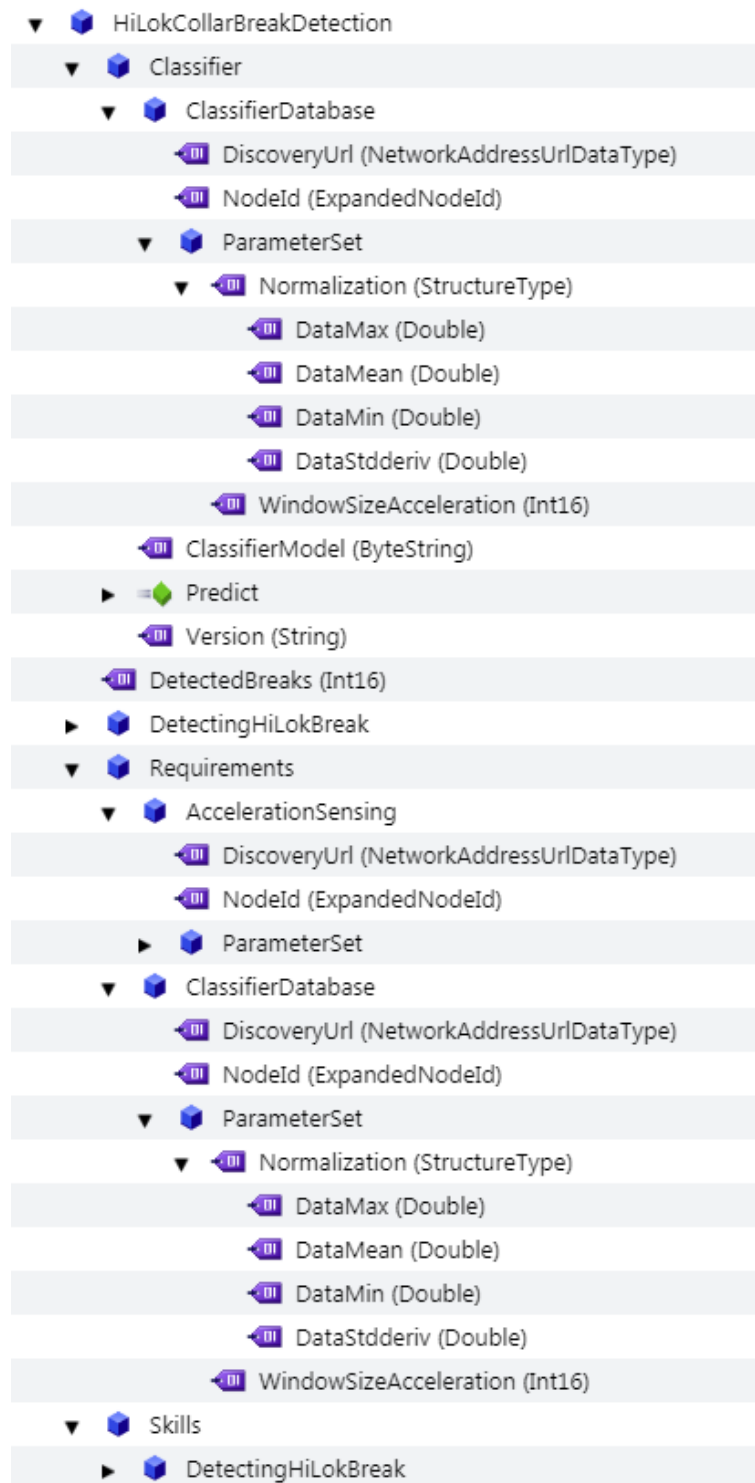


Figure B.1.: Object tree of the skill type definition in OPC UA

C. Configuration Client Program

Listing C.1: Server creation from generated code

```
/*
 * main.c
 *
 * Created on: 26.11.2019
 * Author: jbonas
 */

#include <stdlib.h>
#include <stdio.h>
#include <iostream>
#include <assert.h>

#include "open62541.h"
#include "aml.h"
#include "util.h"

static UA_UInt16 amlIdx;
static UA_UInt16 processIdx;
static UA_Client *amlClient;
static UA_ExpandedNodeId *ua_server_ids;

typedef struct SkillClient{
    UA_Client *client;
    UA_String endpointUrl;
    UA_ExpandedNodeId skillId;
    UA_String skillType;
    UA_NodeId parameterSet;
}SkillClient;

int main(int argc, char *argv[]) {

    // Node IDs for saving the the results from AML address space.
    UA_NodeId bdServerId;
    UA_NodeId bdModuleId;
    UA_NodeId skillServerId;
    UA_NodeId skillId;
    UA_NodeId dependencyModuleId;
    UA_NodeId bdRequirements[2];

    // Create a skill client struct to hold the skill informations
    SkillClient accSensingSkill;

    // Initialize the node IDs
    UA_NodeId_init (&bdServerId);
    UA_NodeId_init (&bdModuleId);
```

```
UA_NodeId_init (&skillServerId);
UA_NodeId_init (&skillId);
UA_NodeId_init (&dependencyModuleId);
UA_NodeId_init (bdRequirements);

// Create and set the AML client configuration.
UA_ClientConfig *config = UA_Client_getConfig(amlClient);
UA_ClientConfig_setDefault (config);

// Connect to the AML server
amlClient = UA_Client_new();
UA_StatusCode retval = UA_Client_connect (amlClient,
                                         "opc.tcp://localhost:16664");
if (retval != UA_STATUSCODE_GOOD) {
    UA_Client_delete(amlClient);
    return EXIT_FAILURE;
}

// Resolve the required namespace indexes from the AML server.
UA_String amlNs = UA_STRING("http://opcfoundation.org/UA/AML/");
UA_String processNs = UA_STRING("http://www.iosb.fraunhofer.de/
                                RivetingProcess_Sensor_UR10.aml");

// Get the index for the AML index with definitions from
// specification.
retval = UA_Client_NamespaceGetIndex(amlClient, &amlNs, &amlIdx);
// Get the index for the previously created AML model namespace
retval = UA_Client_NamespaceGetIndex(amlClient,
                                     &processNs,
                                     &processIdx);

// Check if both namespaces are there, otherwise quit the program.
if(amlIdx == 0 || processIdx == 0){
    printf("Failed to find namespaces");
    return EXIT_FAILURE;
}

printf("Found AML index ns=%d\n", amlIdx);
printf("Found process index ns=%d\n", processIdx);

// Find the server of the break detection application in the AML
// server
getUAServerIdByDiscoveryUrl(amlClient,
