

Interpreting OWL Complex Classes in AutomationML based on Bidirectional Translation

Yingbing Hua*, Björn Hein†
Faculty of Informatics
Karlsruhe Institute of Technology
Karlsruhe, Germany
Email: *yingbing.hua@kit.edu, †bjoern.hein@kit.edu

Abstract—The World Wide Web Consortium (W3C) has published several recommendations for building and storing ontologies, including the most recent OWL 2 Web Ontology Language (OWL). These initiatives have been followed by practical implementations that popularize OWL in various domains. For example, OWL has been used for conceptual modeling in industrial engineering, and its reasoning facilities are used to provide a wealth of services, e.g. model diagnosis, automated code generation, and semantic integration. More specifically, recent studies have shown that OWL is well suited for harmonizing information of engineering tools stored as AutomationML (AML) files. However, OWL and its tools can be cumbersome for direct use by engineers such that an ontology expert is often required in practice. Although much attention has been paid in the literature to overcome this issue by transforming OWL ontologies from/to AML models automatically, dealing with OWL complex classes remains an open research question. In this paper, we introduce the *AML concept models* for representing OWL complex classes in AutomationML, and present algorithms for the bidirectional translation between OWL complex classes and their corresponding AML concept models. We show that this approach provides an efficient and intuitive interface for non-experts to visualize, modify, and create OWL complex classes.

I. INTRODUCTION

The World Wide Web Consortium (W3C) has published several recommendations for building ontologies, with the Resource Description Framework (RDF) and the Web Ontology Language (OWL) being the most popular ones. OWL was designed as an extension of RDF with significant more expressivity and is preferred as a language for conceptual modeling in complex domains. The reasoning facilities of OWL can, therefore, be used to support decision making in the domain of interest.

The Automation Markup Language (AutomationML, or AML) is a neutral, XML-based data format for data exchange between engineering tools [1]. AML is standardized as IEC 62714 and has its root in the data format CAEX (IEC 62424). AML supports the modeling of plant topology, component structure, geometry and kinematics, logic behavior, and communication networks. However, AML per se does not provide a formal semantics for automated data interpretation [2]. In practice, tools need to achieve a common understanding of the data and be responsible for the preservation of semantics.

Efforts have been made on adopting OWL and its reasoning facilities for processing AML data. The typical approach comprises three steps: a) transform engineering data stored in an AML document to an AML ontology by explicitly define the semantics of AML notions; b) after communication with the domain experts, an ontology expert extends the AML ontology with additional knowledge for specific engineering purposes; c) utilizing the reasoner for providing advanced engineering services. For example, with predefined ontological descriptions about error types in plant models, Abele et al. were able to identify modeling errors in the plant topology [3]. Hua et al. proposed a model-driven robot programming approach that is capable of inferring component capability and the associated programming interfaces from AML models [4]. In this paper, we use the term *AML ontology* to indicate an OWL ontology that is converted from an AML document.

It is evident that the approaches mentioned above are based on sophisticated domain knowledge that is modeled as OWL complex classes by nesting logic-based OWL constructors. Therefore, a profound understanding of the domain and the language OWL is required. Recently, Hildebrandt et al. proposed the *domain expert-centric* approach for building ontologies of cyber-physical systems [5]. While this approach tackles the problem of incorporating domain expert's knowledge, it is unclear yet how to deal with OWL complex classes. In the remainder of the paper, we use the term *OWL complex class* and the term *OWL class* interchangeably if the context is clear.

In this paper, we introduce the *AML concept model* for representing ontological semantics in native AML models. Based on a bidirectional translation procedure between OWL and AML, OWL classes can be visualized as AML concept models for inspection and modification, and *proper* AML concept models can be transformed to OWL classes while preserving the ontological semantics. We show that this approach demonstrates an efficient and intuitive interface for a non-expert to interact with OWL complex classes.

