<AutomationML/>

The Glue for Seamless Automation Engineering

Application Recommendation: Formal Description for AutomationML

Document Identifier: AR FD4AML, V 1.0.0

State: September 2025

Version	Changes
1.0.0	Initial creation

Version 1.0.0, September 2025
Contact: www.automationml.org

Table of Contents

Table of Figures

Index of Tables

1 Introduction

1.1 Motivation

Since its founding, the AutomationML association has established AutomationML as a well-known data exchange format in engineering processes of various industries. One of the key factors for this development is its flexibility as well as its extensibility that allow for the adaption of new concepts and the integration of recent developments.

Over the years, many Best Practice Recommendations (BPR) as well as Application Recommendations (AR) were published, creating standards among different fields of expertise.

One downside of this development is the growing size and complexity in correctly modeling AutomationML, especially when using libraries defined in BPRs or ARs. Switching domains can prove quite difficult even for experts in AutomationML, since domain models and specifications differ heavily depending on the area of application. This problem becomes even more apparent as references, modeling guidelines and constraints are only specified in PDF files in natural language. As a result of this, there is no way to formally check conformity to a certain specification. In a worst-case scenario, a developer has to look at (and memorize) multiple documents and understand how they might affect one another to create a correct AutomationML file. This workflow is prone to errors and miscommunication and there is no way to objectively validate finished files.

The working group “Formal Description for AutomationML” was founded to counter these problems by discussing a formal representation. The results of this working group will provide a solution to automatically test AutomationML files for adherence to BPRs and ARs in the future.

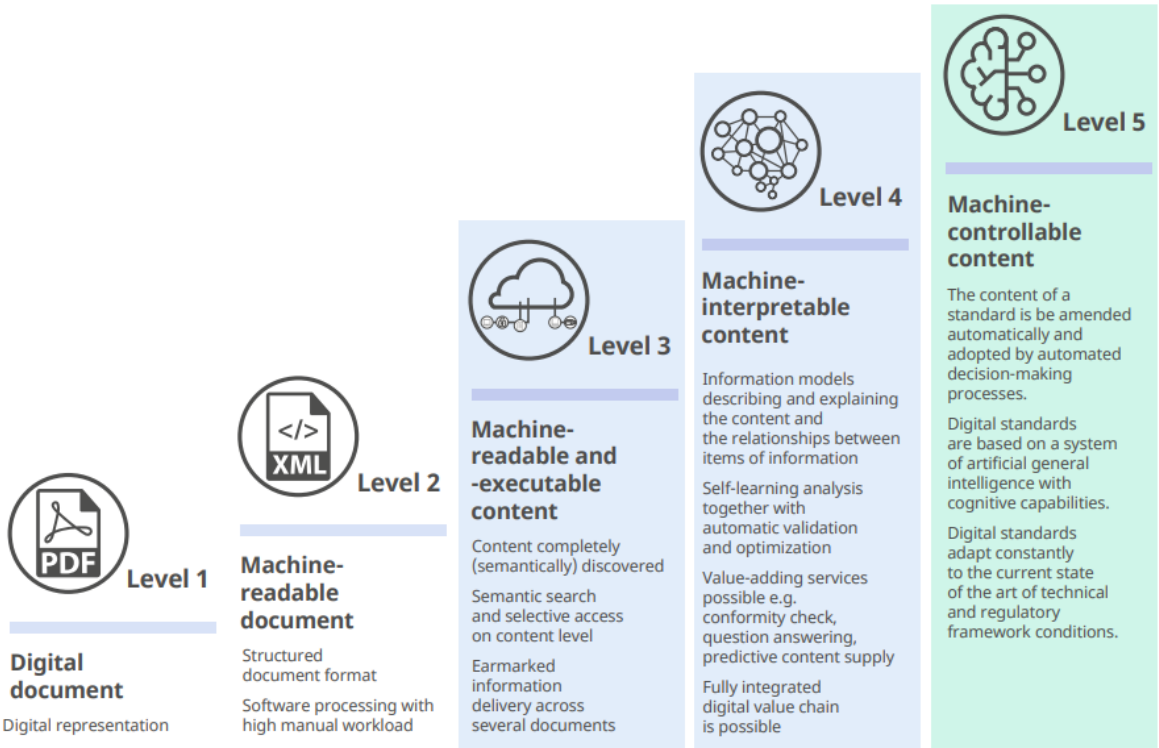

Figure 1: Five levels of smart standards

This above-described issue was addressed by DIN, DKE, IEC as well as ISO in the past. IEC and ISO stress the benefits of smart standards, i.e. standards with machine-readable information models:

possibility of cost optimization for manufacturers, consistency in regulations and standards, acceleration of the creation of standards through automation and a better usability for the and users [1].

A whitepaper formulating levels of smart standards and their benefits was released, calling for the usage of such standards [2]. Figure 1 shows the different levels of smart standards as defined in said paper.

This working group aims to turn ARs and BPRs into level 4 standards, thus paving the way for AI use case as well as big data integration.

1.2 Scope

AutomationML as a standard has different facets of validity, some of which can already be checked as of today. The working group sees six such use-case:

1. Whether the file is valid XML.

2. Whether the file is CAEX-XML-Schema compliant.

3. Whether the file complies with AutomationML whitepaper parts one [3] and two [4].

4. Whether the appropriate library elements from the relevant specification part (AR/BPR) are used.

5. Whether the hierarchy constraints from the relevant specification part (AR/BPR) are followed.

6. Whether the semantic constraints defined in the relevant specification part (AR/BPR) are satisfied.

Points 1 and 2 are covered by standard XML validation mechanisms. For point 3, the AutomationML association offers resources through the AutomationMLEditor and AutomationMLEngine. For limited use cases (e.g. attribute cardinalities, or attribute value constraints) AutomationML further offers validation checks in the form of constraints.

However, points 4, 5 and 6 require more powerful verification models and are the focus of this working group. Examples of such complicated rules and how they are realized in the working group's chosen approach can be found in chapter 4.2.

This working group envisions two main use-cases. One is creating a mapping from AutomationML to an ontology, for which an automated mapping tool was created (see chapter 3 for details), the other focus point is creating SHACL shapes, that allow the automatic verification of an AutomationML files' conformancy to an AR or BPR.

In these use-cases, two personas are relevant: A model designer who wants to create AutomationML files based on a given recommendation (the lower path of Figure 2) and a working group lead of a BPR / AR group who needs to formalize the SHACL statements (the upper path of Figure 2). These two use cases are also shown in the picture below. The goal of the combination of both of these use cases is machine generated feedback, describing what constraint was violated as well as where.

