Automatic Mapping of AutomationML Files to Ontologies for Graph Queries and Validation

Tom Westermann[1]*, Malte Ramonat[1], Johannes Hujer[1],
Felix Gehlhoff[1], Alexander Fay[2]

[1]Institute of Automation Technology,
Helmut Schmidt University, Hamburg, Germany
[2]Chair of Automation Technology,
Ruhr University, Bochum, Germany

Abstract

AutomationML has seen widespread adoption as an open data exchange format in the automation domain. It is an open and vendor neutral standard based on the extensible markup language XML. However, AutomationML extends XML with additional semantics that limit the applicability of common XML-tools for applications like querying or data validation. This article demonstrates how the transformation of AutomationML into OWL enables new use cases in querying with SPARQL and validation with SHACL. To support this, it provides practitioners with (1) an up-to-date ontology of the concepts defined in the AutomationML standard and (2) a declarative mapping to automatically transform any AutomationML model into RDF triples. A study on examples from the automation domain concludes that transforming AutomationML to OWL opens up new powerful ways for querying and validation that would have been impossible without this transformation.

Keywords: AutomationML, Ontologies, Graph Queries, Validation, Semantic Web, Industry 4.0

1 Introduction

AutomationML is an open and vendor neutral solution for data exchange in the domain of automation engineering. Since its first publication in 2008 [1], it has

---

seen widespread adoption in both the scientific community as well as industry, where it is viewed as a central building block for digital twins [2].

AutomationML is designed to represent complex relationships within engineering projects. It is used to model and exchange data related to the design, implementation, and maintenance of automation systems. The AutomationML data model takes the form of a deeply interconnected graph. However, for easier exchange with other engineering tools, AutomationML uses XML as its underlying format for serialization, which follows a tree structure. In the serialization, references between AutomationML elements of the XML-tree are realized through reference paths and identifiers.

In practice, this representation mechanism offers multiple drawbacks. First, commonly used tools to query and validate XML-Files (like XSD or xQuery) were designed to operate on the hierarchical structure of the data format. They therefore have very rudimentary support for operations involving the additional connections defined by AutomationML. Second, the AutomationML data lacks a formal representation which would allow knowledge inference and further analytical applications [3]. Third, the XML-Schema of AutomationML does not allow for native alignment with other data sources, i.e. it is limited to the vocabulary that AutomationML has to offer. However, this alignment is a pillar of the digital twin [2], making it especially necessary in the engineering domain [4].

In terms of formally representing engineering knowledge, technologies like the Resource Description Framework (RDF) and the Web Ontology Language (OWL) are widely used, as they provide a standardized framework for expressing relationships and semantics [5]. These technologies have been applied to create meta models of cyber-physical systems [6] [7], which allows for the integration of heterogeneous data and resolving semantic heterogeneity [4]. RDF and OWL further allow the use of technologies from the Semantic Web Stack like SPARQL and SHACL, which further enable efficient querying and validation of the created knowledge graphs.

The transformation of AutomationML into an ontology-compliant knowledge graph has been in the focus of the research community for a while [3, 8–11], usually with a focus on data integration.

However, none of the existing approaches have systematically examined the extent to which such a transformation enables new applications. While prior work has primarily emphasized data integration, the potential benefits of representing AutomationML as an ontology—such as the ability to formulate expressive semantic queries or to apply logical validation techniques—remain largely unexplored.

To address this gap, the paper compares the querying capabilities of XML-based tools with those of SPARQL, and examines how validation in XML-based environments contrasts with the possibilities offered by SHACL. In doing so, it introduces an up-to-date AutomationML ontology that covers the full vocabulary of the standard, presents a declarative mapping that enables the straightforward transformation of AutomationML files into RDF triples and provides a software implementation of this mapping to support its practical application. Together, these contributions provide practitioners with both the technical foun-

dations and practical means to exploit ontology-based querying and validation in AutomationML beyond the limits of existing XML-based approaches.

The remainder of the article is structured as follows: Section 2 gives more information on the AutomationML format, how it relates to graph theory and the tools from the Semantic Web Stack. Section 3 describes previous attempts to create an AutomationML ontology and a mapping, while Section 4 presents the newly proposed ontology. Section 5 formalizes the mapping rules to transform AutomationML files to OWL, which are exemplified in Section 6. Section 7 evaluates the approach by comparing querying with XML-based tools and SPARQL, as well as validation with XML-based mechanisms and SHACL. The use cases show that transforming AutomationML to RDF allows for more complex graph queries and validation rules in comparison to typical XML-based approaches.

2 Background

2.1 AutomationML

AutomationML (Automation Markup Language) is an open, XML-based standard for the exchange of engineering data. It builds on established and standardized data exchange formats like Computer Aided Engineering Exchange (CAEX) for plant topology [12], COLLADA for geometry and kinematics [13], and PLCopenXML for control logic [14]. It is commonly used in both the scientific community and industry [1, 15] and has subsequently been standardized in IEC 62714 [16].

Recently, a new normative document - the IEC 62714-2 [17] - was released, which expands the scope of the AutomationML standard by introducing semantics libraries. Including the newly added AttributeTypeLibrary, the CAEX submodel of AutomationML now consists of five main pillars (see Fig. 1) [18]:

- The *InstanceHierarchy* is a hierarchy of InternalElements (IE) and contains the data of a specific project.

- The *SystemUnitClassLibrary* provides SystemUnitClasses (SUC), which describe types of physical plant objects, e.g. a specific type of robot or valve.

- The *RoleClassLibrary* provides RoleClasses (RC), which define abstract roles that a physical or logical plant objects may fill. While SystemUnitClasses usually describe some manufacturer specific systems, RoleClasses model the abstract functions of components.

- The *InterfaceClassLibrary* provides InterfaceClasses (IC) which define a type of interface (e.g. a flange or a signal interface). These interfaces are used to describe physical or logical connections between InternalElements (IE) and SystemUnitClasses (SUC) through InternalLinks.

- The *AttributeTypeLibrary* defines a predefined data structure of reusable AttributeTypes (AT) and can be used to model attributes with values,
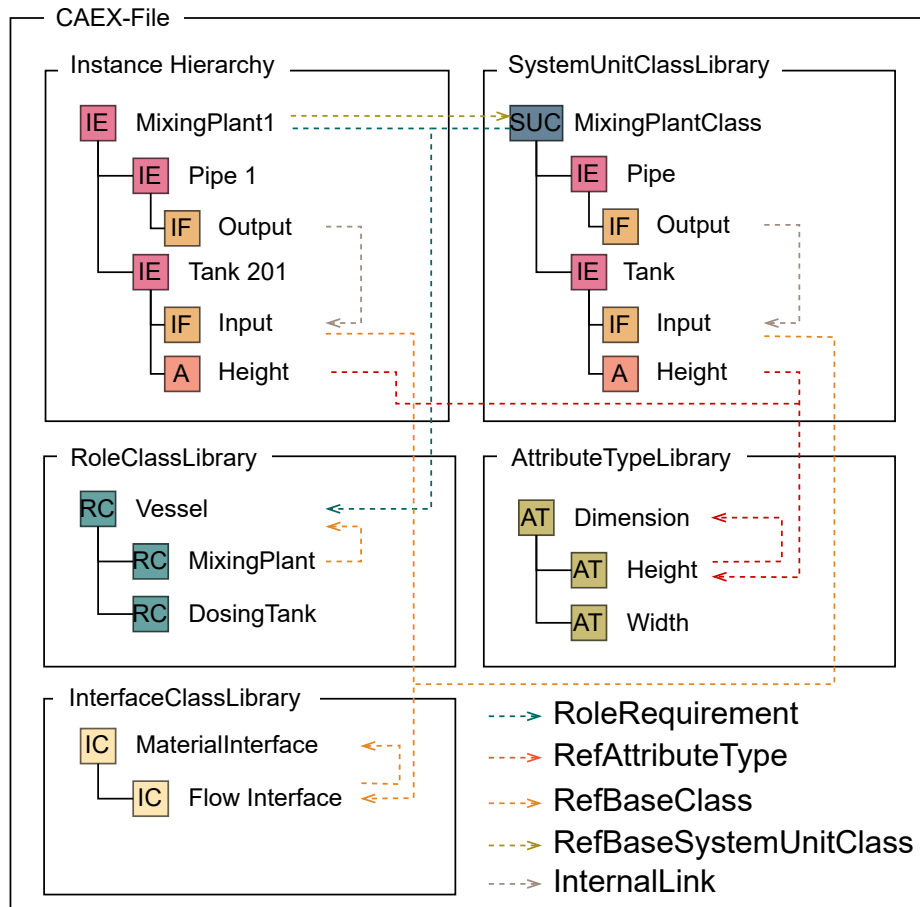
3

Figure 1: The five pillars of CAEX in an AutomationML model alongside some of the most common reference types.

units and descriptions. It allows for the definition of proprietary or standardized vocabularies of attributes, and creates a basis for the semantic interpretability of attributes.

The content of an AutomationML file is structured in the shape of an XML-hierarchy. However, the semantics of this hierarchy can vary depending on the context within the AutomationML structure. For example, in the instance hierarchy, parent-child relationships represent physical or logical containment. In contrast, within class libraries such as RoleClassLibraries, AttributeType-Libraries and InterfaceClassLibraries, the hierarchy is primarily used for structuring and organizing the data. Notably, it does not imply inheritance or specialization by default, but rather provides a logical grouping of related elements for better readability and maintainability.

Beyond the hierarchical organization, AutomationML supports multiple referencing mechanisms to relate elements across the file. Most references – such as those used for class inheritance, interface instantiation, and role or class assignments – are based on a reference path system. These reference paths point to elements within the structured hierarchy using relative or absolute identifiers, allowing connections to be expressed in a way that reflects the modular and nested organization of the XML structure.

In contrast, InternalLinks are used to explicitly define connections between Interface elements by referencing their unique IDs. Since Interfaces typically represent defined interaction points—such as mechanical, electrical, or software interfaces—InternalLinks provide a way to model physical or logical connections between them. This mechanism enables the representation of cross-hierarchy associations that are not captured by structural containment alone.

Listing 1 shows examples of hierarchy-, path- and ID-based references.

AutomationML adopts the CAEX data model and terminology, but some terms acquire context-dependent meanings within AutomationML. Certain AutomationML specific concepts, such as *Groups* or *Facets*, are technically modeled as InternalElement objects with a predefined RoleRequirement, thereby specializing standard CAEX constructs. In other cases, semantics shift depending on the connection type—for example, a *RefBaseSystemUnitClass* link may either indicate an association between an InternalElement and a SystemUnitClass, or represent a master/mirror relationship between two InternalElements.