                            UA_STRING(
                                "opc.tcp://localhost:5002/
                                RivetingSystem/
                                HiLokCollarBreakDetection/"),
                            &bdServerId);
```

```
// Get the node ID of the break detection module in the AML model.
getContainingModule(amlClient, bdServerId, &bdModuleId);

// Get all requirements of the break detection module.
getModuleSkillRequirements(amlClient, &bdModuleId, bdRequirements, 2);

// Get the requirement connector node ID of the targeted
// dependency/requirement connector.
resolveRequirement(amlClient, &bdRequirements[0], &skillId);

printQualifiedName(amlClient,
                   bdModuleId,
                   UA_STRING("Found Module: "));
printQualifiedName(amlClient,
                   bdRequirements[0],
                   UA_STRING("With requirement: "));

// Get the node ID of the module with the targeted dependency.
getContainingModule(amlClient, skillId, &dependencyModuleId);

// Get the node ID of the OPC UA server object of the targeted module.
getModuleUaServerId(amlClient, dependencyModuleId, &skillServerId);

// Save the endpoint url for the skill
accSensingSkill.endpointUrl =
    readUaServerEndpointUrl(amlClient, skillServerId);
accSensingSkill.skillId.namespaceUri =
    readUaServerNamespace1(amlClient, skillServerId);

printf("At server %.*s\n",
       accSensingSkill.endpointUrl.length,
       accSensingSkill.endpointUrl.data);

// Create a new client to connect to the skill and connect
accSensingSkill.client = UA_Client_new();
config = UA_Client_getConfig(accSensingSkill.client);
UA_ClientConfig_setDefault(config);
retval = UA_Client_connect(accSensingSkill.client,
                          (char*) accSensingSkill.endpointUrl.data);

if (retval != UA_STATUSCODE_GOOD) {
    printf("Failed to connect to server at %.*s\n",
          accSensingSkill.endpointUrl.length,
          accSensingSkill.endpointUrl.data);

    UA_Client_delete(accSensingSkill.client);
    return EXIT_FAILURE;
}
```

```
}  
  
// Resolve the required skill by its type name from the targeted  
// server containing the skill  
getSkillByTypeName(accSensingSkill.client,  
                   UA_STRING("AccelerationSensing"),  
                   &accSensingSkill);  
printQualifiedNames(accSensingSkill.client,  
                   accSensingSkill.skillId.nodeId,  
                   UA_STRING("Skill"));  
  
return EXIT_SUCCESS;  
}
```


Versicherung über die Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §16(5) APSO-TI-BM ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen habe ich unter Angabe der Quellen kenntlich gemacht.

Hamburg, 20. Dezember 2019

Ort, Datum

Unterschrift