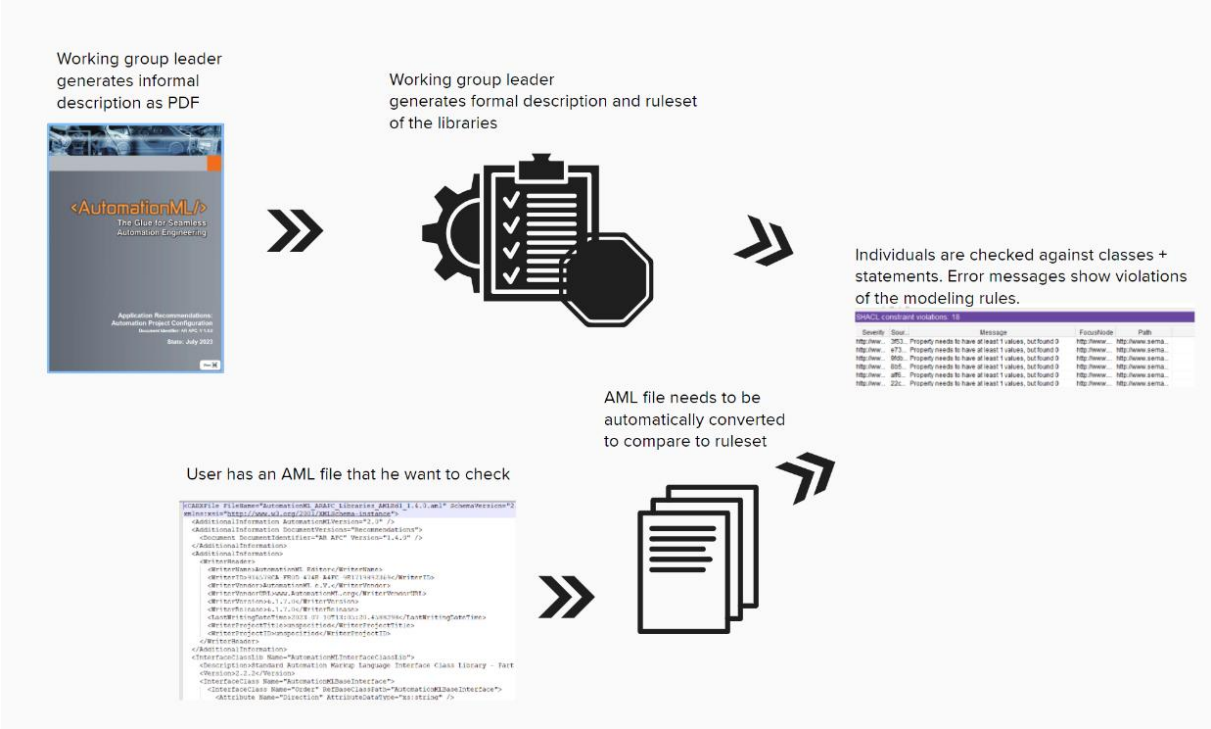

Figure 2: Workflow of creating and using AutomationML SHACL-files

1.3 Use Cases

The working group aims to ease working with different AutomationML domain models and specifications. While doing that, it focuses on the following issues:

1. Helping new users
   The constantly growing number of AutomationML domain models and specifications oftentimes overwhelms new users. Writing clean and correct AutomationML while keeping the domain models in mind leads to errors and frustration. An add-on for the AutomationML editor giving feedback on written classes would lower initial hurdles when picking up AutomationML as new user.
2. Connecting multiple AutomationML domain models and specifications
   Sometimes more than one specification is applicable while developing an AutomationML file. When this happens, the user has to cross check multiple specification-documents and research how they might affect one another. The complexity of this task can be reduced significantly by storing the domain models and specifications in a formalized manner, by merging multiple domain models and giving out a single formal rule set which it can be validated against.
3. Improving AutomationML file quality
   Today, the semantic conformity of a file to a specification is only checked informally by analyzing the source in natural language. Naturally this way of working is prone to errors. Creating a formal description offers more precise specifications and make way for a powerful tool to create even better AutomationML documents.
4. Certification
   In the future, a file that follows its domain model could get a stamp of approval. This could be a free or paid service. It would be a way of proving that a file is written correctly and meets the standards set by the chosen domain models.

1.4 Requirements

There are a number of requirements a suitable description has to meet in order to be considered applicable for given use case. The working group agreed on eight of such requirements:

1. Expressive power: This describes how many rules can be created through chosen means. Since the rules of domain models can get complicated (conditional branching and Internal Elements placing demands on one another are only a few examples). Thus, the chosen approach must include a wide variety of functions and functionalities to be able to properly model these constraints.
2. Extensibility: While an initial set of rules will be created, more and more will be needed in the future. On the one hand, existing recommendations might be changed, on the other hand, new recommendations with even more complex rules might be created. It is important for the chosen description to be easily extensible.
3. Rule creation: Creation of rules should be feasible, even for non-professional developers.
4. Rule application: Rules should be easy to apply, and the result of the check should be easy to understand. In a best-case scenario, the user would press a single button and receive a detailed explanation, where and why they conflict the domain model.
5. Readability: The rules themselves should be easy to read. Everyone should be able to look at them as a resource similar to a .pdf-document. Even without deeper knowledge of the chosen language, the reader should be able to follow the given rules.
6. Tool support: For rule creation there should be a easy to use open-source application. The working groups, which will be part of the rule creation process, must not be required to purchase and learn a tool for using the chosen approach. Instead, this working group aimed for an open-source tool with a set of basic functions that can be understood in a few hours.
7. XML applicability: Since AutomationML is based on XML, it is important for the chosen approach to work with XML. If it doesn't, a conversion has to be implemented.
8. Implementable in the AutomationML Editor: The AutomationML Editor is the preferred tool for users who wish to manually handle AutomationML files. It's crucial that the method selected for managing constraints can be incorporated into the AutomationML Editor without requiring an excessive amount of work.

1.5 Reason for chosen approach

As described in 1.2, AutomationML already possesses ways to check the integrity of a document. These methods are already implemented in the AutomationML-Editor (for creating them as well as using them for validation), they are applicable to XML (since they are natively written for AutomationML), the application and creation of the rules follow the same workflow as writing documents in the AutomationML-Editor and are as such easy for users to adapt to. That being said, the problem that became apparent, as the requirements were discussed was, that these methods were not powerful enough to model complex domain model definitions. The rules that can be written with the AutomationML-Editor internal tools are extensible by nature, since they are short, self-contained statements. But statements in domain models are often conditional or even derived of other statements.

For this reason, ontologies (in combination with ontology-constraints) were decided as the chosen approach. Ontologies offer a powerful tool to model a wide range of different scenarios. They are extensible and easy to combine with one another which makes them particularly convenient for our use case.

To utilize them, a conversion from AutomationML to ontology syntax is needed, which will be discussed in chapter 3. The chosen approach for constraints as well as the tooling will be discussed in

chapter 4. In this context, some examples will showcase the above-mentioned complexity of domain models as well as how this working group's approach to a formalization of said domain model looks like.

Apart from this working group's evaluation several other benefits are given by choosing ontologies. One of them is connectivity to other formats and industry-standards. The IEC emphasizes the importance of semantic interoperability through ontologies [5] as a way to mitigate the growing diversity in systems and platforms. This goal is addressed by the AutomationML e.V. through this working group and is also supported by other standards, such as the Asset Administration Shell (AAS). The results of this working group enable better integration into the expanding field of diverse standards and use cases, further reinforcing AutomationML's position within the production planning landscape.

Another advantage of this conversion is the ability to utilize the benefits of ontologies for AutomationML files after translating them. As such, AutomationML (or more specifically the newly generated aml-ontology) can now be queried similarly to SQL-databases to further interact with and extract data in a more meaningful manner. Ontologies also allow for semantic inferencing, which helps the users searching for patterns or previously unknown relationships in a context of ever-growing data flood.

2 AutomationML Ontology

As stated in the previous Section, the use of Semantic Web Technologies is the most promising approach to extend the capabilities of AutomationML with regards to complex data validation.

To make use of these capabilities two prerequisites need to be met:

1. an AutomationML Ontology, that defines the necessary concepts
2. a mapping application, that automatically converts AutomationML into RDF-Triples.

This section gives a brief overview over the AutomationML Ontology, while the following sections describe a declarative mapping and a mapping application.

The AutomationML Ontology covers all major entity types and entity relations from CAEX 3.0. The different entity types are translated to OWL classes (s. Table 1), while the different entity relations are converted to OWL object properties (s. Table 2). Additionally, OWL data properties are provided to add additional information, like human readable names or attribute values.

Together, the AutomationML Ontology comprises a complete vocabulary of the entities defined by CAEX 3.0. AutomationML also offers the possibility to define additional Classes in the form of Role-, SystemUnit- or InterfaceClasses, as well as AttributeTypes. These additional classes are needed to fully capture the intended semantics of the AutomationML file. However, these additional classes depend on the AutomationML libraries that are imported into the project and are therefore different between different projects. Because of this, they are therefore not part of the core AutomationML Ontology. Instead, they are created from the AutomationML file during the mapping process. Their creation is covered in more detail in the following sections.

Table 1: Main concepts from AutomationML and their corresponding OWL classes.[1]

AutomationML Entity	**OWL Class**
InternalElement	:InternalElement
SystemUnitClass	:SystemUnitClass
RoleClass	:RoleClass
ExternalInterface	:ExternalInterface
InterfaceClass	:InterfaceClass
Attribute	:Attribute
AttributeType	:AttributeType

AutomationML Entity	**OWL Class**
InstanceHierarchy	:InstanceHierarchy
RoleClassLibrary	:RoleClassLib
SystemUnitClassLibrary	:SystemUnitClassLib
InterfaceClassLibrary	:InterfaceClassLib
AttributeTypeLibrary	:AttributeTypeLib
Constraints	:Contraints
Metadata	:Origin, :Project

[1] More details about these mappings are given in the following chapter 3. A full list of mappings would go beyond the scope of this paper. A more comprehensive explanation on the choices made when create the mapping can be found in [12]. Apart from that, the implementation of the tool discussed in chapter 5 is open source. As a result, the functionality used for the conversion is open to look at on its Git page (https://github.com/hsu-aut/aml2owl.)