This paper is organized as follows. Section II discusses related work on model transformation between OWL and AML. Section III gives a brief overview of OWL and AML, and introduces the important notions used in this paper. In section IV we present the *AML concept model* that is developed for preserving ontological semantics in AML. In section V we describe the bidirectional translation between OWL complex

arXiv:1906.04240v1 [cs.AI] 4 Jun 2019

classes and AML concept models. Finally, we demonstrate the utility of this approach with two typical use cases of ontology engineering in section VI and conclude the paper with future works in section VII.

II. RELATED WORK

The first result about converting AML to OWL appeared in 2009 by Runde et al. in their German paper [6]. Two approaches were proposed and discussed. The *abstract approach* represents the CAEX vocabulary directly as OWL classes in the ontology and transforms CAEX classes, objects and attributes as individuals of these OWL classes. The *concrete approach* generates an OWL class for each CAEX class with an annotation about its original type in the CAEX schema. For example, an AML role class Robot will be converted to an OWL class with the annotation RoleClass. Subsequent researches generally follow either the abstract or the concrete approach. For example, Kovalenko et al. proposed a lightweight ontology for covering core concepts of CAEX using the abstract approach [7], while Hua et al. followed the concrete approach for learning unknown engineering concepts from AML data [2].

The backward transformation from OWL to AML is less studied, although the first approach was already published in 2010 in [8]. The transformation begins with mapping atomic OWL classes to appropriate CAEX classes using the CAEX type annotation of each OWL class. It proceeds with OWL individuals of the top level OWL classes and transforms them into proper CAEX objects. Then the transformation handles each property associated with the individuals until all information in OWL is processed.

It is evident that existing methods only target at "simple" knowledge types, that is, atomic classes, objects, and properties. For handling complex ontological knowledge, e.g. OWL complex classes, one challenge arises that no regular AML model can preserve complex ontological semantics.

In the remainder of the paper, we assume that we are given an AML ontology converted from an AML document following the approach proposed in [2]. Our goal is to develop a modeling approach that enhances native AML models with ontological semantics and a translation procedure between such native AML models and OWL complex classes.

III. PRELIMINARIES

A. AutomationML

AML data is stored in an XML document which conforms to the underlying CAEX XML schema. An AML document usually contains a set of class libraries and a structured collection of engineering objects that represents the plant topology. We emphasize the following core concepts of CAEX that we consider in this paper:

- **Role class (RC):** a role class refers to a type of engineering objects, e.g. Robot. As AML follows the object-oriented paradigm, role classes can be organized in inheritance hierarchies within so-called *role class libraries*.

- **Interface class (IC):** an interface class represents a type of engineering interfaces, e.g. SignalInterface or AttachmentInterface. Similar to role classes, inheritance is allowed between interface classes and an *interface class library* stores a set of interface classes.
- **Internal element (IE):** an internal element is the model of an engineering object, e.g. a joint inside a robot or a real robot in the plant. By referring to a role class, the meaning of an internal element is declared. For describing the plant topology, internal elements are organized as tree structures in the *instance hierarchy*.
- **External interface (EI):** an external interface is the model of an engineering interface, e.g. an IO pin of a controller. The type of an external interface is defined by referring to an interface class.
- **System unit class (SUC):** a system unit class is a reusable engineering template that contains an internal structure, where internal elements are used to represent individual parts of the structure.

For all the concepts mentioned above, CAEX attributes can be defined to describe their properties. In the rest of the paper, we use the notion *AML model* to refer to any XML model that can be generated according to the CAEX schema.

B. OWL

OWL¹ belongs to the family of expressive Description Logics (DL) and is closely related to *SROIQ* [9]. An OWL ontology defines a finite set of classes (e.g. Robot), individuals (e.g. a robot instance) and properties in a domain of discourse, and describes relations between these artifacts. One further distinguishes between *object properties* and *data properties*. The former one is used for relations between individuals (e.g. a robot has a controller), and the latter one is for describing the concrete qualities of an individual (e.g. weight of a robot). Although an AML ontology generated by [2] has merely two object properties hasIE/hasEI, we also consider the following inverse properties in this paper:

$$\text{isIEOf} \equiv \text{hasIE}^{-}, \text{isEIOf} \equiv \text{hasEI}^{-}$$

An OWL class is either an atomic class or a complex one when it is generated by so-called concept constructors [9]. Table I shows the concept constructors of OWL, their correspondences in the terminology of DL, their DL syntax², and their formal model-theoretic semantics in OWL. We use conventional notions for the syntax: *A* represents an atomic class, *C* or *D* stands for an OWL (complex) class, *R* stands for an OWL property, *a* or *b* stands for an OWL individual, *DR* is used for the data range of data properties, and *lt* is used for a literal value. The nested OWL class *C* inside a restriction e.g. $\exists R.C$ is called the *filler* of the restriction.