2.2 AutomationML and Graph Theory

In mathematical terms, a graph G is defined as an ordered pair:

$$G = (V, E) \tag{1}$$

where:

V is a set of vertices (nodes), and

E is a set of edges, where each edge is a pair of distinct vertices:

$$E \subseteq \{\{u, v\} \mid u, v \in V, \ u \neq v\}$$

5

```
<InstanceHierarchy Name="MyInstanceHierarchy">
 <InternalElement Name="MixingPlant1">
  <InternalElement Name="Pipe1">
   <ExternalInterface Name="Output"
    ID="6eea7a40-43fd-4aee-a1d3">
   </ExternalInterface>
  </InternalElement>
  <InternalElement Name="Tank1">
   <ExternalInterface Name="Input"
    ID="ce45d7b3-169d-4be8-9746">
   </ExternalInterface>
   <Attribute Name="Length"
    RefAttributeType="MyAtLib/Dimensions/Length">
   </Attribute>
   <InternalLink Name="Pipe1_to_Tank1"
    RefPartnerSideA="6eea7a40-43fd-4aee-a1d3"
    RefPartnerSideB="ce45d7b3-169d-4be8-9746"/>
  </InternalElement>
 </InternalElement>
</InstanceHierarchy>
```

Listing 1: Excerpt from the InstanceHierarchy depicted in Figure 1 showing all three reference mechanisms: Hierarchy-, path-, and ID-based.

Graphs can take on various shapes and possess different properties, depending on how their vertices and edges are arranged.

XML-based files can be considered as a specific subtype of graphs. Here, the XML-Elements are considered as the nodes, whereas the hierarchical organization of elements forms the edges. If viewed in this way, XML files naturally take the shape of a Tree (s. solid black lines of Figure 2). Trees are a specific type of acyclic, connected graph:

$$T = (V, E) \tag{2}$$

where:

- Each node has a unique path from the root, ensuring a clear hierarchical organization of data.

- Every edge is directed, so for each edge $(u, v) \in E$, u is considered the *parent* and v the *child*.

XML's hierarchical nature ensure that the data can be traversed, queried, and manipulated as a tree, making it an ideal format for structured data representation. XML-based tools for querying and validation heavily rely on this hierarchical parent / child structure.

However, as stated in Section 2.1, AutomationML also allows additional references between nodes besides their hierarchical position in the XML file.
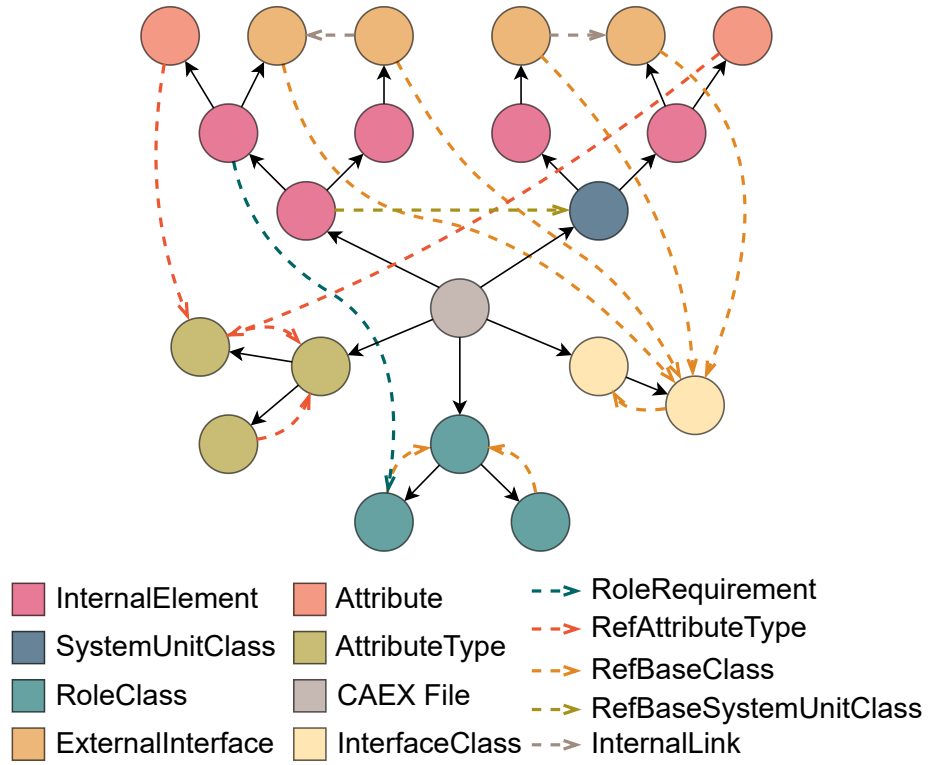
Figure 2: AutomationML file from Figure 1 visualized as a graph. Solid black lines are derived from the XML hierarchy, while dashed lines use ID- or path-based reference mechanisms.

Once the ReferencePaths and InternalLinks are considered, the AutomationML model no longer satisfies the property of having a unique path between every pair of nodes. An example of such a AutomationML graph can be seen in Figure 2. It is therefore no longer a Tree as defined by Definition 2

The data instead forms a *labeled directed multigraph*. It is defined as a triple

$$G = (V, E, L), \tag{3}$$

where:

  V is a set of vertices (or nodes),

  E is a multiset of ordered pairs of *directed edges*. Each edge is represented as (u, v), where $u, v \in V$, with u as the *source* and v as the *target*, and

  L is a function that assigns a label to each edge in E, mapping $L : E \rightarrow \mathcal{L}$, where $\mathcal{L}$ is a set of labels.

In this type of graph, multiple edges can exist between the same nodes. Edges are described further by their labels, that define which type of edge (e.g. RefBaseClass, InternalLink etc.) exists between the nodes.

For the use of XML tools on AutomationML files, this mismatch between XML's tree structure and AutomationML's multigraph structure results in some restrictions.

XML tools for queries (like xPath and xQuery) as well as for validation (like XSD and Schematron) were designed to handle hierarchical data in the shape of Trees (s. Equation 2). They therefore only offer limited support for connections that are not part of the XML tree structure (i.e. all dashed lines in Figure 2).

For data in the shape of labeled directed multigraphs, other data formats and technologies exist, that are able to natively deal with the data's cyclical and interconnected nature. Semantic Web Technologies are commonly used for this task and will be explained in more detail in the following Section.

2.3   Semantic Web Technologies

Semantic Web Technologies are a framework of technologies designed to enable machines to meaningfully interpret and process data. The Resource Description Framework (RDF), its Schema Extension (RDFS) and the Web Ontology Language (OWL) are related technologies used in the context of the Semantic Web Technologies. They are often used together to represent and reason about knowledge in a machine-readable format.

RDF is a framework for representing information about resources on the web. It provides a standardized means to describe resources using subject-predicate-object triples. Each triple represents a statement or a fact about a resource. Mathematically speaking, an RDF-Graph takes the shape of a labeled and directed multigraph, as it allows different types of directed edges.

8

The Web Ontology Language OWL, on the other hand, is a language for defining ontologies. An ontology is a formal description of concepts, relationships, and properties within a certain domain [19]. While RDF provides a foundation for expressing relationships, OWL enhances the expressiveness and formalism of these relationships, enabling the creation of more advanced knowledge representations.

The RDF Mapping Language (RML) is a mapping language defined to express customized mapping rules from heterogeneous data structures and serializations to the RDF data model [20]. It has seen widespread use since its inception and can be applied to structured data sources like CSV, JSON or XML-based formats like AutomationML. Numerous platform independent and open source tools are available to automatically convert data sources to RDF-Triples, given a suitable mapping.

SPARQL is a query language and protocol designed for querying and manipulating data stored in RDF format. It can be used to express patterns and conditions in order to retrieve, construct, analyze, or modify information stored in RDF graphs [21].

The Shapes Constraint Language (SHACL) is a language used for describing and validating the structure and content of data in RDF graphs. SHACL provides a powerful mechanism for defining and enforcing data validation rules in RDF graphs, helping ensure data quality and consistency in semantic web applications [22].

3   Requirements and State of the Art

In the past, numerous attempts have been made to create an AutomationML ontology. The following Section first outlines the requirements that a usable AutomationML ontology needs to fulfill. Afterwards it gives an overview of related works and previous AutomationML ontologies, before comparing previous ontologies to the requirements.

3.1   Requirements

The resulting knowledge graph should be both highly expressive as well as easy to use for people that are already familiar with AutomationML's terminology. This results in a number of requirements, that influence later design decisions. In order to make the ontology easy to understand and use for AutomationML-practitioners, the terminology of the ontology should closely follow the established AutomationML terminology, resulting in Requirement 1. Since queries as well as validation use cases make use of the entire width of information that is stored in an AutomationML file, the ontology should likewise cover the entire vocabulary that AutomationML has to offer (1). However, as stated in Section 2.1, the vocabulary within AutomationML is subject to contextual interpretation based on hierarchical positioning and inter-element relationships. Recognizing these contextual semantics and making them explicitly available

are important for both accurate modeling as well as usability of the resulting knowledge graph (1).

AutomationML is an actively maintained and developed data exchange format. The ontology should therefore consider the most recent version of the AutomationML specification (1).

Most applications of Semantic Web Technologies make heavy use of OWL terminology concepts like class/property hierarchies and property characteristics. Besides following the AutomationML terminology, the ontology should therefore supplement the AutomationML concepts with the logically interpretable vocabulary used by OWL (1).

The terminological definitions in an AutomationML ontology usually consist of two parts: First, the 'core' AutomationML terminology, that defines the AutomationML metamodel (e.g. what a RoleClass is). This is needed to formally describe any AutomationML file. The second part is the individual classification hierarchy (e.g. which RoleClasses there are). These hierarchies change between projects, and can therefore not be modeled overarchingly for all AutomationML files. Since some translation effort for the creation and population of the ontology is therefore project specific, modeling 'just' the core AutomationML terminology in an ontology is not sufficient. Creating a full AutomationML ontology for a project thus necessitates a reusable mapping procedure (1).

Table 1: Requirements for a functional AutomationML ontology.

R1. Class/Property names should follow the AutomationML terminology.

R2. The ontology should cover the entire CAEX vocabulary.

R3. The ontology should allow explicit modeling of context-dependent AutomationML concepts.

R4. The ontology should follow the current AutomationML specification.

R5. Enrich and align concepts from AutomationML with OWL concepts.

R6. A mapping/translation approach for the automatic conversion from AutomationML files to the ontology must be available.

3.2 Prior AutomationML-Ontologies

In the past, multiple attempts to create an AutomationML ontology have been published.

Runde et al. [23] were the first to show the potential of combining CAEX with semantic web technologies. Their approach focuses mostly on represent-

ing CAEX using OWL concepts. However, the approach is lacking with regard to the representation of attributes, and deviates from the terminology that CAEX/AutomationML establish.

Abele et al. [8] cover the most important aspects of the CAEX vocabulary, and even some basic context-dependent concepts like MirrorObjects. However, neither of those are complete as lots of additional information about the entities is not covered.

Grangel-Gonzales et al. [9] focus on the Attribute structure of AutomationML and are therefore lacking coverage of the remaining CAEX vocabulary. AutomationML files are mapped automatically using a XSLT-based tool.

Kovalenko et al. [24] focus on the description of the AutomationML vocabulary, but the authors do not completely cover the full AutomationML schema specification, particularly for data properties.

Kovalenko et al. later present an Ontology of the CAEX-submodel of the AutomationML-standard [3]. This version is essentially a combination of the previous works from [9] and [24]. The ontology provides fairly good coverage of the basic CAEX concepts, with some notable exceptions like constraints. The context dependent concepts (like e.g. Mirror- or MappingObjects) remain missing.