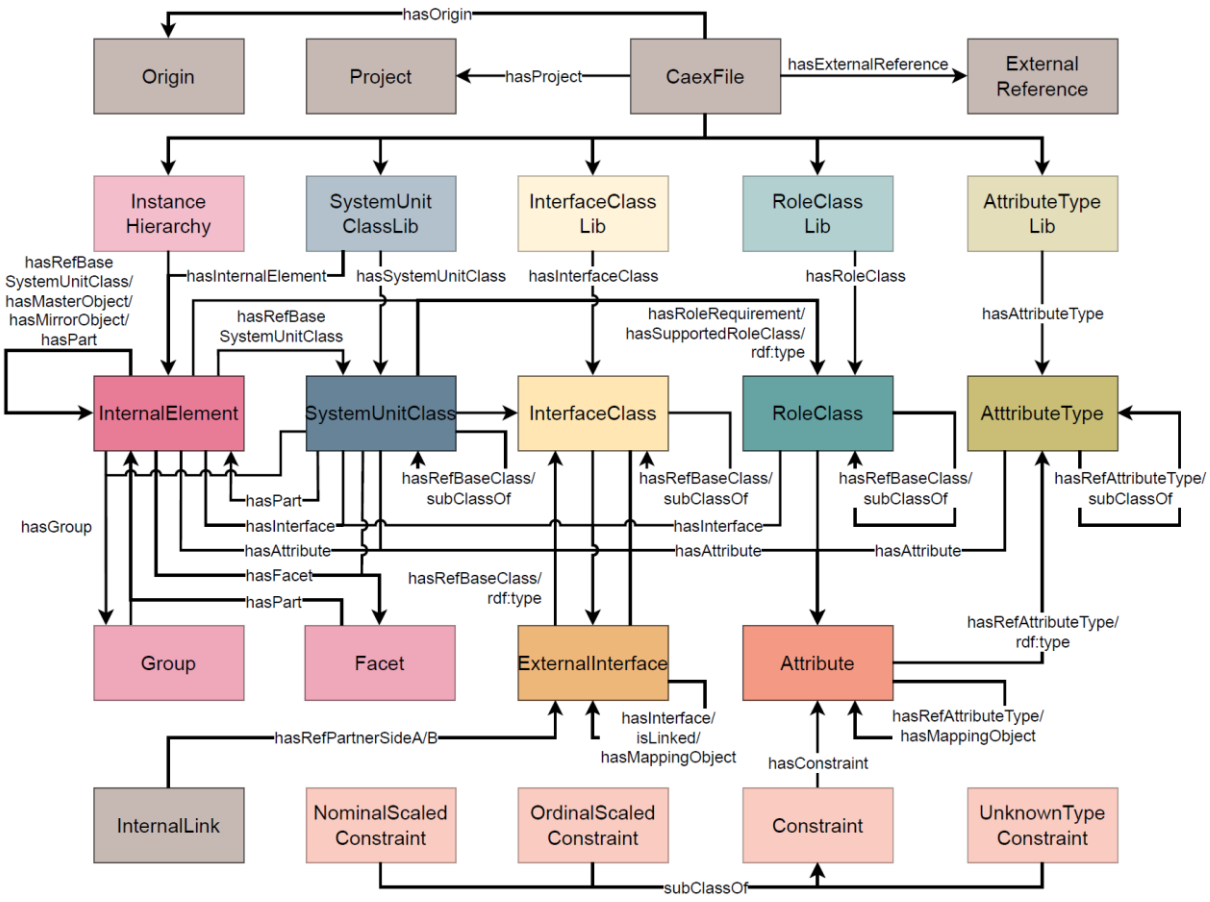

Figure 3: Overview of the AutomationML Ontology

Table 2: Main Relations from AutomationML and their corresponding OWL Object Properties.

AutomationML Relation	**OWL Class**	**Usage**
XML Element Structure	:hasAttribute, :hasInterface, :hasPart	Multiple meanings, dependent on use. Connecting AutomationML Elements to their Attributes/Interfaces/Subcomponents
InternalLink	:isLinkedTo	Connecting two Interfaces
RoleRequirement	:hasRoleRequirement	Assigning a RoleRequirement to an InternalElement or SUC.
SupportedRoleClass	:hasSupportedRoleClass	Assigning a supported RoleClass to an InternalElement or SUC.
RefBaseClassPath	:hasRefBaseClass, rdfs:subClassOf, rdf:type	Multiple meanings, dependent on use.
RefAttributeType	:hasRefAttributeType, rdfs:subClassOf, rdf:type	Multiple meanings, dependent on use.
RefBaseSystemUnitClass	:hasRefBaseSystemUnitClass, :hasMirror	Multiple meanings, dependent on use.
Name	:hasName	Assigning a human readable label.

3 Mapping between AutomationML and OWL

Following the introduction of the ontology, this chapter presents the declarative, rule-based mapping from AutomationML to OWL. The primary goal of this mapping is to translate the structured information encoded in an AutomationML model into an OWL representation, thereby enabling subsequent validation using SHACL constraints. The chapter first outlines the core transformation rules that map AutomationML elements to OWL classes, properties, and individuals. This is followed by a description of an application developed to automate the mapping process. The tool serves as a bridge between AutomationML-based data and semantic technologies, forming the technical basis for SHACL-based constraint checking.

3.1 Mapping Rules and Examples

The following Section will give a brief overview on the mapping rules that are applied to populate the RDF-graph based on an AutomationML-File.

3.1.1 IRI Creation

Every resource within an RDF graph must be identified by an Internationalized Resource Identifier (IRI), distinguishing 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 instead 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 ExternalInterfaces), 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.
Listing 1 and Figure 4 show an example of this.

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

Listing 1: IRI creation

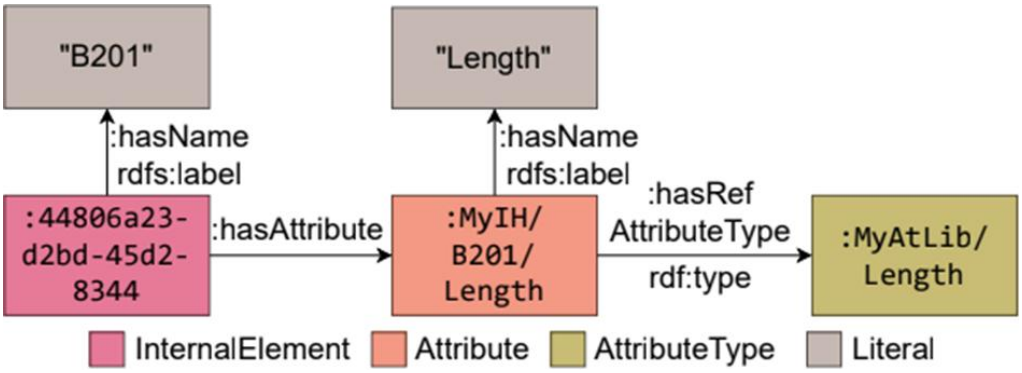

Figure 4: IRI creation

3.1.2 Creation of Classes and Instances

For the mapping process, a decision had to made which AutomationML concepts would be mapped as classes and as instances. For InternalElements, ExternalInterfaces and Attributes, this decision is trivial: They are unique individuals. For RoleClasses, SystemUnitClasses and InterfaceClasses the distinction is less clear, since 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 AutomationML can also function as a template for instances - i.e. they can possess 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. 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 AttributeType is converted to an OWL class. The inheritance mechanisms of AutomationML are used to create a OWL class hierarchy (s. Section X). Class membership of instances is implied, if instances reference the class using hasRefSystemUnitClass, hasRefBaseClass or hasRefAttributeType respectively.

RoleClasses define the abstract functions of either InternalElements or SystemUnitClasses.. AutomationML allows two types of RoleClass assignments:

1. RoleRequirement can be used to describe roles that an InternalElement or a SystemUnitClass is required to fill in a plant.
2. SupportedRoleClass can be used if an InternalElement or a SystemUnitClass can fill a role.