In this paper, we consider an OWL complex class constructed by using arbitrarily many of the constructors in the

¹While OWL is the short name of the Web Ontology Language whose expressive power goes beyond the scope of description logics, we use this notion to refer to the specific sub-language OWL 2 DL.

²Please refer to [10] for more details of the DL syntax.

TABLE I
SYNTAX AND SEMANTICS OF OWL CONSTRUCTORS

	OWL Terminology	DL Terminology	DL Syntax	Semantics
covered in this paper	atomic class	atomic concept	A	$A^{\mathcal{I}}$
	Thing	top concept	\top	$\Delta^{\mathcal{I}}$
	Nothing	bottom concept	\perp	\emptyset
	ObjectIntersectionOf	intersection	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
	ObjectUnionOf	union	$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
	ObjectOneOf	nominal	$\{a, b, \dots\}$	$\{a^{\mathcal{I}}, b^{\mathcal{I}}, \dots\}$
	ObjectSomeValuesFrom	existential restriction	$\exists R.C$	$\{x \mid \exists y. (x, y) \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$
	DataSomeValuesFrom	existential restriction	$\exists R.(DR)$	$\{x \mid \exists y. (x, y) \in R^{\mathcal{I}} \wedge y \in (DR)^{\mathcal{I}}\}$
	ObjectExactCardinality	exact restriction	$= nR.C$	$\{x \mid \{y \mid (x, y) \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\} = n\}$
	ObjectMinCardinality	at-least restriction	$\geq nR.C$	$\{x \mid \{y \mid (x, y) \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\} \geq n\}$
	ObjectMaxCardinality	at-most restriction	$\leq nR.C$	$\{x \mid \{y \mid (x, y) \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\} \leq n\}$
	ObjectHasValue	fills restriction	$\exists R.\{a\}$	$\{x \mid (x, a^{\mathcal{I}}) \in R^{\mathcal{I}}\}$
	DataHasValue	fills restriction	$\exists R.\{lt\}$	$\{x \mid (x, (lt)^{\mathcal{I}}) \in R^{\mathcal{I}}\}$
	ObjectAllValuesFrom	universal restriction	$\forall R.C$	$\{x \mid \forall y. (x, y) \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$
ObjectComplementOf	complement	$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$	
uncovered in this paper	DataExactCardinality	exact restriction	$= nR.(DR)$	$\{x \mid \{y \mid (x, y) \in R^{\mathcal{I}} \wedge y \in (DR)^{\mathcal{I}}\} = n\}$
	DataAllValuesFrom	universal restriction	$\forall R.(DR)$	$\{x \mid \forall y. (x, y) \in R^{\mathcal{I}} \rightarrow y \in (DR)^{\mathcal{I}}\}$
	DataMinCardinality	at-least restriction	$\geq nR.(DR)$	$\{x \mid \{y \mid (x, y) \in R^{\mathcal{I}} \wedge y \in (DR)^{\mathcal{I}}\} \geq n\}$
	DataMaxCardinality	at-most restriction	$\leq nR.(DR)$	$\{x \mid \{y \mid (x, y) \in R^{\mathcal{I}} \wedge y \in (DR)^{\mathcal{I}}\} \leq n\}$
	ObjectHasSelf	local reflexivity	$\exists R.Self$	$\{x \mid (x, x) \in R^{\mathcal{I}}\}$

covered part of Table I, and assume that no constructors in the *uncovered* part would appear. While it seems to be a strong assumption, we argue that: a) since CAEX attributes are mapped to data properties and one CAEX attribute is usually assigned to an object only once, the universal, at-least and at-most restrictions on data properties can be omitted; b) the local reflexivity cannot appear in an AML ontology, since we can not have an internal element (or external interface) which is the internal element (or external interface) of itself.