Since the release of all considered ontologies, the AutomationML standard underwent continuous development and has since been released as Version 3.0 in 2022 [17]. All ontologies were released before this date and can therefore not fulfill Requirement 1. Because of this, they lack concepts for the semantic AttributeLibraries introduced in [17].

Table 2: Requirement fulfillment of previous AutomationML Ontologies.

Previous Ontology	[8]	[24]	[9]	[3]
R1: Terminology	◑	◑	◑	◑
R2: Complete Coverage	◑	◑	○	●
R3: Context-dependent Entities	◑	○	○	○
R4: Current AutomationML Version	○	○	○	○
R5: Enrich Model with OWL concepts	●	●	○	●

As Table 2 shows, none of the considered Ontologies fulfill all requirements. However, the ontology presented by Kovalenko et al. [3] achieves good coverage of the basic CAEX concepts. Since the new AutomationML version [17] introduces significant changes to the AutomationML vocabulary as well as the XML serialization, a new ontology needed to be developed. Since many of the basic CAEX concepts did not change, and [3] achieved good coverage in that area, their ontology was used as a starting point.

Section 4 describes the content of this newly developed ontology.

3.3   Prior Mapping Approaches

Multiple publications approached the translation of AutomationML to OWL, using different target ontologies as well as translation technologies.

Out of the ontologies mentioned in Section 3.2, Runde et al. [23], Kovalenko et al. [24] and Grangel-Gonzales et al. [9]mention an automatic translation. Runde et al. [23] propose a semi-automated mapping approach, but do not mention the implementation technology. Kovalenko et al. [24] propose a transformation from AutomationML into OWL using the software framework *Apache Jena*, which is unavailable. Finally, Grangel-Gonzales et al. [9] propose translating AutomationML to RDF using *Krextor*, an XSLT-based framework for converting XML to RDF. However, *Krextor* has not been maintained.

Sabou et al. [25] present the *AutomationML-Analyzer*, which is a tool that showcases the functionalities made possible by the combination of AutomationML-files with semantic web technologies. The authors use the aforementioned AutomationML Ontology presented in [24]. The mapping is performed using a two-step process, in which the AutomationML-files are first converted to Ecore, a domain model of the Eclipse Modeling Framework, and then converted from Ecore to OWL. This conversion procedure is identified as less than ideal by the authors, as they propose a direct mapping as a potential future work.

With regard to mapping AutomationML files to OWL, Hua et al. [26] introduce *AutomationML concept models* for representing OWL complex classes in AutomationML. Additionally, the authors present algorithms for the bidirectional translation between OWL complex classes and their corresponding AutomationML concepts. Their focus, however, is on leveraging the semantics of the AutomationML concept models, not on providing practitioners with a straightforward method of translation between the two representations.

Table 3: Summary of previous approaches to translate AutomationML to OWL.

Approach	Technology	Target Ontology	Available
Runde et al. [23]	-	[23]	○
Kovalenko et al. [24]	Apache Jena	[24]	○
Grangel-Gonzales et al. [9]	Krextor/XSLT	[9]	●
Sabou et al. [25]	Ecore/Java	[24]	●
Hua et al. [26]	Custom/Java	NA	●

The preexisting mapping definitions target differ in their underlying technology as well as their target ontologies. As stated in Section 3.2, these ontologies to not fully meet the outlined requirements, and will therefore be replaced by a new AutomationML ontology. Since the publication of these mapping approaches, *RML* has emerged as the de facto standard declarative mapping language to automatically convert structured data into RDF triples, making it a natural choice for the implementation of a new mapping. Sections 5 and 6 describe the mapping rules for the transformation as well as a detailed example.

4 AutomationML Ontology

To address the requirements outlined in Section 3.1, a new ontology has been developed. This section introduces the AutomationML ontology by outlining its intended scope, presenting its main contents, and explaining the underlying design principles that guided its development.

4.1 Ontology Scope

AutomationML is structured around a clear separation between the metamodel and the model level.

The metamodel is defined by the CAEX schema (IEC 62424) together with AutomationML-specific constraints. It specifies the abstract language of AutomationML: which elements exist (InternalElement, SystemUnitClass, RoleClass, InterfaceClass, Attribute), how they may be related, and which properties they can carry. In ontological terms, this corresponds to the T-Box, i.e., the definition of the vocabulary and its structural rules. The core ontology introduced in this work captures precisely this metamodel.

On the model level, AutomationML distinguishes between two types of content:

- Libraries (e.g., RoleClassLib, SystemUnitClassLib, InterfaceLib, Attribute-TypeLib) define reusable class hierarchies. They extend the metamodel vocabulary with domain- or project-specific categories. In ontology terms, these libraries represent T-Box extensions, since they define new classes and relations, e.g., *MixingTank* as a subclass of *Tank*.

- Instances populate these libraries with concrete elements that describe a system. For example, a project may instantiate a particular *MixingTank* as part of a specific plant. These correspond to A-Box assertions, since they describe individual objects and their relationships.

Because libraries and instances are typically user-defined and vary across projects, it is not feasible to provide a predefined ontology that covers them comprehensively. Instead, the ontology developed here focuses on the core vocabulary of AutomationML (metamodel, T-Box). Section 5 then specifies the rules for extending this core ontology with project-specific libraries (T-Box) and populating it with project instances (A-Box).

As described in Section 3.2, a new AutomationML ontology was designed that aims to to fulfill the Requirements outlined in Section 3.1. The section first gives an overview of the CAEX vocabulary that could be reused (1), before describing the concepts that had to be added to fulfill Requirements 1 and 1.

4.2 Ontology Summary

As discussed in Section 3.2, the AutomationML ontology developed by Kovalenko et al. [3] already provides substantial coverage of the core CAEX terms.

This ontology was therefore chosen as the starting point for the development of the present AutomationML ontology.

Several modifications and extensions were applied to better align the ontology with current requirements:

- Terminology alignment: The terminology was adjusted to closely match the official AutomationML vocabulary (table 1).

- Completeness and specificity: The ontology was extended with previously unmodeled concepts from the CAEX metamodel (table 1) and with context-dependent AutomationML concepts (table 1).

- Version updates: New concepts introduced in the latest AutomationML version, such as AttributeTypes, were incorporated (table 1).

- Documentation and annotations: Additional descriptions and annotations were added to improve clarity and usability.
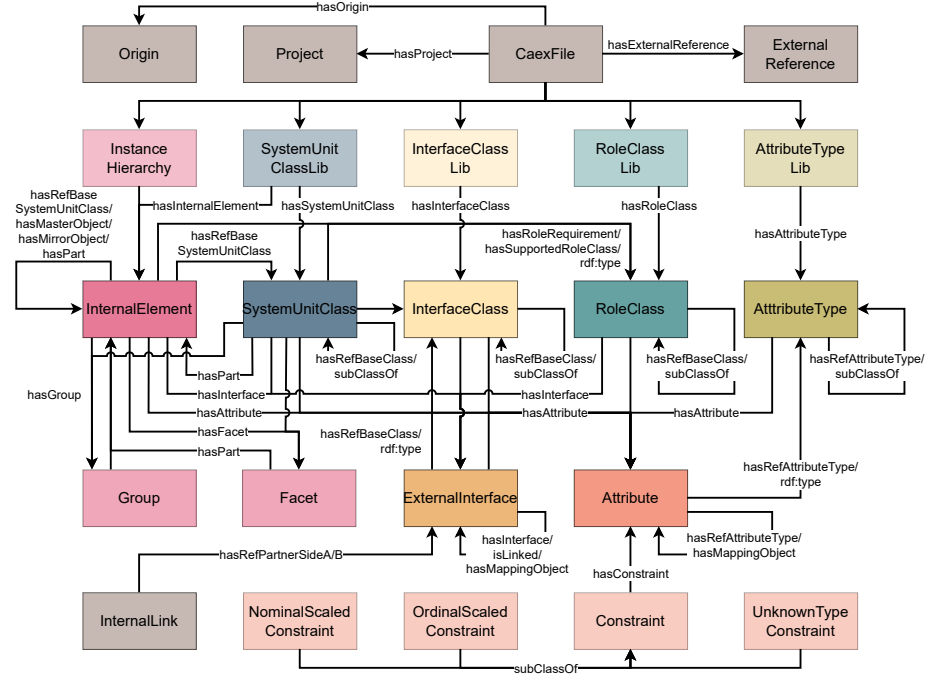
Figure 3: Graphical overview of the classes and most common object properties of the AutomationML-Ontology.

The ontology follows the AutomationML metamodel and retains its original terminology. For instance, a RoleClass continues to reference its parent class

using the *hasRefBaseClass* object property. The corresponding ontology concept, *rdfs:subClassOf*, is applied as a separate object property, in accordance with the mapping rules defined in Section 5.

A graphical overview of the classes and object properties in the AutomationML ontology is shown in Figure 3.

4.3 Design Practices and Reuse

The AutomationML ontology is published under W3ID, which is a permanent identifier, that is administered by the W3C Permanent Identifier Community Group. The ontology is permanently available using the ontology base IRI *https://w3id.org/hsu-aut/AutomationML*. Administration and maintenance of the ontology is supported by the Institute of Automation Technology at Helmut-Schmidt-University. Both the ontology itself as well as its classes and relationships are annotated with metadata documenting their creation process and correct usage.

5 Mapping AutomationML to OWL

While Section 4 described the ontology that covers the AutomationML metamodel, the following section defines how the metamodel is extended by a class hierarchy. With that aim, the following Section provides:

1. An alignment of AutomationML vocabulary with OWL-specific vocabulary (e.g. rdf:type and rdfs:subClassOf),

2. A formal description of the relationships between CAEX, AutomationML, and OWL vocabulary,

3. A formal description of context dependent AutomationML concepts,

The mapping rules in this section - as well as all other rules needed to automatically populate an AutomationML ontology with the data from an AutomationML file - are available in the RDF-Mapping-Language RML.

5.1 Creation of IRIs

Every resource inside an RDF graph has to be identified by an IRI, that distinguishes it from other resources. For AutomationML files, two main ways exist to create these identifiers: The first option relies on the IDs that are provided in the AutomationML file. While this approach is straightforward, IDs are only mandatory for InternalElements and ExternalInterfaces, where AutomationML also uses these IDs internally for references. For all other elements in AutomationML files, the use of IDs is optional and can therefore not reliably be used for referencing. Instead, elements without an ID are referenced through a reference path, which is the second option to create unique IRIs. In AutomationML,

these elements must therefore have a name that is unique to their level in the element hierarchy.

In the RML mapping, this reference approach was mimicked: IDs are used as IRIs where they are mandatory (i.e. for InternalElements and External-Interfaces), while IRI-safe ReferencePaths are used for all remaining elements. References to AttributeTypes, Role-, SystemUnit- and InterfaceClasses take the form of reference paths as well, which makes the creation of ObjectProperties between instances and their respective classes straightforward. All instances are further given non-unique human-readable labels (via *rdfs:label*) based on their names in AutomationML.