These relationships are modeled using two complementary mechanisms. If an InternalElement or SystemUnitClass references a RoleClass via SupportedRoleClass, it is considered an instance of that class. Additionally, the object properties aml:hasRoleRequirement and aml:hasSupportedRoleClass are assigned to preserve the information regarding which roles need to be filled, and which can be filled by any given InternalElement or SystemUnitClass.

Listing 2 contains an example of an `InternalElement`, that has all major 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 5 shows the corresponding relationship between OWL instances and classes from Listing 2: 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 AutomationML data additionally connects the instances to their classes using the AutomationML terminology.

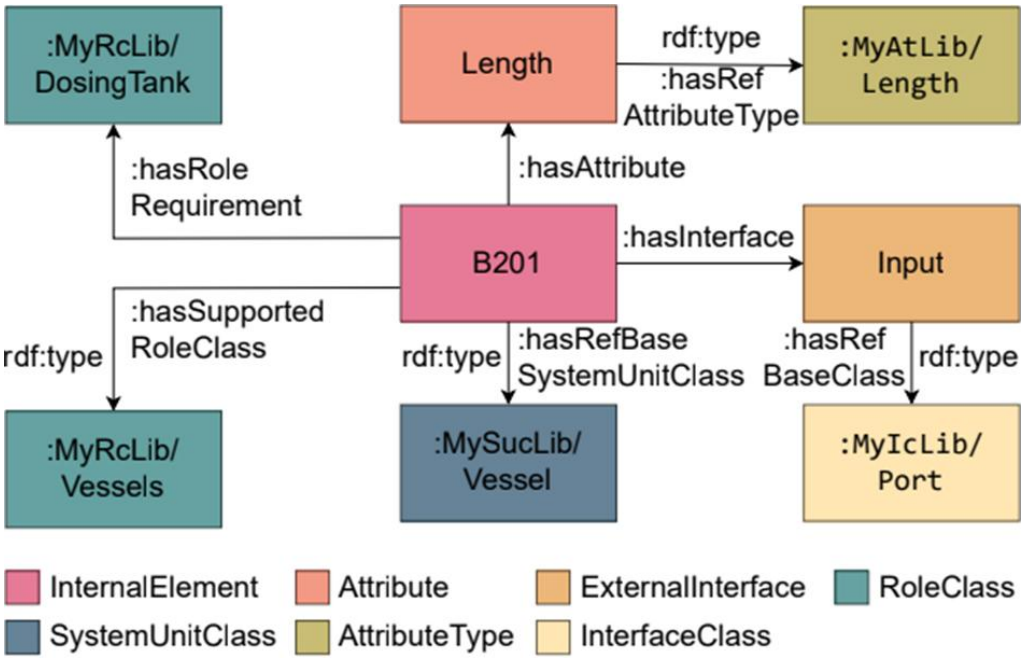

Figure 5: Creation of Classes and Instances

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

Listing 2: Creation of Classes and Instances

3.1.3 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. The example in Listing 3 and Figure 6 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. OWL Class hierarchies are similarly created from all AutomationML libraries, i.e. also from the Role Class, System Unit Class and Interface Class Libraries.

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

Listing 3: Mapping of Class Hierarchies

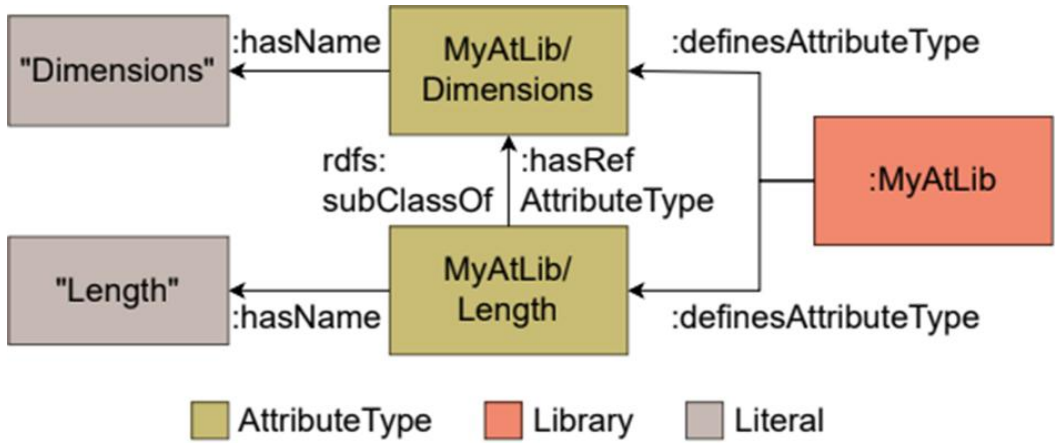

Figure 6: Mapping of class Hierarchies

3.1.4 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 hierarchy in the source document. In the Ontology, this implies a PartOf-Relationship and is therefore modeled using the object properties:

- aml:hasPart. for nested structures of InternalElements, Interfaces and Attributes
- aml:hasInterface and aml:hasAttribute for assignment of Interfaces and Attributes to AutomationML Elements

3.1.5 Inheritance of Subcomponents

In AutomationML, an object's internal structure is inherited from a parent class to the child class.
In the AutomationML-file, this relationship is not explicitly modeled, but can instead be inferred through the inheritance relationship. It is additionally possible to overwrite inherited objects.
For the mapping to RDF, this means that an object is inherited from the parent class if the inherited object was not overwritten. If the object was overwritten, a new instance is created instead.

Listing 4 shows an example of this: It shows three SUCs, that have an Interface called *Input*. The SUC *Tank* inherits this interface from the SUC *Vessel*. It is therefore not duplicated in the RDF-Representation. This approach mimics how AutomationML handles these situations.

```
<SystemUnitClassLib Name="MySUCL">
  <SystemUnitClass Name="Vessel">
    <ExternalInterface Name="Input" ID="ce45d7b3-169d-4be8-9746"
      RefBaseClassPath="MyIcLib/Port"/>
  </SystemUnitClass>
  <SystemUnitClass Name="Tank" RefBaseClassPath="MySUCL/Vessel"/>
```

```
  <SystemUnitClass Name="MixingTank" RefBaseClassPath="MySUCL/Vessel/MixingTank">
    <ExternalInterface Name="Input" ID="0f4ffc29-bb2a-47c1-8d53"
      RefBaseClassPath="MyIcLib/Port"/>
  </SystemUnitClass>
</SystemUnitClassLib>
```

Listing 4: Example of Inheritance of Subcomponents

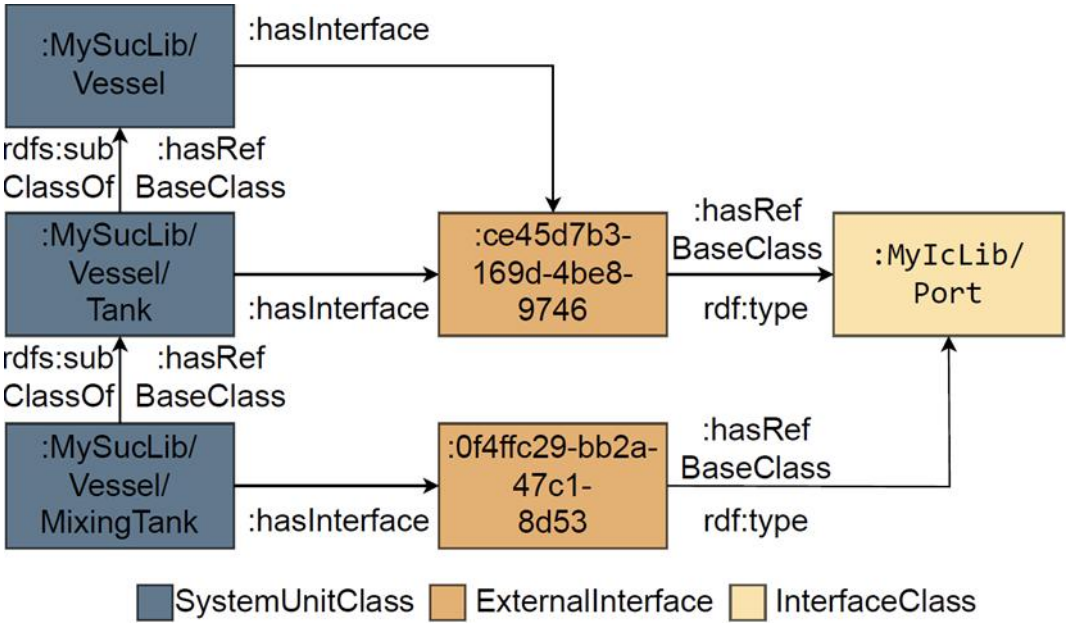

Figure 7: RDF-Representation of three SystemUnitClasses with Interfaces. One interface is Inherited from the parent element.