Formula 1 shows some examples of OWL classes used in this paper. Class A refers to Robots without any internal element of the type \neg IOController. Class B refers to internal elements of a Robot from the manufacturer KUKA. Class C refers to Robots with an IOController that has at least three IOInterfaces. Class D refers to IOInterfaces from objects that have at least three IOInterfaces. Apparently, as the complexity grows, the intended meaning of an OWL complex class becomes more difficult to understand. In the next section, we introduce the *AML concept model* that is able to represent OWL complex classes as native AML models.

$$\begin{aligned}
A &\equiv \text{Robot} \sqcap \neg \exists \text{hasIE}.(\neg \text{IOController}) \\
B &\equiv \exists \text{isIEOf}.(\text{Robot} \sqcap \text{hasManufacturer.}''\text{KUKA}'') \\
C &\equiv \text{Robot} \sqcap \exists \text{hasIE}.(\text{IOController} \sqcap \geq 3 \text{hasEI.IOInterface}) \\
D &\equiv \text{IOInterface} \sqcap \exists \text{isEIOf}.(\geq 3 \text{hasEI.IOInterface}) \quad (1)
\end{aligned}$$

IV. THE AML CONCEPT MODEL

Consider the OWL class constructors in Table I. It is evident that while atomic classes can be represented as AML classes directly [2], most of the features in OWL are not supported by AML. Therefore, we propose the following approach to represent OWL class constructors as dedicated AML models:

Atomic class: similar to the conventional translation procedure as proposed by [8], an atomic class is represented by a CAEX role or interface class. A class reference in CAEX is

therefore equivalent to a class assertion in OWL. For example, an internal element a of the role class A is represented as $A(a)$.

Thing: Thing is the most general concept in OWL and contains all individuals. Therefore, it is represented by a CAEX object with no specific configurations.

Nothing: Nothing is the most specific concept in OWL and contains no individual. Nothing is handled as the complement of Thing (see the complement case below).

Intersection: an intersection $C \sqcap D$ contains individuals that are instances of all the operands C and D in the intersection. Therefore, an intersection is represented by the composition of several AML models that correspond to each of the operands, including CAEX class references, attributes, and subordinate object structures.

Union: a union $C \sqcup D$ contains individuals that are instances of at least one operand C or D of the union. XML does not support unions in general. In this paper, we handle each operand of a union separately and generate one AML model for each of them.

Nominal: a nominal $\{a, b, \dots\}$ enumerates all individuals that an OWL class shall contain. Similar to the union constructor, nominals cannot be directly represented in XML, and we generate one AML model for each element inside a nominal.

Existential restriction: an existential restriction $\exists R.C$ or $\exists R.(DR)$ states the existence of the relation R with the filler C or the data range DR . If R is an object property, the existential restriction is represented by a child object (internal element or external interface) while the filler C is represented by the model of the child object. If R is a data property, the existential restriction is represented by a CAEX attribute while the data range DR is represented by the configuration of the CAEX attribute, e.g. data type and value requirements.

Object cardinality restrictions: an object cardinality restriction, i.e. an exact restriction $= nR.C$, an at-least restriction $\geq nR.C$, or an at-most restriction $\leq nR.C$, defines the

number of child objects of the class C w.r.t. the relation R . The CAEX attributes $minCardinality$ and $maxCardinality$ are added to the child objects to represent the minimum and maximum number respectively. The exact cardinality of n is represented by $minCardinality = maxCardinality = n$.

Fills restriction: a fills restriction $\exists R.\{a\}$ or $\exists R.\{lt\}$ corresponds to an existential restriction with a Singleton filler. If R is an object property, the CAEX attribute $isIdentifiedByID$ is used to restrict the ID of the child object, as ID is unique in AML. If R is a data property, lt is set as the required value of the corresponding CAEX attribute.