5.2   Creation of Classes and Instances

For the mapping process, a decision had to be made which AutomationML concepts would be mapped as classes and as instances. For InternalElements and ExternalInterfaces, this decision is trivial: They are unique individuals, as evidenced by the mandatory use of an ID.

Conceptually, classes in AutomationML are defined slightly different from classes in OWL. On one hand, they function like OWL classes - i.e. as a conceptual definition of a group of objects. On the other hand, classes in Auto-mationML can also function as a template for instances - i.e. they can posses attributes or be composed of multiple subcomponents. In this sense they show characteristics of both OWL classes as well as OWL instances.

OWL 2 DL introduced the concept of *punning* [27]. It allows repurposing the same identifier to represent both a class and an instance. Using this mechanism, AutomationML classes can be represented as OWL classes as well as OWL instances simultaneously. Punning is therefore used extensively in the mapping of RoleClasses, SystemUnitClasses, InterfaceClasses and AttributeTypes.

In the class view, any RoleClass, SystemUnitClass, InterfaceClass, or At-tributeType is converted to an OWL class. Class membership is implied, if instances reference the class using *hasRefBaseSystemUnitClass*, *hasRefBaseClass*, or *hasRefAttributeType* respectively (s. Equation 4).

$$
\begin{aligned}
&\forall (A, B) \\
&(Attribute(A) \land\ hasRefAttributeType\,(A, B)) \\
&\lor (Interface(A) \land\ hasRefBaseClass\,(A, B)) \\
&\lor (InternalElement(A) \\
&\land hasRefBaseSystemUnitClass\,(A, B)) \\
&\implies\ rdf:type\,(A, B)
\end{aligned}
\tag{4}
$$

In the instance view, any RoleClass, SystemUnitClass, InterfaceClass or At-tributeType can be further described with any additional information from the AutomationML file. In this context, the OWL classes *SystemUnitClass*, *Role-Class*, *InterfaceClass*, and *AttributeType* are be treated as metaclasses, that

16

define the structure and behavior of other classes. Classes defined in AutomationML are linked to their respective metaclasses using *rdf:type*.

5.3   Mapping of Roles

As described in Section 4, RoleClasses define the abstract functions of either InternalElements or SystemUnitClasses. AutomationML allows two types of RoleClass assignments:

- *RoleRequirement* can be used to describe roles that an InternalElement or a SystemUnitClass is required to fill in a plant.

- *SupportedRoleClass* can be used if an InternalElement or a SystemUnitClass is capable of filling a role.

These relationships are modeled in two complementary mechanisms: If an InternalElement or SystemUnitClass references a RoleClass via *SupportedRoleClass*, it is considered to be an instance of that class (s. Equation 5). Additionally, the object properties *hasRoleRequirement* and *hasSupportedRoleClass* are assigned to explicitly distinguish between required and supported RoleClasses.

$$
\begin{aligned}
\forall (A, B) \\
(InternalElement(A) \lor SystemUnitClass(A)) \\
\land\ hasSupportedRoleClass\ (A, B) \\
\implies\ rdf:type\ (A, B)
\end{aligned}
\tag{5}
$$

5.4   Mapping of Class Hierarchies

AutomationML allows modeling class hierarchies, in which subclasses can be derived from base classes. Inheritance between classes has to be explicitly modeled using *RefAttributeType* for AttributeTypes or *RefBaseClassPath* for SystemUnit-, Interface-, and RoleClasses.

In the SystemUnitClass-, RoleClass-, InterfaceClass-, and AttributeType-Libraries, this inheritance mechanism results in a class hierarchy using *rdfs:subClassOf* according to Equations 6 and 7.

$$
\begin{aligned}
\forall (A, B) \\
(SystemUnitClass(A) \lor InterfaceClass(A) \\
\lor RoleClass(A)) \land\ hasRefBaseClass\ (A, B) \\
\implies\ rdfs:subClassOf\ (A, B)
\end{aligned}
\tag{6}
$$

$$
\begin{aligned}
\forall (A, B) \\
AttributeType(A) \land\ hasRefAttributeType\ (A, B) \\
\implies\ rdfs:subClassOf\ (A, B)
\end{aligned}
\tag{7}
$$

5.5 Mapping of Internal Structures

Some elements in AutomationML can have an internal structure, i.e. they consist of multiple individual components. For example, both InternalElements and SystemUnitClasses can contain other InternalElements, Interfaces, and Attributes, that can, in turn, have an internal structure as well. In the same way, ExternalInterfaces can contain other ExternalInterfaces (e.g. a voltage interface on USB) and Attributes can contain other Attributes (e.g. x and y values on coordinates). In AutomationML, this relationship can be inferred from the XML-hierarchy in the source document. In the ontology, this implies a PartOf-Relationship. It is therefore modeled using the object property *hasPart*.

5.6 InternalLinks

In AutomationML, an InternalLink describes a directional connection between two interfaces. InternalLinks are crucial because they establish relationships between different components within a system model. The mapping uses this information in two ways: On one hand it follows the AutomationML terminology, in which an InternalLink instance has two RefPartnerSides. However, in comparison to AutomationML, OWL offers more suitable ways of representing links between elements. Additionally, an InternalLink is therefore represented using the :isLinked-Property that links both RefPartnerSides of an InternalLink while maintaining directionality.

$$
\begin{aligned}
\forall (A, B, C) \\
ExternalInterface(A) \wedge ExternalInterface(B) \\
\wedge\, InternalLink(C) \\
\wedge\, hasRefPartnerSideA(C, A) \\
\wedge\, hasRefPartnerSideB(C, B) \\
\implies\, isLinked\,(A, B)
\end{aligned}
\tag{8}
$$

5.7 Mirror-Objects

Depending on the context, the RefBaseSystemUnitPath is used in AutomationML to describe two different relationships. If it connects an InternalElement with a SystemUnitClass, the InternalElement is based on the SystemUnitClass (s. Section 5.2). However, if it connects an InternalElement with another InternalElement, it implies a Mirror-Relationship between the two elements. In this case, the second element follows all changes made to the first one. By introducing the :hasMirrorObject and :hasMasterObject, this connection is made

explicit in the ontology.

$$\forall (A, B)$$
$$InternalElement(A) \land InternalElement(B)$$
$$\land\, hasRefBaseSystemUnitClass(A, B) \qquad (9)$$
$$\implies (hasMasterObject\,(A, B)$$
$$\land\, hasMirrorObject\,(B, A))$$

5.8 Mapping-Objects

In AutomationML, MappingObjects are used to link and correlate elements from different models, ensuring interoperability and data consistency across various engineering domains and tools. If e.g. an InternalElement and its RoleClass have corresponding Attributes or Interfaces with matching names, a MappingObject is implicitly assumed. If the names do not match, the relationship is explicitly stated in the AutomationML file. In the latter case, the mapping is straightforward. In the former case, the mapping information has to be extracted from the data according to Equations 10 / 11.

$$\forall (A, B, C, D, E, F)$$
$$InternalElement(A) \land RoleClass(B)$$
$$\land\, Attribute(C) \land Attribute(D)$$
$$\land\, hasAttribute(A, C) \land hasAttribute(B, D) \qquad (10)$$
$$\land\, hasName(C, E) \land hasName(D, F)$$
$$\land\, hasRoleRequirement(A, B) \land\ Equal(E, F)$$
$$\implies hasMappingObject\,(C, D)$$

$$\forall (A, B, C, D, E, F)$$
$$InternalElement(A) \land RoleClass(B)$$
$$\land\, Interface(C) \land Interface(D)$$
$$\land\, hasInterface(A, C) \land hasInterface(B, D) \qquad (11)$$
$$\land\, hasName(C, E) \land hasName(D, F)$$
$$\land\, hasRoleRequirement(A, B) \land\ Equal(E, F)$$
$$\implies hasMappingObject\,(C, D)$$

5.9 Facets and Groups

Both Facets and Groups are not explicitly part of the original CAEX-Vocabulary. Instead, they are modeled as InternalElements that have a RoleRequirement of Group or Facet from the AutomationMLBaseRoleClassLibrary. Their membership in the class Group/Facet is therefore inferred according to Equations 12 and 13.

$$\forall(A, B)$$
$$(InternalElement(A) \lor SystemUnitClass(A))$$
$$\land InternalElement(B) \land hasPart(A, B) \qquad (12)$$
$$\land hasRoleRequirement(B, Facet)$$
$$\implies Facet(B) \land hasFacet(A, B)$$

$$\forall(A, B)$$
$$(InternalElement(A) \lor SystemUnitClass(A))$$
$$\land InternalElement(B) \land hasPart(A, B) \qquad (13)$$
$$\land hasRoleRequirement(B, Group)$$
$$\implies Group(B) \land : hasGroup(A, B)$$

5.10 Inheritance of Subcomponents

In AutomationML, the internal structure of a class is inherited from a parent class to the child class. In the AutomationML model, this relationship is not explicitly modeled, but can instead be inferred through the inheritance relationship. It is additionally possible to override inherited objects. For the mapping to RDF, this means that an internal class structure is inherited from the parent class if the inherited structure was not overridden (s. Equation 14).

$$\forall(A, B, C, D, E)$$
$$RoleClass(A) \land RoleClass(B)$$
$$\land hasRefBaseClass(A, B)$$
$$\land hasAttribute(B, C) \land hasName(C, D) \qquad (14)$$
$$\land \neg(\exists E\, (hasAttribute(A, E) \land hasName(E, D)))$$
$$\implies hasAttribute(A, C)$$

5.11 Mapping Process and Usage

Fully mapping an AutomationML file to RDF triples is a two-step process. In the first step, the provided RML mapping is executed. This populates the knowledge graph with all RDF triples corresponding to the contents of the file. In a second step, a set of SPARQL-Queries is executed, which adds additional triples, that cannot otherwise be created by means of pure RML mapping. This information includes the inheritance of Attributes and Interfaces from a parent element, as well as creating explicit MappingObjects of Attributes and Interfaces.

A mapping application is available*, that allows automatic mapping of a file from the command line. It includes both the RML mapping as well as the SPARQL queries.

Alternatively, the mapping and the SPARQL queries are individually available on GitHub†. The mapping can be executed using any RML-compliant mapping tool, while the SPARQL queries can be executed using any compliant SPARQL processing tool.

6 Mapping Example

To illustrate the mapping rules from the previous section, a Four-Tank Mixing System will be used as a running example. Further models and test cases are available on Github.

6.1 System Description: Four-Tank Mixing System

In the main use case, a laboratory scale process plant in the Institute of Automation Technology at Helmut-Schmidt-University is considered. It is used for fluid mixing and consists of three dosing tanks, a mixing tank, eleven valves and two pumps. Pipes connect the various components for fluid transport. A Piping and Instrumentation Diagram (P&ID) of the system can be seen in Figure 4. The system's topology was described in AutomationML according to the best practice recommendations by the AutomationML society.

The remainder of this section shows how the main concepts of AutomationML are translated to RDF-Triples using excerpts from the AutomationML file of this system.

6.2 Creation of IRIs

As described in Section 5.1, the creation of IRIs for the ontology follows the reference mechanisms that AutomationML is using internally. Therefore, IDs are only used for the IRI creation of InternalElements and ExternalInterface. For all other elements, unique paths are created using the "Name"-Attribute. This mimics AutomationMLs approach to references.