3.1.6 Mapping of 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. In AutomationMl these Links function as directional connections between two interfaces. The mapping uses this information in two ways:

Listing 5 shows an example of two `InternalElements`, whose interfaces are connected by an `InternalLink`. Figure 8 shows both mechanisms of linking their 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 `InternalElement` records (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-"/>
  </InternalElement>
  <InternalElement Name="B201" ID="44806a23-d2bd-45d2-8344">
```

```
    <ExternalInterface Name="Input" ID="ce45d7b3-169d-4be8-9746"/>
  </InternalElement>
  <InternalLink Name="Pipe_B201" RefPartnerSideA="6eea7a40-43fd-4aee-a1d3"
                                 RefPartnerSideB="ce45d7b3-169d-4be8-9746"/>
</InstanceHierarchy>
```

Listing 5: Mapping of InternalLinks

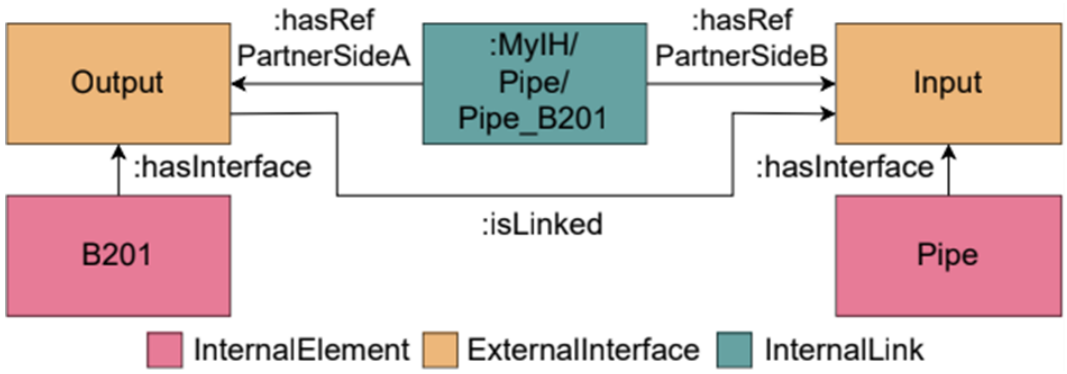

Figure 8: Mapping of InternalLinks

4 Formal Rule Description/ Constraint Language

After mapping the AutomationML file to the OWL format, SHACL shapes can be applied to the ontology to validate its conformancy to an AR or BPW. The following section will introduce the Shapes Constraint Language SHACL using both a generic application as well as an example AutomationML project.

4.1 Shapes Constraint Language

The Web Ontology Language (OWL) operates under the Open World Assumption (OWA), meaning that the absence of information does not imply its negation. Consequently, logical reasoning alone cannot verify whether specific facts exist within an ontological model. While the OWA is beneficial for inferring new knowledge in the presence of incomplete data, it can create challenges when ontologies interact with systems that rely on the Closed World Assumption (CWA).

Consider the following example of an OWL class `Driver` defined in Manchester Syntax, which represents individuals who are authorized to operate a vehicle. In a well-structured ontology, we want to ensure that only those individuals who possess a valid driving license can be classified as `Driver`. To implement this rule, we define `Driver` as a subclass of all individuals who have at least one `hasLicense` relationship to an instance of `DrivingLicense`.

```
Class: Driver
    SubClassOf: hasLicense some DrivingLicense
```

Listing 6: OWL class expression defining everything that has a driving license to be a Driver

This definition states that for an individual to be considered a `Driver`, they must have at least one `hasLicense` relation linking them to an instance of `DrivingLicense`. However, as OWL operates under the OWA, the absence of information does not imply its negation. As a result, the reasoner will not flag an inconsistency if an individual is explicitly assigned the type `Driver` but lacks a `hasLicense` relationship.

To ensure that specific information is present within an ontology, validation mechanisms that follow the Closed World Assumption (CWA) are required. One widely recommended language for validating such models against predefined constraints is the Shapes Constraint Language (SHACL) [6]. Designed specifically for this purpose, SHACL verifies whether a given model adheres to a defined set of constraints. SHACL is a W3C-standardized language used to define complex conditions for validating ontological models represented as RDF graphs [7]. In SHACL, constraints are expressed as shapes, which themselves take the form of an RDF graph. These shape graphs are then evaluated against an RDF data graph to determine compliance [6]. The core SHACL class, `sh:Shape`, has two primary sub-classes: `sh:NodeShape` and `sh:PropertyShape`. Node shapes define constraints that apply directly to individual nodes, referred to as focus nodes. Property shapes, on the other hand, impose constraints on nodes that are related to a given focus node, referred to as value nodes. Each property shape enforces constraints on value nodes along a specified path [6], which is represented by the `sh:path` property. These two shape types can be combined, meaning a node shape can include multiple property shapes using the `sh:property` construct. Additionally, SHACL supports various types of paths, including inverse paths, alternative paths, and sequential paths [7], allowing for flexible constraint modeling.

With a SHACL processor, a given data graph can be validated against one or more shape graphs to generate a validation report. Each constraint within the shape graph is checked, and if all constraints are met, the data graph is considered valid. If a constraint is violated, a negative validation report is

returned, containing the focus node, path, and possibly the value node of the constraint, along with a predefined message and severity of the violation.

The following example of a SHACL constraint can be used to check all individuals of the `Driver` class `for at least one` **`hasLicense`** `relation`. This constraint would fail for all individuals of the `Driver` class without a `hasLicense` relation.

```
:DriverShape a sh:NodeShape ;
    sh:targetClass :Driver ;
    sh:property [
        sh:path :hasLicense ;
        sh:minCount 1 ;
        sh:message "A driver must have at least one driving license." ;
    ] .
```

Listing 7: Example of a SHACL shape for a driver. Every driver must have at least one driving license

4.2 Using SHACL for AutomationML-Library constraints

The following Section will describe an exemplary use case, where SHACL is used to validate the correct usage of imported AutomationML elements. For the validation example, we are considering the Application Recommendation Automation Project Configuration (AR APC), that was published by the AutomationML Association in X.

The AR APC provides a RoleClass- and InterfaceClassLibrary, which define Role- and InterfaceClasses, that can be reused in other projects. The AR APC further defines how those Role- and InterfaceClasses should be used.

To show how SHACL can be used, we will consider the validation of a communication network according to the AR APC. For this example, two RoleClasses (and their respective Interfaces) are of importance – Subnets and Nodes.

The RoleClass Subnet is used to represent the properties and functionalities of Networks (e.g. Ethernet, ProfiBus, etc.). The RoleClass Node is used to represent all the interface related networking information of a networking node (e.g. logical address, subnet mask). Both Subnets and Nodes have an Interface, that has the InterfaceClass LogicalEndPoint. If the Interfaces of a Subnet and a Node are connected in this manner, the Node is part of the Subnet.

To minimize the ambiguity in its usage, the AR APC further defines rules for the proper usage and linking of Subnets and Nodes. For the application example, the following three rules will be considered:

Rule 1: Subnets and Nodes must have exactly one LogicalEndpoint each.

Rule 2: The LogicalEndPoint of a Subnet can only be connected to another LogicalEndPoint.

Rule 3: The LogicalEndPoint that a Subnet's LogicalEndPoint is connected to must be part of a Node (and the other way around).

Figure 9 shows an example of these rules. It shows a Subnet with a LogicalEndPoint, that is linked to three LogicalEndPoints of different Nodes. However, only one of the connections – the one with the LogicalEndPoint of Node1 - is valid according to the usage rules from the AR APC.

The connection to the LogicalEndPoint of Node2 violates Rule 2, since it does not connect two instances of the InterfaceClass LogicalEndPoint.

The third connection however, violates Rule 3. While it connects two LogicalEndPoints, neither of them is part of a Node. Additionally, the Subnet also has two LogicalEndPoints, which violates Rule 1.

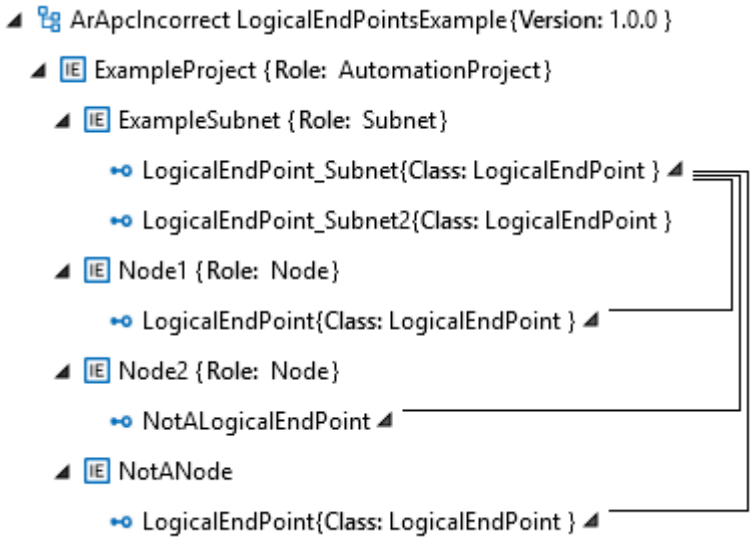

Figure 9: Subnets and Nodes according to the AR APC.

Since the validation of these modeling rules involves validating non-trivial rules, they cannot be validated using common XML-based validation approaches like XSD. Instead, the AutomationML file is translated to OWL using the approach outlined in Chapter 5. This allows the formulation of SHACL constraints that validate the correct usage of the Subnet and Node RoleClasses.

For all rules, the shapes consist of two parts: the first is the definition of target nodes, which in our case is done using a target definition written in the query language SPARQL. All elements that match this query are then compared to the second part of the shape, which is a definition of a shape that these elements must conform to. These shapes can be written in SHACL or once again as a SPARQLConstraint.

For the first rule (s.Listing 8), the SPARQLTarget selects all InternalElements with the RoleRequirement *Subnet*. The SHACL constraint then checks that all targeted InternalElements have exactly one Interface, that has the InterfaceClass *LogicalEndPoint.*