Universal restriction: a universal restriction $\forall R.C$ forces all child objects w.r.t. the relation R to be instances of the class C . For example, $\forall hasIE.C$ describes things that have internal elements of type C only. While universal restrictions can not be directly represented in XML, it can be simulated by disallowing child objects that are instances of the class $\neg C$ [11] using the exact cardinality $= 0 R.(\neg C)$.

Complement: a complement $\neg C$ contains all individuals that are not instances of C . Since an OWL class can have arbitrarily nested complements, we first transform an OWL class to its negation normal form (NNF) so that complements are only bound to atomic classes [12]. For example, the NNF of the OWL class A in Formula 1 is:

$$NNF(A) \equiv Robot \sqcap \forall hasIE.IOController$$

Obviously, $NNF(A)$ does not contain any complements. In fact, complements can only appear in the following three cases in the NNF of an OWL class:

- A complement can be bound to an atomic class as $\neg A$ or a data range as $\neg DR$, and is not part of any restrictions. In this case, a CAEX attribute $negated=true$ is added to the AML model. Note that intersections of a mixture of positive and negative atomic classes, e.g. $\neg A_1 \sqcap A_2$, cannot be modeled in AML.
- A complement can be the filler of an existential restriction, i.e. $\exists R.(\neg A)$ or $\exists R.(\neg DR)$. As with the existential restriction, a child CAEX object or CAEX attribute is first generated. Then the CAEX attribute $negated=true$ is added to the child model.
- A complement can be the filler of a universal restriction as $\forall R.(\neg A)$ (recall that we ignore universal restrictions on data properties). In this case, we disallow child objects of the class A w.r.t. the relation R , which can be expressed using the exact cardinality $= 0 R.A$.

Table II summarizes the introduced CAEX attributes that are used to capture the semantics of OWL constructors mentioned above. We call them *concept attributes*. The attribute *primary* is a helper flag to indicate which element in an AML model is described by the OWL class. We call an AML model with concept attributes as an *AML concept model* and enumerate the values of concept attributes based on possible forms of NNF in Table III. Intuitively, AML concept models can be nested to represent nested OWL class expressions. An AML concept model is *proper* if it has exactly one primary element.

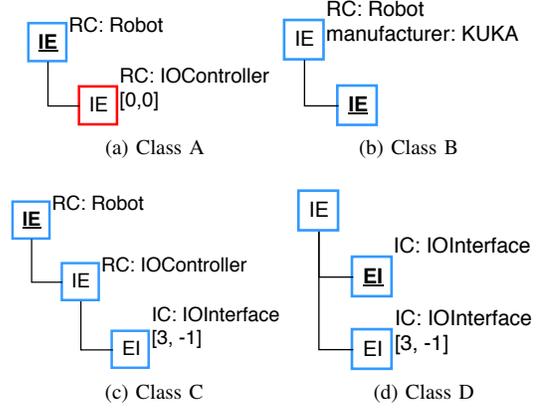


Fig. 1. The AML concept models for the OWL classes in Formula 1.

Note that intersections, unions, and nominals are omitted in the mapping since we handle each element of them individually.

Figure 1 illustrates the AML concept models of the NNF of the OWL classes A, B, C and D in Formula 1 as tree structures. Internal elements (IE) and external interfaces (EI) are represented by tree nodes, and their class references and attributes are depicted as labels on the top right corner. A negated object is marked as red. The primary object is marked as bold with an underline. Numbers in square brackets are the min and max cardinality of the object, while a value -1 means that it is unlimited. Note that for the classes B and D, the primary object is not the root node since XML cannot depict "part-of" relations (i.e. $isIEOf$, $isEIOf$). Therefore, each inverse property is simulated as a predecessor node in the XML tree.

V. TRANSLATION BETWEEN OWL AND AML

The core idea of the translation is to exploit the **tree structure** of OWL class expressions. More concretely, we introduce *AML concept trees* that depict OWL complex classes in a tree structure similar to AML concept models. Then we describe the *forward* translation $TransF : OWL \mapsto AML$ via the AML concept trees. Finally, we show that the *backward* translation $TransB : AML \mapsto OWL$ can be directly carried out using the mappings in Table III.