Listing 2 and Figure 5 show an example of this logic. While the InternalElement *B201* is assigned an IRI from its ID, both the Attribute *Length* as well as the AttributeType *Length* use their paths as an IRI.

6.3 Creation of Classes and Instances

The following section provides an example of the creation of classes and instances. Listing 3 shows an example of an InternalElement, that has all major

---

*https://github.com/hsu-aut/aml2owl
†https://github.com/tomwestermann/AutomationML2Ontology

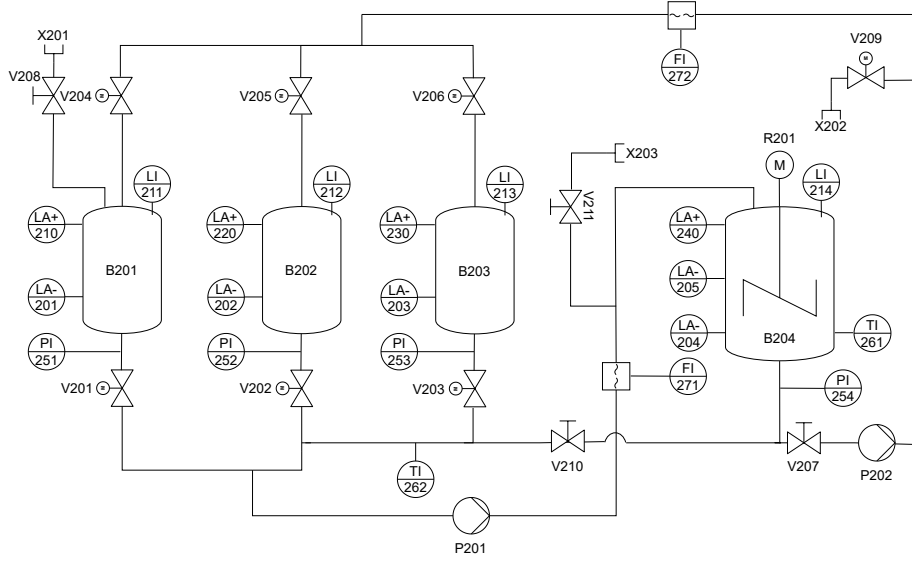Figure 4: P&ID of the Fluid Mixing System.

```
<InstanceHierarchy Name="MyIH">
 <InternalElement Name="B201"
  ID="44806a23-d2bd-45d2-8344">
  <Attribute Name="Length"
   RefAttributeType="MyAtLib/Length">
  </Attribute>
 </InternalElement>
</InstanceHierarchy>
```

Listing 2: Example of an InternalElement with an Attribute "Length".

elements that AutomationML uses: A SystemUnitClass, an Attribute, an ExternalInterface as well as two RoleClasses - one as a SupportedRoleClass and one as a RoleRequirement.

Figure 6 shows the corresponding relationship between OWL instances and classes from Listing 3: For example, the InternalElement *B201* is:

- an instance of the SystemUnitClass *Vessel*, since it is its RefBaseSystemUnitClass,

- an instance of the RoleClass *Vessels*, since it supports that RoleClass,

- not an instance of the RoleClass *DosingTank*, since having a RoleRequirement does not imply class membership.

The Attribute *Length* and the Interface *Input* are also instances of their respective AttributeTypes and BaseClasses. The RDF representation of the Auto-
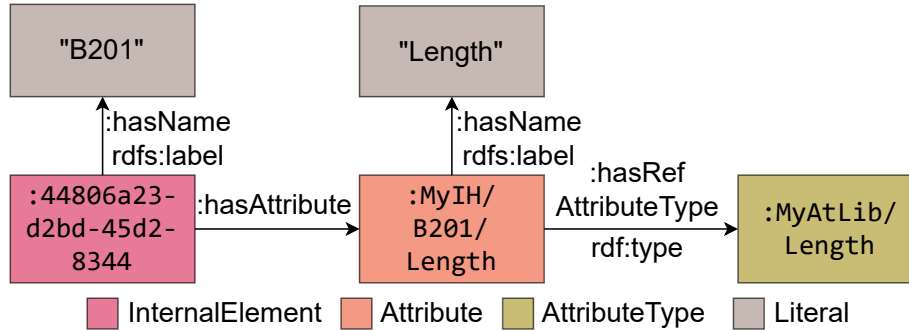
Figure 5: RDF representation of the AutomationML content from Listing 2.

```
<InternalElement Name="B201"
 ID="44806a23-d2bd-45d2-8344"
 RefBaseSystemUnitPath="MySucLib/Vessel">
<Attribute Name="Length"
 RefAttributeType="MyAtLib/Length">
</Attribute>
<ExternalInterface Name="Input"
 ID="ce45d7b3-169d-4be8-9746"
 RefBaseClassPath="MyIcLib/Port">
</ExternalInterface>
<SupportedRoleClass
 RefRoleClassPath="MyRcLib/Vessel" />
<RoleRequirements
 RefBaseRoleClassPath="MyRcLib/DosingTank" />
</RoleRequirements>
</InternalElement>
```

Listing 3: InternalElement with all basic AutomationML elements.

mationML data additionally connects the instances to their classes using the AutomationML terminology.

6.4   Mapping of Class Hierarchies

The previous section showed how an InternalElement (or its Attributes and Interfaces) can relate to OWL classes that are derived from AutomationML libraries.

Since AutomationML allows inheritance, these OWL classes become part of a class hierarchy. AutomationML additionally provides further meta information about these classes (e.g. default values, semantic references, provenance, etc.). To adequately capture this information, punning, i.e. the simultaneous use of the same IRI as a class and an instance, is used.
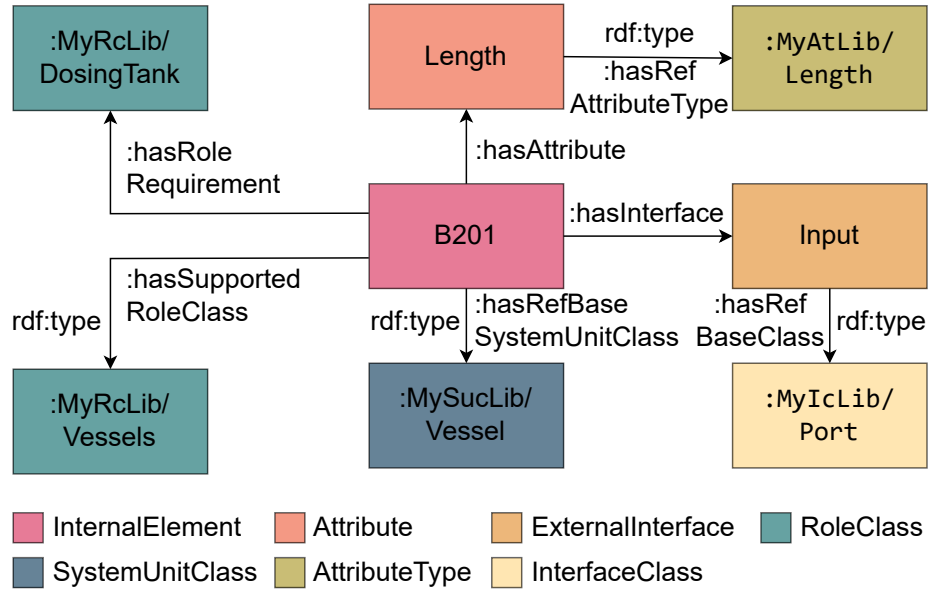
Figure 6: Excerpt of resulting RDF-Triples for Classes and Instances from Listing 3.

The example in Listing 4 and Figure 7 shows how the AttributeType *Length* from the previous example is mapped and embedded into an OWL class hierarchy. For AttributeTypes, the *rdfs:subClassOf*-relationship can be inferred from the *RefAttributeType* in AutomationML.

```
<AttributeTypeLib Name="MyAtLib">
 <AttributeType
  Name="Dimensions">
 <AttributeType Name="Length"
  RefAttributeType="MyAtLib/Dimensions" />
 </AttributeType>
</AttributeTypeLib>
```

Listing 4: Excerpt from the AttributeLibrary.

OWL Class hierarchies are created from all AutomationML libraries, i.e. also from the RoleClass-, SystemUnitClass-, and InterfaceClassLibraries. The mapping approach is similar for all other libraries.

6.5 Mapping of InternalLinks

Listing 5 shows an example of two InternalElements, whose interfaces are connected by an InternalLink. Figure 8 shows both mechanisms of linking their
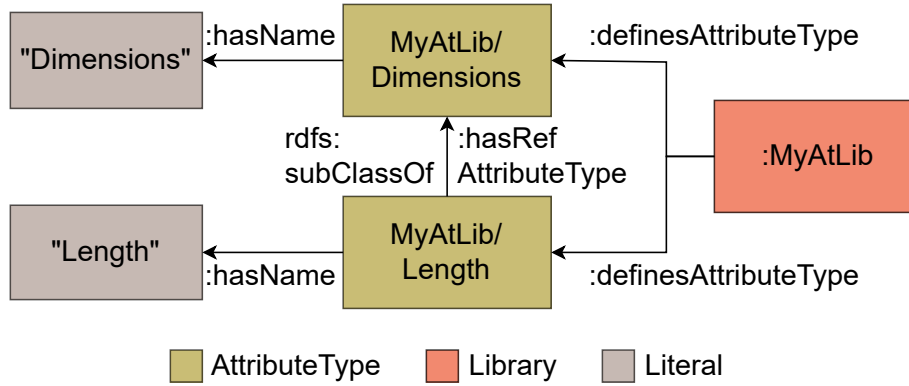
Figure 7: Excerpt of resulting RDF-Triples from Listing 4.

Interfaces. On one hand, they are connected in a way that very closely follows the AutomationML-vocabulary with an instance of the class InternalLink that connects the Interfaces through the Object Property :hasRefPartnerSideA/B. On the other hand, they are connected directly through the object property :isLinked. This makes traversing the connections between InternalElements (e.g. for queries) more intuitive, while preserving the terminology most familiar to AutomationML practitioners.

```
<InstanceHierarchy Name="MyIH">
 <InternalElement Name="Pipe"
  ID="a20e3eac-9ec0-41f1-852a">
  <ExternalInterface Name="Output"
   ID="6eea7a40-43fd-4aee-a1d3-">
  </ExternalInterface>
  <InternalLink Name="Pipe_B201"
   RefPartnerSideA="6eea7a40-43fd-4aee-a1d3"
   RefPartnerSideB="ce45d7b3-169d-4be8-9746"/>
 </InternalElement>
 <InternalElement Name="B201"
  ID="44806a23-d2bd-45d2-8344">
  <ExternalInterface Name="Input"
   ID="ce45d7b3-169d-4be8-9746">
  </ExternalInterface>
</InstanceHierarchy>
```
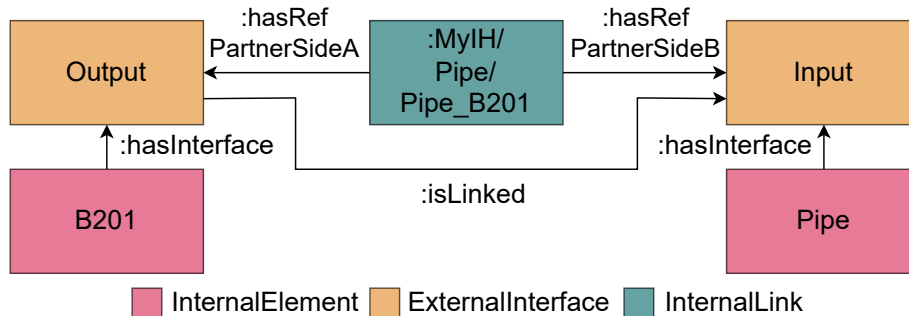
Listing 5: Example of an InternalLink that connects a Pipe with a Tank.