```
ex:InternalElementSubnetShape
    a sh:NodeShape ;
    sh:target [
        a sh:SPARQLTarget ;
        sh:select """
            PREFIX aml: <https://w3id.org/hsu-aut/AutomationML#>
            SELECT ?ie
            WHERE {
                ?ie a aml:InternalElement ;
                aml:hasRoleRequirement aml:AutomationProjectConfigurationRoleClassLib%2FSubnet.
            }
           """ ;
```

```
    ] ;
    sh:property [
        sh:path aml:hasInterface ;
        sh:class aml:CommunicationInterfaceClassLib%2FLogicalEndPoint ;
        sh:minCount 1 ;
        sh:maxCount 1 ;
        sh:message "A Subnet must have exactly one LogicalEndpoint to connect the subnet to
Nodes."@en;
    ] .
```

Listing 8: SHACL Shape, that Targets all InternalElements with the RoleRequirement Subnet, and checks that it has exactly 1 LogicalEndPoint.

For the second and third rule (s. Listing 9 and Listing 10), the SPARQLTarget selects all Interfaces that have the InterfaceClass *LogicalEndPoint* and are part of an InternalElement with the RoleRequirement *Subnet*. On those targets, two constraints are declared: The first, written in SHACL, states that the LogicalEndPoint must be linked to another LogicalEndPoint. The second, written as a SPARQLConstraint, states that the LogicalEndPoint it is connected to must be part of an InternalElement that has the RoleRequirement *Node*.

```
ex:LogicalEndpointsOfSubnetsShape
    a sh:NodeShape ;
    sh:target [
        a sh:SPARQLTarget ;
        sh:select """
            PREFIX aml: <https://w3id.org/hsu-aut/AutomationML#>
            SELECT ?lep
            WHERE {
                ?lep a aml:CommunicationInterfaceClassLib%2FLogicalEndPoint .
                ?ie a aml:InternalElement ;
                    aml:hasRoleRequirement aml:AutomationProjectConfigurationRoleClassLib%2FSubnet .
                ?ie aml:hasInterface ?lep.
            }
        """ ;
    ] ;
    sh:property [
        sh:path [ sh:alternativePath ( aml:isLinkedTo [ sh:inversePath aml:isLinkedTo ]  ) ];
        sh:class aml:CommunicationInterfaceClassLib%2FLogicalEndPoint ;
        sh:minCount 1 ;
        sh:message "The LogicalEndPoint of a Subnet must be connected to another LogicalEndPoint."@en;
    ] ;
    sh:sparql [
        a sh:SPARQLConstraint ;
        sh:message "LogicalEndPoints must only be linked to LogicalEndPoints that are part of Nodes."@en ;
        sh:select """
            PREFIX aml: <https://w3id.org/hsu-aut/AutomationML#>
            SELECT ?this
            WHERE {
                {
```

```
                # Get all directly or inversely linked LogicalEndPoints
                ?this (aml:isLinkedTo|^aml:isLinkedTo) ?otherLEP .
                # Follow path to the InternalElement that owns it
                ?ie aml:hasInterface ?otherLEP .
                # Fail if that InternalElement is NOT a Node
                FILTER NOT EXISTS {
                    ?ie aml:hasRoleRequirement aml:AutomationProjectConfigurationRoleClassLib%2FNode .
                }
            }
        }
    """ ;
] .
```

Listing 9: SHACL Shape, that Targets all LogicalEndPoints that are part of Subnets and verifies that it is a) connected to another LogicalEndPoint that b) is part of a Node.

Upon execution of the validation, three violations of the previously defined rules are marked and described alongside additional information. Among other things, the validation result points out the offending node (described by its ID) as well as a previously defined human-readable result message. All three errors from the example files are correctly detected.

```
[ rdf:type     sh:ValidationReport;
  sh:conforms  false;

  sh:result   [ rdf:type sh:ValidationResult;
                sh:focusNode  aml:5f9364fa-34b0-4818-9255-96a684af069a;
                sh:resultMessage "The LogicalEndPoint of a Subnet must be connected to another LogicalEndPoint."@en;
                sh:resultPath     [ sh:alternativePath  ( aml:isLinkedTo
                                        [ sh:inversePath  aml:isLinkedTo ])
                                  ];
                sh:resultSeverity  sh:Violation;
                sh:sourceConstraintComponent  sh:MinCountConstraintComponent;
                sh:sourceShape                []
              ];
  sh:result    [ rdf:type                     sh:ValidationResult;
                 sh:focusNode                 aml:18026f8e-7d79-461c-8857-d237860006e3;
                 sh:resultMessage             "LogicalEndPoints must only be linked to LogicalEndPoints that are part
of Nodes."@en;
                 sh:resultSeverity            sh:Violation;
                 sh:sourceConstraint          [] ;
                 sh:sourceConstraintComponent  sh:SPARQLConstraintComponent;
                 sh:sourceShape               ex:LogicalEndpointsOfSubnetsShape;
                 sh:value                     aml:18026f8e-7d79-461c-8857-d237860006e3
               ];
  sh:result    [ rdf:type                     sh:ValidationResult;
                 sh:focusNode                 aml:0dee2e6e-dcbe-46ec-8270-b63ebfe01a2b;
                 sh:resultMessage              "A Subnet must have exactly one LogicalEndpoint to connect the subnet
to Nodes."@en;
```