A. From OWL to AND-tree

We define a tree conventionally as a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where \mathcal{V} is a finite set of nodes and \mathcal{E} is a finite set of edges, to which the following rules apply:

- A tree \mathcal{G} has a unique *root* node that has no predecessor.
- Each node $n \in \mathcal{V}$ has a unique predecessor.

We call *leaf nodes* the tree nodes that have no successor, i.e. at the bottom of the tree. Furthermore, a *branching node* is an inner tree node that has a unique predecessor and arbitrarily many successors. Based on these notions, an *AND-tree* is a tree with the following properties:

- The root of an AND-tree represents the expression of an OWL complex class.

TABLE II
THE AML CONCEPT ATTRIBUTES FOR CAPTURING ONTOLOGICAL SEMANTICS.

Name	Type	Default	Designation	Semantics in OWL
negated	bool	false	whether the class reference or the data range of an AML concept model shall be negated	complement
minCardinality	integer	1	minimum number of occurrence of this AML concept model	minCardinality
maxCardinality	integer	unlimited	maximum number of occurrence of this AML concept model	maxCardinality
identifiedByID	bool	false	whether the ID of the AML concept model is used as an individual name in OWL	nominal
primary	bool	false	whether this AML concept model is the primary object	target individuals

TABLE III
MAPPING BETWEEN OWL CONSTRUCTORS AND AML CONCEPT ATTRIBUTES.

OWL Class Expression	Negated	minCardinality	maxCardinality
simple complement $\neg C$ or $\neg DR$	true	1	unlimited
existential restriction $\exists R.C$ or $\exists R.DR$	false	1	unlimited
existential restriction $\exists R.(¬C)$ or $\exists R.(¬DR)$	true	1	unlimited
universal restriction $\forall R.C$	true	0	0
universal restriction $\forall R.¬C$	false	0	0
at-least restriction $\geq nR.C$	false	n	unlimited
at-most restriction $\leq nR.C$	false	0	n

Algorithm 1 Construct

Input: The class expression ce of an OWL class C

Output: A tree node $root$

- 1: make a tree node $root$ for ce
- 2: **if** ce is an atomic class **then**
- 3: **return** $root$
- 4: **else if** (ce is an intersection) **then**
- 5: **for** $operand \in ce$ **do**
- 6: let $child = \text{Construct}(operand)$
- 7: add $child$ as a successor to $root$
- 8: **end for**
- 9: **else if** (ce is a restriction) **then**
- 10: let $child = \text{Construct}(ce.filler)$
- 11: add $child$ as a successor to $root$
- 12: **end if**
- 13: **return** $root$

- Each branching node of an AND-tree represents either an intersection or a restriction (see the notions in Table I).
- Each leaf node of an AND-tree represents either OWL Thing, OWL Nothing or an atomic class.

For each OWL complex class without unions and inverse properties, an AND-tree can be constructed by making a successor node for each operand of an intersection and the filler of a restriction, as shown in Algorithm 1.

We illustrate the construction process in Figure 2. Each box represents a tree node, and the number on the upper left corner of each box shows the sequence of node construction. The root node of the AND-tree corresponds to the OWL class D in Formula 1. Since the root is an intersection, the algorithm will handle each operand of it individually through line 4 to 6. The atomic operand `IOInterface` is returned directly and added as a child to the root in line 7. For the complex operand `$\exists isEIOf.(\geq 3hasEI.IOInterface)$` , the algorithm recursively generates sub-nodes until the final atomic filler `IOInterface` is reached in line 10. Note that all nodes are

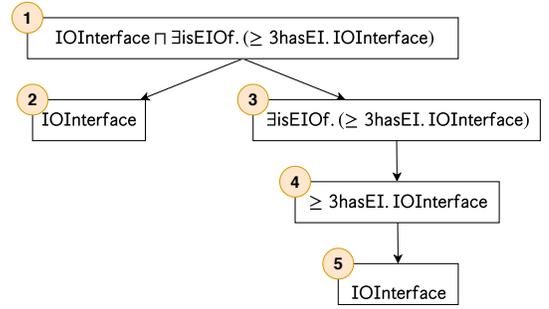


Fig. 2. The AND-tree constructed from the OWL class D in Formula 1. The numbers in the tree nodes show the sequence of node construction.

generated immediately in line 1 when `Construct` is called.