Figure 8: Resulting RDF-Graph from Listing 5 of two InternalElements with two linked Interfaces.

7 Application

A reliable and reproducible way of translating AutomationML-Files to RDF-Triples opens up a plethora of new applications using the different technologies from the semantic web stack (s. Section 2.3). In this section, two such applications, namely querying and validation, and their advantages will be described. The first application describes the use of the query language SPARQL to construct complex graph queries. The capabilities of SPARQL are compared to xPath and xQuery, which are considered the most common query languages that operate directly on the XML representation of AutomationML. The second application describes an automatic validation of an AutomationML file by using SHACL to formulate validation patterns. SHACL's capabilities are then compared to XSD and Schematron, two common approaches to validate the content of XML files.

7.1 Complex Graph Queries

As stated in Section 2.2, the AutomationML model describes a labeled and directed multigraph, that is serialized as an XML file in the shape of a Tree.

Common tools to perform queries on XML data (like xPath or xQuery) were designed to navigate tree structures, and are therefore lacking mechanisms to handle the additional edges in the graph created by ReferencePaths and InternalLinks.

However, many industrial applications could significantly benefit from the possibility to query the full spectrum of information provided by the Automation-ML-file.

To illustrate the advantages of such graph queries, we revisit the Four-Tank-Mixing System described in Section 6.1. We will compare the extraction of information using different approaches: xPath and xQuery, which operate directly on the XML logic underlying the AutomationML file, and SPARQL, which queries an RDF graph generated from the AutomationML file.

Sets of search queries of varying complexity will be considered to highlight the strengths and specific limitations of each approach, demonstrating the potential benefits of employing graph queries in relevant scenarios.

Use Case 1: Conditional Element-Selection

In the first Use Case, the simple selection of elements based on some selection criteria was considered. Three different scenarios were designed. Each scenario differs in the degree of separation in the transformed graph between the element A that should be selected and the element X at which the selection criteria is applied.

1. *Scenario 1 - First Degree Selection*:

    Select all InternalElements A that have a certain RoleRequirement B:

    $InternalElement(A)$
    $\land\, hasRoleRequirement(A, B)$

2. *Scenario 2 - Second Degree Selection*:

    Select all InternalElements A that have a RoleRequirement B, where B references a RoleClass C as a RefBaseClass:

    $InternalElement(A)$
    $\land\, hasRoleRequirement(A, B)$
    $\land\, hasRefBaseClass(B, C)$

3. *Scenario 3 - nth Degree Selection*:

    Select all InternalElements A that have a RoleRequirement B, where B references a RoleClass C as a RefBaseClass through transitive closure.

    $InternalElement(A)$
    $\land\, hasRoleRequirement(A, B)$
    $\land\, hasRefBaseClass * (B, C)$

Example queries of these three scenarios were applied to the the Mixing System: All InternalElements should be selected that have the RoleRequirement of being 1) a Vessel, 2) a Plant Equipment (or any direct descendant) or 3) a Resource (or any transitive descendant). For the example in Figure 9, the first scenario should return just *B201*, while the queries from the other scenarios should return both InternalElements.

For all three of these Scenarios, queries were written in each of the three query languages.

The queries regarding the first scenario were easily achievable in all three languages (s. Listing 6).

However, both the second and third scenario make use of the AutomationML reference paths. The query in Scenario 2 not only references the RoleRequirement of InternalElement A, but also the RoleRequirement's RefBaseClass. In
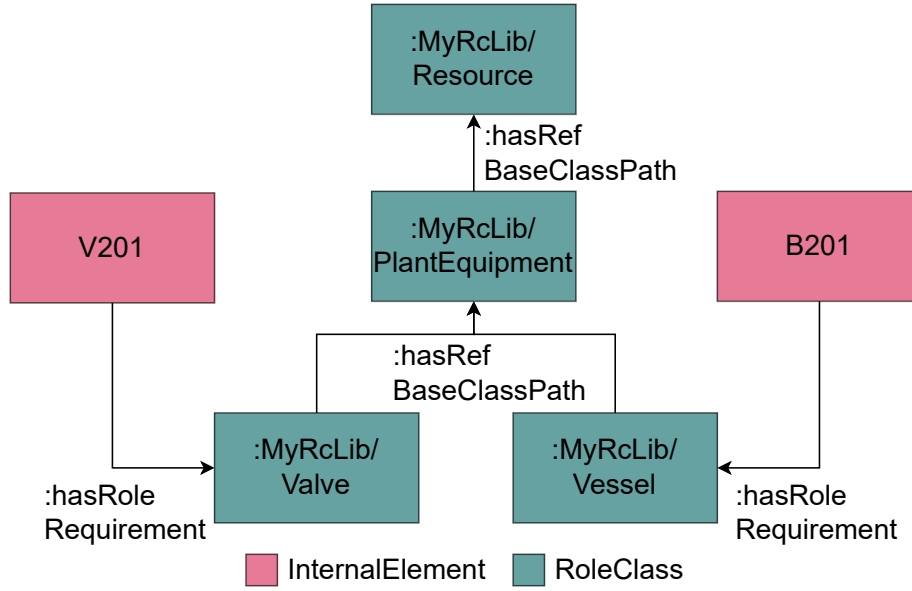
Figure 9: Two InternalElements with a RoleClass Hierarchy.

the XML tree, this information is located at a different branch of the tree at the RoleRequirement node and thus necessitates a traversal of the graph that does not follow the hierarchical structure of the XML file. Scenario 3 further extends this by combining the filter condition with an open-ended transitive closure statement, i.e. a sequence of RefBaseClassPaths of indeterminate length.

Both xPath and xQuery operate directly on the XML-tree, and have therefore no mechanisms to handle ReferencePaths. xQuery offers some mechanisms to circumvent this limitation: Technically, the second scenario could still be solved using xQuery. The approach includes reconstructing the reference path of all RoleClasses, iterating over them, and individually comparing them to the desired value (s. Listing 7). However, this approach is very limited in scope, e.g. capturing the open-ended transitive closure statement from Query 3 in this way is not possible. Using SPARQL, due to the support of Property Paths, all three queries can be answered in a much more compact and arguably more intuitive form (s. Listing 7).

Use Case 2: Automatic Model Creation

In the second use case, a more complex query will be formulated, with the aim of automatically producing a Material Flow Graph of the Four-Tank-Mixing System. The automatic generation of material flow graphs or other causal graphs has many applications in areas like the automatic generation of simulation models [28] [29], or the contextualization and diagnosis of data anomalies [30] and alarm floods [31] [32] [33].

xPath:

```
//InternalElement[RoleRequirements[
  @RefBaseRoleClassPath='MyRcLib/Vessel']]
```

xQuery:

```
for $ie in //InternalElement
  where $ie/RoleRequirements/
    @RefBaseRoleClassPath = "MyRcLib/Vessel"
  return $ie
```

SPARQL:

```
SELECT ?ie
WHERE {
  ?ie rdf:type aml:InternalElement .
  ?ie aml:hasRoleRequirement aml:MyRcLib/Vessel.}
```

Listing 6: xPath, xQuery and SPARQL Query for Scenario 1.

As in the previous Use Case, two scenarios with different levels of complexity will be covered:

1. *Scenario 1 - Bidirectional FlowGraph*:

   Create a flow graph assuming bidirectional flow.

2. *Scenario 2 - Valve-state dependent FlowGraph*:

   Create a valve-state dependent directional flow graph.

In the first scenario, the content of the AutomationML-file should be used to create a material flow graph of all possible paths through the system. In the second scenario, the query was extended to include information about the state of valves, as they enable or disable product flow in parts of the system.

Therefore, both queries build upon the same basic graph pattern, but extend it with additional conditions that have to be met.

In this use case, it is assumed that the material flow interface in AutomationML was modeled using the InterfaceClass *Port* or interfaces that inherit from this class. The *Port*-InferfaceClass is defined in the AutomationMLBaseInterfaceLibrary. In addition, any valves in the model can have a boolean ValveState-Attribute that indicates, whether the valve is open or closed.

Since it does not support multi-step-patterns, xPath was unable to create a material flow graph for either of the three scenarios. Using xQuery, a Flow Graph for the first Scenario could be created. However, adding additional checks and constraints quickly results in very deeply nested loops over all possible connections. While technically possible, the queries this approach produced were opaque and error-prone.

<div align="center">**xQuery:**</div>

```
let $roleClasses :=
  //RoleClass[@RefBaseClassPath='PlantEquipment']
for $ie in //InternalElement
 where some $rc in $roleClasses satisfies
  $ie/RoleRequirements/@RefBaseRoleClassPath =
   string-join($rc/ancestor-or-self::*/@Name,'/')
 return $ie
```

<div align="center">**SPARQL:**</div>

```
SELECT ?ie
WHERE {
  ?ie rdf:type aml:InternalElement .
  ?ie aml:hasRoleRequirement/(rdfs:subClassOf)*
      aml:PlantEquipment .}
```

<div align="center">Listing 7: xQuery for Scenario 2 and SPARQL Query for Scenario 3.</div>

SPARQL, on the other hand, offers the possibility to define repeating patterns inside the graph that should be traversed. Due to these features, all queries can be answered using a more straightforward and compact SPARQL-Query.