```
                sh:resultPath                aml:hasInterface;
                sh:resultSeverity            sh:Violation;
                sh:sourceConstraintComponent sh:MaxCountConstraintComponent;
                sh:sourceShape               []
              ]
] .
```

Listing 10: Validation result of the three SHACL constraints on the example from Figure 9.

As these results show, translating AutomationML to OWL offers new possibilities of validating the correct use of imported libraries. Using SHACL greatly expands the capabilities of common validation approaches for XML based data like XSD.

Templates for the most common validation shapes are provided in a separate document and may be used as guidance for the creation of custom constraints.

5 *AML2OWL* Mapping Application

Through different tools, ontologies can be edited in a hierarchical tree structure (similar to the AutomationML-Editor) this eases the use of this technology even for less experience's users.

That being said, there are two limitations of our approach: Firstly, there is no support SHACL-rule creation. As such the rules must be created in a simple text editor. After some discussion the working group agreed that most of the time the same ruleset will be needed (checking for a naming convention, or the need to implement a certain attribute, for example). Appended to this document a file can be found which includes copy-paste phrases for the most common rules. Complicated rules will have to be written with help of SHACL-experts.

The second downside to ontologies is the missing conversion to XML (and as such AutomationML). For this a conversion-tool will be provided which translates a given AutomationML file into an ontology.

In order to automatically execute the transformations presented in the previous section all transformations are expressed as declarative mapping rules using the RDF Mapping Language (RML) [1] as well as SPARQL updates. A mapping application called *AML2OWL* was developed in order to automatically execute these mapping rules. After mapping the contents of an AutomationML file, *AML2OWL* can also be used to validate the generated ontology against SHACL constraints.

AML2OWL is implemented in the Java programming language and is available as open-source software at https://github.com/hsu-aut/aml2owl.

5.1 Structure

AML2OWL provides the two core functions of (1) automated mapping of AutomationML models to OWL and (2) SHACL validation of the generated models.

The mapping function of *AML2OWL* can be roughly divided into two main steps. First, RML-Mapper[2] is used to execute a mapping document with all mapping rules (see Section 5.1). Executing these rules generates the core structure of the resulting ontology. Afterwards, SPARQL update statements are executed to create additional connections in the generated ontology. These SPARQL statements are more complicated mapping rules, which could only be expressed with great difficulty using RML.

The validation function is built with the TopBraid SHACL API[3] and allows the validation of the generated ontology against a document containing SHACL shapes. The validation function checks all SHACL shapes and produces a SHACL validation report, which contains a boolean statement about the overall conformance as well as detailed validation results in case the model does not conform with one or more shapes.

5.2 Usage

AML2OWL offers two modes of operation: (1) as a command-line interface (CLI) that can be used to manually trigger mapping and validation and (2) as a library that can be easily integrated into other Java applications.

---

[2] RML-Mapper is an open source library to execute arbitrary RML rules. It is available at https://github.com/RMLio/rmlmapper-java.

[3] The TopBraid SHACL API is a powerful library that allows to validate RDF models against arbitrary SHACL shapes. It is availiable at https://github.com/TopQuadrant/shacl.

For library usage, AML2OWL is available as a Maven dependency, which can be easily integrated into your own Java projects built with Maven. First, add the following dependency:

```
<dependency>
    <groupId>io.github.aljoshakoecher</groupId>
    <artifactId>io.github.aljoshakoecher.aml2owl-lib</artifactId>
    <version>2.0.0</version>
</dependency>
```

Listing 11: Maven dependency for the AML2OWL mapper

Make sure to use the latest version (see https://github.com/hsu-aut/aml2owl/releases for a list of all releases). Afterwards, you can create an instance of the mapper to perform mappings, as illustrated in Listing 12.

```
import java.nio.file.Path;
import java.nio.file.Paths;
import org.apache.jena.rdf.model.Model;
import aml2owl.checking.ShaclValidator;

// Get a mapper instance
AmlOwlMapper mapper = new AmlOwlMapper();
Path amlFilePath = Paths.get("amlFile.aml");

// Map an AutomationML file and get an ontology instance (Apache Jena Model)
Model mappedModel = mapper.executeMapping(amlFilePath, null);
```

Listing 12: Example usage of the AML2OWL mapper in a Java program

The resulting `mappedModel` is an ontology conforming to the AutomationML ontology and the mapping rules defined in chapter 3. If you want to perform mapping and validation in one step, see the example shown in Listing 13.

```
import java.nio.file.Path;
import java.nio.file.Paths;
import org.apache.jena.rdf.model.Model;
import aml2owl.checking.ShaclValidator;

// Get a validator instance
ShaclValidator validator = new ShaclValidator();
Path amlFilePath = Paths.get("amlFile.aml");
Path shapePath = Paths.get("shaclShapeFile.ttl");

// Map an AutomationML file and validate the resulting ontology against a file containing SHACL shapes
Model report = validator.mapAndCheckConformance(amlFilePath, shapePath);
```

Listing 13: Example usage of mapping and validation in one step

The resulting report is a standard-compliant SHACL report object. If you want to quickly test whether the AutomationML model conforms with all shapes, you can use the utility function isConforming(model) of the ShaclReportUtil class.
More documentation as well as examples can be found at https://github.com/hsu-aut/aml2owl/.

For CLI usage, first download the latest version of the CLI application from https://github.com/hsu-aut/aml2owl/releases (for example, Aml2Owl-2.0.0.jar). Next, open a console or PowerShell window in the folder where the application was downloaded. You can then execute the mapping application with the command shown in Listing 14:

```
java -jar Aml2Owl-<version>.jar <command> <arguments>
```

Listing 14: CLI use of the AML2OWL application

In this command, the placeholder `<version>` should be replaced with the downloaded version number, e.g., 2.0.0. The placeholders <command> and <arguments> need to be replaced with one of the two commands below and the corresponding arguments, respectively.

The following two commands are provided:

- `map`: Automatically maps a given AutomationML file into an ontology. The map command requires one parameter: a path to an AutomationML file to be mapped. A complete command for v2.0.0 looks like this: java -jar .\Aml2Owl-2.0.0.jar map ".\test-file.aml".
  If successfully executed, this command will produce an ontology stored as a Turtle file with file name MappingOutput.ttl located right next to the jar.

- `map-and-validate`: Automatically maps a given AutomationML file into an ontology and validates the result against a given SHACL file. The map-and-validate command requires one parameter: a path to an AutomationML file to be mapped. A complete command for v2.0.0 looks like this: java -jar .\Aml2Owl-2.0.0.jar map-and-validate ".\test-file.aml" ".\test-shapes.ttl".
  In this case, test-file.aml is mapped and validated against all SHACL shapes contained in test-shapes.tll. If successfully executed, this command will produce a SHACL validation report stored as a Turtle file with file name ValidationOutput.ttl located right next to the jar.

5.3 Reference Document of exemplary SHACL Shapes

As mentioned above, more and more working groups are expected to give SHACL-Statements together with their BPRs/ ARs in the future. To help the development of these, a document with simple SHACL-Statements to copy and paste is given with this working group's ARs. Complicated statements will be written by SHACL experts, but easy cardinality- or existence-constraints are expected to be written by the working groups themselves. More of such standard-statements will be added as more recommendations get translated.

```
ex:InternalElementSubnetShape a sh:NodeShape ;
  sh:target [
    a sh:SPARQLTarget ;
    sh:select """
      PREFIX aml: <https://w3id.org/hsu-aut/AutomationML#>
      SELECT ?this
      WHERE {
        ?this a aml:InternalElement ;
          aml:hasRoleRequirement aml:[PATH_TO_THE_ROLE] .
      } """ ;
  ] ;
  sh:property [
    sh:path aml:hasInterface ;
    sh:class aml:[NAME_OF_INTERFACE] ;
    sh:maxCount 1 ;
    sh:message "InternalElement with [RoleRequirement] can't have more than one Interface.";
  ];
  sh:property  [ sh:class  <https://w3id.org/hsu-aut/AutomationML#[PATH_TO_THE_ATTRIBUTE]>;
    sh:maxCount  1;
    sh:minCount  1;
    sh:message   "Must have exactly one Instance of Attribute with [AttributeType].";
    sh:path      aml:hasAttribute;
  ].
```

Listing 15: Basic SHACL statements to copy and paste. Placeholders are written in square brackets and colored red.

A statement always consists of a selection of the class the constraint is generated for (the first block Listing 15). Here the name of the class the constraint is relevant for has to be filled in. The path has to be given in the same format the ontology generates it; it can be looked up by opening the ontology and clicking on the relevant class, if unsure, which name to give. An example for such a path would be "AutomationProjectConfigurationRoleClassLib%2FSubnet" for the class "Subnet".

After the selection, the property of the selected class is defined. In the current copy-paste-document two kinds of properties are given: Interfaces (hasInterface) and Attributes (hasAttribute). SHACL is not limited to these, but this working group expects these properties to be the most occurring.

In case of the hasInterface-property (second block in Listing 15), the path to the interface has to be given, as well as the path to the target class. These both are expected in the same format, as the class selection above (an example for an interface-path would be "CommunicationInterfaceClassLib%2FLogicalEndPoint" to model a logical endpoint). After that the constraint type has to be defined. The example-document gives a maxCount 1 and a minCount 1, to allow exactly one of the wanted Interface.

In case of has Attribute (third block in Listing 15), the path of the attribute has to be given. This path follows the same pattern as the selection of interfaces and classes (AttributeLib%2FMyAttributeType is an example for the selection of MyAttributeType). The same way, as is given in hasInterface, the copy-paste-document has a minCount 1/maxCount 1- pair as constraints to force exactly one Attribute of given kind.

6 Outlook

This working group lays the groundwork for machine readable and verifiable AutomationML Structures and as such, smart standards.

Through this working group a powerful addition to AutomationML is created. In the past, basic AutomationML-concepts like cardinality or the datatype of an attribute could be defined (a functionality which will continue in future versions). These were helpful but too limited for many use cases and not specific enough for the day-to-day use of AutomationML. The proposed ontology, along with SHACL functionalities, will complement the core AutomationML functionality, further enhancing the use of domain models by making them formalized and machine-readable. This will help users to understand the model more precisely and opens the door to topics like automatic semantic checks.

The ontology developed as part of this working group, while not the main goal, allows AutomationML to leverage the broader benefits of ontologies: Multiple domain-model ontologies could be connected into one unified AutomationML ontology. Users will be able to query this ontology to gain a deeper understanding of the domain models or combine elements from different models to create a new one that remains compatible with the existing ones. This would enhance connectivity between domain models and provide a more efficient way to store and manage the growing number of domain models.

While these benefits are significant, the work in this field is far from complete. Further refinement of the findings from this working group would be beneficial in the future:

The appended document describing a reference for common SHACL rules is a first approach, but it lacks usability, since users must manually search for the needed lines and copy and paste them into their document.

From a usability point of view, the generation of SHACL-rules could be offered via an add-on in the AutomationML Editor, enabling the working group leads to generate constraint-files in a sub-window while looking at the original AutomationML library. This would fall in line with the workflow of the AutomationML Editor as it is today and lower the user's hesitation to use SHACL.

In the future, efforts should be made to further help nonexperts with generating these rules. One approach could be an application enabling the user to write constraints through drag-and-drop application. Another approach would be the usage of LLMs [8] such as ChatGPT to create such rules. First research proved them to be successful in writing code [9] for different programming languages. Such an application would enable the creation of SHACL constraints through the input of natural language. As such, even complicated rules could be generated without the need to understand any SHACL syntax. That being said, at this point first tests with OpenAI's ChatGPT 4.0 as well as some casual language models such as Mistral didn't show great results. Further research is needed to enable the usage of these models for SHACL generation.

7 References

[1] ISO, "IEC/ISO SMART," 16 09 2022. [Online]. Available: https://www.iso.org/smart. [Accessed 16 06 2025].

[2] D. A. Czarny, G. Bülow, D. Lochner, C. Diedrich and J. Diemer, "scenarios for digitizing standardization and standards," 2021.

[3] AutomationML e.V., *Whitepaper AutomationML Edition 2.1 Part 1 - Architecture and General Requirements,* 2018.

[4] AutomationML e.V., *WP Part 2 - Semantics libraries, Also contains corresponding libraries,* 2022.

[5] C. Diedrich, K. Schneider, J. Hodges, D. Anicic, F. Jankowiak, E. S. Jeong and et al, "Semantic interoperability: challenges in the digital transformation age," IEC, 2019.

[6] A.Hogan, The Web of Data. 1st ed., Cham: Springer International Publishing; Imprint: Springer, 2020., 2020.

[7] W3C, "W3C Shapes Constraint Language (SHACL)," 20 07 2017. [Online]. Available: https://www.w3.org/TR/shacl/.

[8] P. Wrobel, "Generation of formal description for AutomationML using Large Language Models," 4 02 2025.

[9] N. Nguyen and S. Nadi, "An Empirical Evaluation of GitHub Copilot's Code Suggestions," *19th International Conference on Mining Software Repositories,* pp. 1-5, 2022.

[10] B. D. Meester, P. Heyvaert and T. Delva, "RDF Mapping Language (RML)," 20 06 2024. [Online]. Available: https://rml.io/specs/rml/. [Accessed 18 02 2025].

[11] M. Musen, "The Protégé project: A look back and a look forward. AI Matters. Association of Computing Machinery Specific Interest Group in Artificial Intelligence," 06 2015.

[12] T. Westermann, M. Ramonat, J. Hujer, F. Gehlhoff and A. Fay, "Automatic Mapping of AutomationML Files to Ontologies for Graph Queries and Validation," 2025. [Online]. Available: https://arxiv.org/abs/2504.21694.

8 Appendix

8.1 Formal mapping rules for AutomationML to OWL mapping

```
∀(A,B)
(Attribute(A) ∧ hasRefAttributeType(A,B))
∨ (Interface(A) ∧ hasRefBaseClass(A,B))
∨ (InternalElement(A) ∧ hasRefBaseSystemUnitClass(A,B))
⇒ rdf : type(A,B)
```

Listing 16: Formal mapping rule of Classes and Instances

```
∀(A,B)
(InternalElement(A) ∨ SystemUnitClass(A)) ∧ hasSupportedRoleClass (A,B)
⇒ rdf : type (A,B)
```

Listing 17: Formal mapping rule of Roles

```
∀(A,B)
(SystemUnitClass(A) ∨ InterfaceClass(A) ∨ RoleClass(A)) ∧ hasRefBaseClass(A,B)
⇒ rdfs : subClassOf(A,B)

∀(A,B)
AttributeType(A) ∧ hasRefAttributeType(A,B)
⇒ rdfs : subClassOf(A,B)
```

Listing 18: Formal mapping rules of Class Hierarchies

```
∀(A,B,C)
ExternalInterface(A) ∧ ExternalInterface(B)
∧ InternalLink(C)
∧ hasRefPartnerSideA(C,A)
∧ hasRefPartnerSideB(C,B)
⇒ isLinked(A,B)
```

Listing 19: Formal mapping of InternalLinks

```
∀(A,B)
InternalElement(A) ∧ InternalElement(B)
∧ hasRefBaseSystemUnitClass(A,B)
⇒ (hasMasterObject(A,B)
∧ hasMirrorObject(B,A))
```

Listing 20: Formal mapping of Mirror-Objects

```
∀(A,B,C,D,E,F)
InternalElement(A) ∧ RoleClass(B)
∧ Attribute(C) ∧ Attribute(D)
∧ hasAttribute(A,C) ∧ hasAttribute(B,D)
∧ hasName(C,E) ∧ hasName(D,F)
∧ hasRoleRequirement(A,B) ∧ Equal(E,F)
⇒ hasMappingObject(C,D)


∀(A,B,C,D,E,F)
InternalElement(A) ∧ RoleClass(B)
∧ Interface(C) ∧ Interface(D)
∧ hasInterface(A,C) ∧ hasInterface(B,D)
∧ hasName(C,E) ∧ hasName(D,F)
∧ hasRoleRequirement(A,B) ∧ Equal(E,F)
⇒ hasMappingObject(C,D)
```

Listing 21: Formal mappings of Mapping-Objects

```
∀(A,B)
(InternalElement(A) ∨ SystemUnitClass(A))
∧ InternalElement(B) ∧ hasPart(A,B)
∧ hasRoleRequirement(B, Facet)
⇒ Facet(B) ∧ hasFacet(A,B)


∀(A,B)
(InternalElement(A) ∨ SystemUnitClass(A))
∧ InternalElement(B) ∧ hasPart(A,B)
∧ hasRoleRequirement(B, Group)
⇒ Group(B) ∧ hasGroup(A,B)
```

Listing 22: Formal mappings of Facets and Groups

```
∀(A,B,C,D,E)
RoleClass(A) ∧ RoleClass(B)
∧ hasRefBaseClass(A,B)
∧ hasAttribute(B,C) ∧ hasName(C,D)
∧ ¬(∃E (hasAttribute(A,E) ∧ hasName(E,D)))
⇒ hasAttribute(A,C)
```

Listing 23: Formal mapping of Inheritance and Subcomponents