A SPARQL query that solves the second scenario can be found in Listing 8.

```
construct{?s aml:flows ?o.} where {
    ?s a aml:InternalElement.
    ?o a aml:InternalElement.
    ?s   aml:hasInterface ?i1.
    ?o   aml:hasInterface ?i2.
    ?i1  aml:isLinkedTo   ?i2.
    OPTIONAL {
      ?s   aml:hasAttribute ?vs.
      ?vs  aml:hasRefAttributeType aml:ValveState.
      ?vs  aml:hasAttributeValue ?state.
    }
    Filter(!bound(?state) || ?state = 'true').
}
```

<div align="center">Listing 8: SPARQL query, that creates a material flow graph while considering valve states.</div>

Comparison of xPath, xQuery, SPARQL

The three query languages show differences in their ability to extract information from the AutomationML model. All three query languages are able to perform basic queries that encompass selecting elements based on filtering conditions.
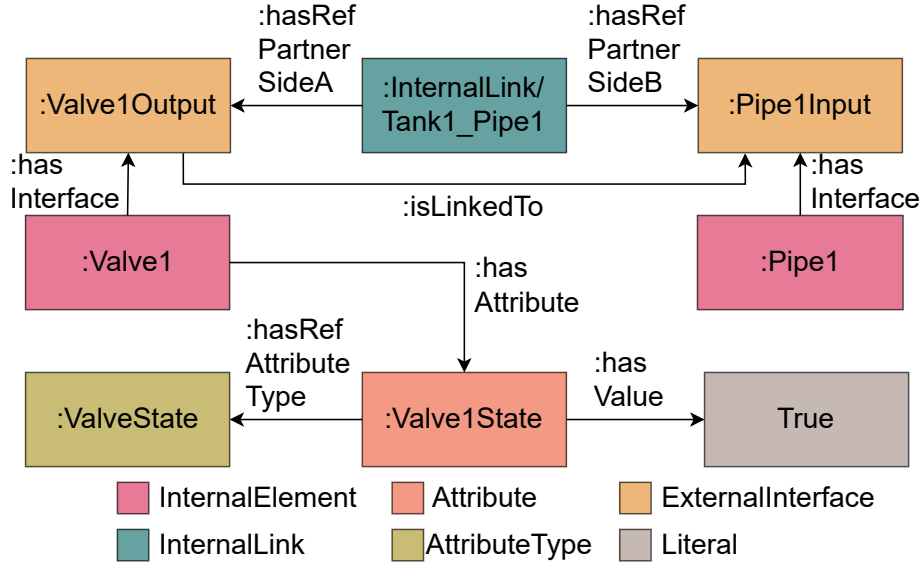
Figure 10: Example of a valve that is connected to a pipe. The valve has a state, that defines whether it is open or not.

However, once the AutomationML-ReferencePaths become involved, difficulties emerge in xPath and xQuery.

For this comparison, we refer to the different categories of filtering conditions from Section 7.1: In First Degree Filtering and Comparisons, the filter condition applies to the direct attributes that an element has. These tasks also encompass all queries in which only the ReferencePath is of interest, and the information at the destination of the path is irrelevant. All query languages were able to fulfill these tasks. In Second-Order Filtering, the filter condition applies to an element that the first element references via a reference path. For these queries, xPath is unable to retrieve the targeted elements. xQuery, on the other hand, is technically able to answer those queries through a combination of loops and string reconstructions and comparisons. However, the resulting queries are complicated and not robust.

In n-th order filtering, the filter condition applies to an element that the first element references through a chain of two or more subsequent reference paths. This case also includes use cases like multilevel inheritance or complex path patterns. In these cases, xQuery is not suitable to create functional queries. All of these shortcomings of xPath and xQuery to query AutomationML files stem from the mismatch between the XML tree structure that both languages were designed for, and the labeled directed multigraph that the contents of an AutomationML file actually represent. Since it is a dedicated graph query language, SPARQL is able to handle such use cases, while also offering explicit support for more advanced and graph specific features like property paths and
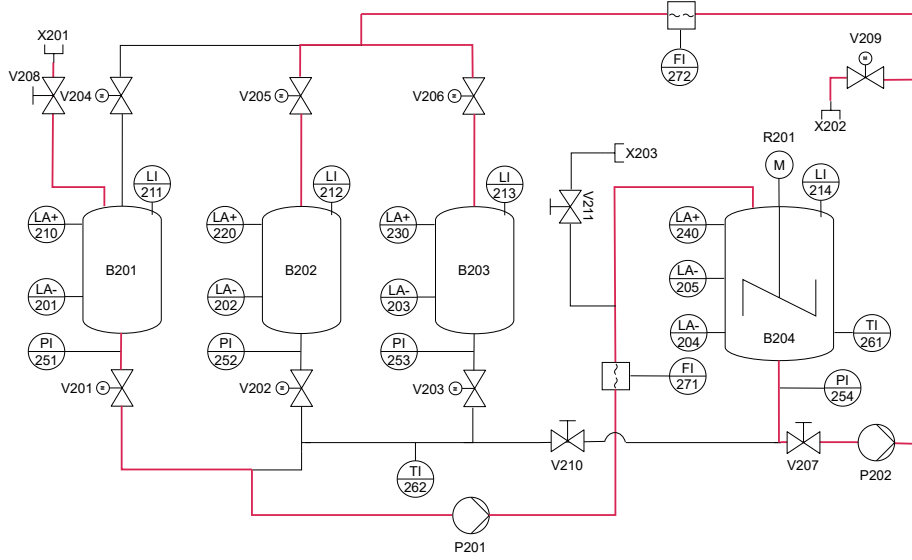
31

Figure 11: Illustration of the mixing module's resulting flow graph (shown in red), if some of the valves are closed.

complex graph patterns.

In summary, the comparison between the query languages shows that xPath and xQuery are at best sufficient for simple queries and information retrieval. If the scenarios become more complex, SPARQL offers greater functionalities and ease of use.

7.2   Model Validation

Besides the potential for querying, representing an AutomationML file in the form of an ontology also offers possibilities for validation of the file's content. A case study was designed to compare three common validation languages - XSD, Schematron and SHACL - for their suitability for AutomationML data.

XML Schema, usually abbreviated XSD, operates directly on the XML files. It is commonly used to validate the syntactic correctness of an XML file. Schematron is an XML schema language used for rule-based validation of XML documents. In comparison to XSD, Schematron offers a more flexible, rule-based approach. SHACL on the other hand is a language for validating RDF data. It provides mechanisms to define constraints on RDF graphs, ensuring that data conforms to specific shapes and patterns.

Use Case: Library-Level Rules

In this use case, the validation concerns the correct usage of elements that were imported from user defined libraries. These libraries are commonly accompanied

by textual specifications that define the proper usage of the imported elements. As those specifications lack a formal or machine interpretable definition, their correct usage needs to be manually checked by domain experts. For this use case, the *AutomationML Application Recommendation Automation Project Configuration* (AR APC) was considered. It is a framework that describes guidelines for the configuration of automation projects. [34]

For this example, the SensorPort-RoleClass was considered (s. Figure 12). It is a RoleClass used to model the connection between devices for the transfer of sensor signals. As such, it can contain an Interface that connects to SensorPortsInterfaces. The AR APC defines two constraints on this class:

- *Scenario 1 - Cardinality Constraint:*

  All Instances of the class can have at most one SensorPortInterface.

- *Scenario 2 - Connection Constraint:*

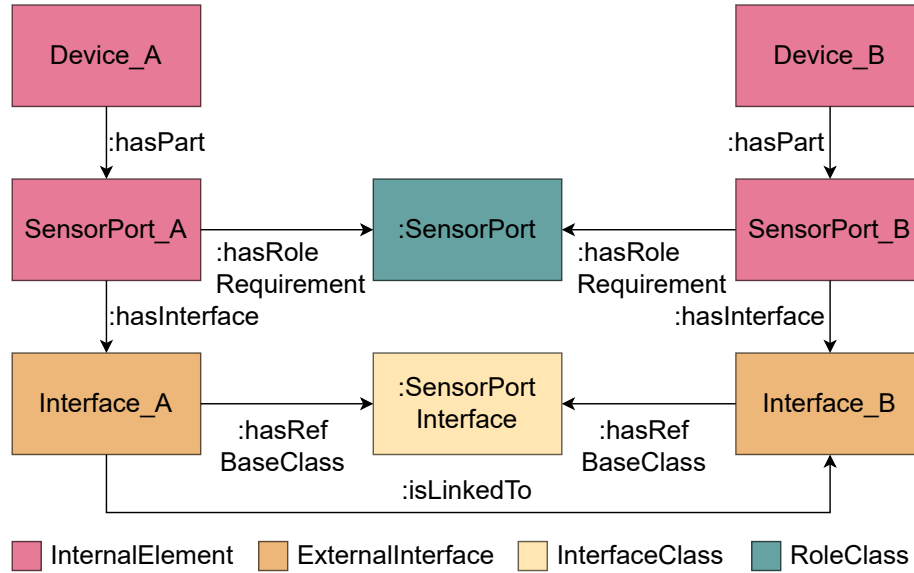 SensorPort-Interfaces can only be connected to other SensorPort-Interfaces.

Figure 12: Example of a valid connection of two Devices using the SensorPort element.

Comparison of XSD, Schematron and SHACL

Both scenarios constitute conditional constraints, i.e. they check the value of an XML-Attribute (RefBaseClassPath) and only apply the validation if it matches a certain value (*CommunicationRoleClassLib/PhysicalDevice/SensorPort*).

33

Schematron:

```xml
<rule context="caex:InternalElement[
 caex:RoleRequirements[@RefBaseRoleClassPath=
 'ArApcRcLib/SensorPort']]">
 <assert test="
  every $link in caex:InternalLink satisfies
   (//caex:ExternalInterface[
    @ID=$link/@RefPartnerSideA]/@RefBaseClassPath=
     'ArApcIcLib/SensorPortInterface' and
    //caex:ExternalInterface[
    @ID=$link/@RefPartnerSideB]/@RefBaseClassPath=
    'ArApcIcLib/SensorPortInterface')">
 </assert>
</rule>
```

SHACL:

```
:SPIConnectionShape
  a sh:NodeShape ;
  sh:targetClass aml:SensorPortInterface;
  sh:property [
    sh:path aml:isLinkedTo;
    sh:class aml:SensorPortInterface;] .
```

Listing 9: Schematron and SHACL constraints for Scenario 2.

While XSD can validate basic data types and structure, it doesn't support conditional validation and thus lacks the ability to handle complex relationships between elements or logic. Compared to XSD, Schematron and SHACL are far more expressive and flexible. However, Schematron is tied to XML's tree structure and can't natively handle ReferencePaths or InternalLinks. Solving validation scenarios thus involves complex iterations over the data, just as it was the case with xQuery.

Between the three approaches, XSD is most suitable for basic structural validation (e.g., ensuring elements appear in the correct order or enforcing data types), while both Schematron and SHACL can be used for cross-element validation with a lower level of complexity (s. Table 4). Compared to the other validation approaches, SHACL has access to a semantically described relationship model and can therefore define complex graph shapes that the data must conform to. As the complexity of the validation pattern increases, SHACL's advantages are expected to grow more pronounced.

Table 4: Capabilities of evaluated Validation Languages

Validation Property	XSD	Schematron	SHACL
Basic Structural Validation	+	+	+
Library-Level Cardinality Validation	-	+	+
Library-Level Connection Constraint	-	o	+

8 Conclusion

AutomationML is a widespread and expressive exchange format in the automation domain. In the past, multiple attempts have been made to incorporate the knowledge contained in AutomationML files into knowledge graphs. While these approaches have been partially successful, no common method of mapping the information contained in AutomationML-files to knowledge graphs exists. To mitigate this issue, this article provides:

1. An AutomationML ontology that is compatible with version 3 of the CAEX specification.

2. A declarative mapping that offers practitioners a straightforward transformation of AutomationML files to RDF triples.

The ontology[‡] as well as the mapping files and applications[§] are freely available to practitioners and can be found on Github. They were validated on multiple exemplary AutomationML files and were able to capture and adequately replicate the information contained in the file.

Two studies were conducted to evaluate the mappings' potential for graph queries and validation. They showed that the mapping to RDF offers expanded possibilities for querying as well as data validation.

8.1 Discussion

In comparison to previous approaches to mapping AutomationML files to RDF graphs, the use of an RML mapping offers multiple advantages. For one, all used artifacts (RML mapping and SPARQL queries) are independent of any implementation language. In case of updates to AutomationML specification, only the mapping needs to be updated, while the tools that use them can remain the same. This should result in lower total maintenance. The evaluation showed that mapping AutomationML files to RDF enables more complex use cases for querying and validation. Since common XML-based solutions were designed to handle a different type of graph structure, they can only operate on a subset of the information that AutomationML actually provides.

The experiments regarding automatic validation show that mapping the AutomationML file to RDF and subsequently applying SHACL validation rules

---

[‡]https://github.com/hsu-aut/IndustrialStandard-ODP-AutomationML

[§]https://github.com/hsu-aut/aml2owl

improve upon the validation capabilities of the current approach using XSD. The same conclusion was reached regarding the querying capabilities, in which SPARQL proved to be more capable and user friendly than xPath and xQuery.

8.2   Outlook and Future Work

This work is part of an ongoing effort to create a formal description of AutomationML. A plug-in for the AutomationML Editor is currently in development. It can be used to translate an AutomationML file to RDF and then query it using SPARQL or evaluate it against SHACL validation rules.

The artifacts provided by this work enable new uses of the information in an AutomationML file. The capability to formulate complex queries and create models based on the file contents allow new applications in the field of informed machine learning, in which prior knowledge of a system is used to improve various common AI tasks such as anomaly detection or diagnosis. These approaches would especially benefit from a connection between the static information from the AutomationML file with run time data from the system. Additionally, having a simple way to represent the files contents in an RDF graph allows the integration of the knowledge from the AutomationML file with other information artifacts about a system.

Acknowledgment

References

[1] R. Drath, A. Luder, J. Peschke, and L. Hundt, "AutomationML - the glue for seamless automation engineering," in *2008 IEEE Conference on Emerging Technologies & Factory Automation*, DOI 10.1109/ETFA.2008.4638461, pp. 616–623. Piscataway, NJ: IEEE, 2008.

[2] B. A. Talkhestani, T. Jung, B. Lindemann, N. Sahlab, N. Jazdi, W. Schloegl, and M. Weyrich, "An architecture of an intelligent digital twin in a cyber-physical production system," *at - Automatisierungstechnik*, vol. 67, DOI 10.1515/auto-2019-0039, no. 9, pp. 762–782, 2019.

[3] O. Kovalenko, I. Grangel-González, M. Sabou, A. Lüder, S. Biffl, and S. Auer, "AutomationML Ontology: Modeling Cyber-Physical Systems for Industry 4.0," *Semantic Web*, vol. 2018, 2018. [Online]. Available: https://www.semantic-web-journal.net/system/files/swj1855.pdf

[4] V. Jirkovsky, M. Obitko, and V. Marik, "Understanding data heterogeneity in the context of cyber-physical systems integration," *IEEE Transactions on Industrial Informatics*, vol. 13, DOI 10.1109/TII.2016.2596101, no. 2, pp. 660–667, 2017.

[5] D. Kalibatienė and O. Vasilecas, "Survey on ontology languages," in *International Workshop on Bibliometric-enhanced Information Retrieval*, 2011.

[6] P. Derler, E. A. Lee, and A. S. Vincentelli, "Modeling cyber–physical systems," *Proc. IEEE*, vol. 100, DOI 10.1109/JPROC.2011.2160929, no. 1, pp. 13–28, 2012.

[7] C. Hildebrandt, A. Köcher, C. Küstner, C.-M. López-Enríquez, A. W. Müller, B. Caesar, C. S. Gundlach, and A. Fay, "Ontology building for cyber–physical systems: Application in the manufacturing domain," *IEEE Transactions on Automation Science and Engineering*, vol. 17, DOI 10.1109/TASE.2020.2991777, no. 3, pp. 1266–1282, 2020.

[8] L. Abele, C. Legat, S. Grimm, and A. W. Müller, "Ontology-based validation of plant models," in *2015 IEEE 13th International Conference on Industrial Informatics (INDIN)*, DOI 10.1109/INDIN.2013.6622888, pp. 236–241, 2015.

[9] I. Grangel-González, D. Collarana, L. Halilaj, S. Lohmann, C. Lange, M.-E. Vidal, and S. Auer, "Alligator: A deductive approach for the integration of industry 4.0 standards," in *International Conference Knowledge Engineering and Knowledge Management*, pp. 272–287. Springer, Cham, 2016.

[10] I. Grangel-González, L. Halilaj, M.-E. Vidal, O. Rana, S. Lohmann, S. Auer, and A. W. Müller, "Knowledge graphs for semantically integrating cyber-physical systems," in *Database and Expert Systems Applications: 29th International Conference, DEXA 2018*, pp. 184–199. Springer, Cham, 2018.

[11] E. M. N. Gonçalves, A. Freitas, and S. Botelho, "An AutomationML based ontology for sensor fusion in industrial plants," *Sensors*, vol. 19, DOI 10.3390/s19061311, no. 6, p. 1311, 2019. [Online]. Available: https://www.mdpi.com/1424-8220/19/6/1311

[12] I. E. C. (IEC), "IEC 62424:2016: Representation of process control engineering," International Electrotechnical Commission (IEC), Tech. Rep., Jul. 2016.

[13] I. O. for Standardization (ISO), "ISO 17506:2022: Industrial automation systems and integration — COLLADA digital asset schema specification for 3D visualization of industrial data," International Organization for Standardization (ISO), Tech. Rep., Mar. 2022.

[14] I. E. C. (IEC), "IEC 61131-10:2019: Programmable controllers – Part 10: PLC open XML exchange format," International Electrotechnical Commission (IEC), Tech. Rep., Jul. 2019.

[15] T. Mersch and M. Schleipen, "A workflow towards a strongly typed AutomationML API," *at - Automatisierungstechnik*, vol. 71, DOI 10.1515/auto-2023-0038, no. 8, pp. 709–719, 2023.

[16] International Electrotechnical Commission (IEC), "IEC 62714-1:2018: Engineering data exchange format for use in industrial automation systems engineering - Automation Markup Language - Part 1: Architecture and general requirements," International Electrotechnical Commission (IEC), Geneva, Switzerland, Standard, Apr. 2018.

[17] International Electrotechnical Commission (IEC), "IEC 62714-2:2022: Engineering data exchange format for use in industrial automation systems engineering - Automation Markup Language - Part 2: Semantics libraries," International Electrotechnical Commission (IEC), Geneva, Switzerland, Standard, Oct. 2022.

[18] R. Drath, Ed., *AutomationML: A practical guide*, ser. De Gruyter Textbook. Berlin and Boston: de Gruyter Oldenbourg, 2021.

[19] R. Jasper and M. Uschold, "A framework for understanding and classifying ontology applications," in *Proceedings 12th Int. Workshop on Knowledge Acquisition, Modelling, and Management KAW*, vol. 99, pp. 16–21, 1999.

[20] A. Dimou, M. V. Sande, P. Colpaert, R. Verborgh, E. Mannens, and R. Walle, "RML: A generic language for integrated RDF mappings of heterogeneous data," *Proceedings of the 7th Workshop on Linked Data on the Web*, 2014.

[21] World Wide Web Consortium (W3C), "SPARQL 1.1 Query Language," W3C Recommendation, Mar. 2013, Accessed: 2024-02-02. [Online]. Available: https://www.w3.org/TR/sparql11-query/

[22] World Wide Web Consortium (W3C), "Shapes Constraint Language (SHACL)," W3C Recommendation, Jul. 2017, Accessed: 2024-02-02. [Online]. Available: https://www.w3.org/TR/shacl/

[23] S. Runde, K. Güttel, and A. Fay, "Transformation of the caex plant design data in owl: An application of semantic web technologies in automation technology," *VDI Berichte*, no. 2067, p. 175 – 178, 2009, cited by: 9.

[24] O. Kovalenko, M. Wimmer, M. Sabou, A. Lüder, F. Ekaputra, and S. Biffl, "Modeling AutomationML: Semantic Web technologies vs. Model-Driven Engineering," *2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA)*, 2015.

[25] M. Sabou, F. Ekaputra, O. Kovalenko, and S. Biffl, "Supporting the engineering of cyber-physical production systems with the AutomationML analyzer," in *2016 1st International Workshop on Cyber-Physical Production Systems (CPPS)*, DOI 10.1109/CPPS.2016.7483919, pp. 1–8. Piscataway, NJ: IEEE, 2016.

[26] Y. Hua and B. Hein, "Interpreting owl complex classes in automationml based on bidirectional translation," in *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, DOI 10.1109/ETFA.2019.8869456, pp. 79–86, 2019.

[27] E. Wallace and C. Golbreich, "OWL 2 web ontology language new features and rationale (second edition)," W3C, W3C Recommendation, Dec. 2012, https://www.w3.org/TR/2012/REC-owl2-new-features-20121211/.

[28] S. Sierla, M. Azangoo, and V. Vyatkin, "Generating an industrial process graph from 3d pipe routing information," in *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, DOI 10.1109/ETFA46521.2020.9212175, pp. 85–92. IEEE, 08.09.2020 - 11.09.2020.

[29] M. Ramonat and A. Fay, "Method for automatic simulation model calibration and maintenance for brownfield process plants," in *2023 IEEE 32nd International Symposium on Industrial Electronics (ISIE)*, DOI 10.1109/ISIE51358.2023.10227947, pp. 1–6. IEEE, 19.06.2023 - 21.06.2023.

[30] M. Ramonat, F. C. Kunze, F. Gehlhoff, and A. Fay, "Identifying root-causes of deviations between simulation and real plant data based on an adaptive causal directed graph," in *29th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 1–8. IEEE, 2024.

[31] E. Arroyo, A. Fay, M. Chioua, and M. Hoernicke, "Supporting plant disturbance analysis by dynamic causal digraphs and propagation look-up tables," in *INES 2015*, A. Szakál, Ed., DOI 10.1109/INES.2015.7329722, pp. 283–289, INES and Institute of Electrical and Electronics Engineers and IEEE Industrial Electronics Society and IEEE International Conference on Intelligent Engineering Systems. IEEE, 2015.

[32] F. C. Kunze and A. Fay, "Dynamic analysis of evolving industrial alarm floods using an adaptive causal directed graph," in *2023 IEEE 2nd Industrial Electronics Society Annual On-Line Conference (ONCON)*, DOI 10.1109/ONCON60463.2023.10430834, pp. 1–6. IEEE, 2023.

[33] B. Vogel-Heuser, A. Fay, B. Rupprecht, F. C. Kunze, V. Hankemeier, T. Westermann, and G. Manca, "Exploring challenges of alarm root-cause analysis across varying production process types," *at - Automatisierungstechnik*, vol. 72, DOI doi:10.1515/auto-2023-0180, no. 4,

pp. 369–386, 2024. [Online]. Available: https://doi.org/10.1515/auto-2023-0180

[34] AutomationML Association, "New Version of the AR APC," https://www.automationml.org/news/new-version-of-the-ar-apc-is-now-available/, 2023.