It becomes more involved if the OWL class C contains any disjunctions (unions or nominals) because XML does not support *or* statements generally. The solution is to construct m AND-trees for a disjunction with m elements. However, since disjunctions can appear in any nested part inside an OWL class expression, we need to traverse the logical structure of the class expression to produce a set of AND-trees that is logically equivalent to the OWL class.

Algorithm 2 shows the AND-tree construction process for classes involving disjunctions. If the input class expression ce is a disjunction, then a set of tree nodes are generated for the elements of the disjunction (line 4). In case the input is an intersection, the recursive call of `ConstructD` in line 12 will handle possible nested disjunction in each element and produce a set of nested trees. These nested trees need to be multiplexed with the existing trees in *roots* through line 13 to 15. The algorithm treats restrictions similarly to intersections despite that the filler of a restriction is used to produce nested trees in line 19. It is worth noting that only $m - 1$ copies of *root* are made in line 14 and 21 since the original *root* also counts during the construction.

Figure 3 illustrates the tree construction process of the OWL

Algorithm 2 ConstructD

Input: The class expression ce of an OWL class C **Output:** A set of tree nodes $roots$

```
1: initialize  $roots = \{\}$ 
2: if  $ce$  is an union or a nominal then
3:   for each  $element$  in  $ce$  do
4:     add ConstructD( $element$ ) to  $roots$ 
5:   end for
6: else
7:   make a tree node  $n$  for  $ce$ , add  $n$  to  $roots$ 
8:   if  $ce$  is an atomic class then
9:     return  $roots$ 
10:  else if ( $ce$  is an intersection) then
11:    for  $operand \in ce$  do
12:      let  $nestedTrees = ConstructD(operand)$ 
13:      for  $root$  in  $roots$  do
14:        copy  $root$   $nestedTrees.size - 1$  times
15:        add the root of each  $tree \in nestedTrees$  as a
16:        successor to exactly one copy of  $root$ 
17:      end for
18:    end for
19:  else if ( $ce$  is a restriction) then
20:    let  $nestedTrees = ConstructD(ce.filler)$ 
21:    for  $root$  in  $roots$  do
22:      copy  $root$   $nestedTrees.size - 1$  times
23:      add the root of each  $tree \in nestedTrees$  as a
24:      successor to exactly one copy of  $root$ 
25:    end for
26:  end if
27: end if
28: return  $roots$ 
```

class $Robot \sqcap \exists hasIE.(IOController \sqcup IODevice)$. In the first step, a root node is generated that contains the complete class expression (line 7). Then, for each operand of the intersection, a child node is generated in step 2 and 3 (line 12). Since the Robot node is atomic, no further construction is required in the recursive call (line 9). On the other hand, the restriction node $\exists hasIE.(IOController \sqcup IODevice)$ is copied in step 4 (line 21), since its filler is a union and produces two atomic nodes IOController and IODevice (line 19). In step 5 and 6, the atomic nodes are added to the original and copied restriction nodes (line 22). Finally, the root node $Robot \sqcap \exists hasIE.(IOController \sqcup IODevice)$ is copied once to accept the two distinct restriction nodes in step 7 (line 14-15).

B. Working with Inverse Properties

For OWL classes that describe objects in the instance hierarchy, inverse properties might appear for gathering information about their ancestors or siblings (see the OWL classes B and D in Formula 1). Due to structural restrictions in AML, we assume that the following conditions hold when an inverse property $R^- \in \{isEIOf, isEIOf\}$ appear:

C1: R^- does not appear in the filler of any restriction that has R as property, e.g. $\exists R.(\exists R^-.C)$.

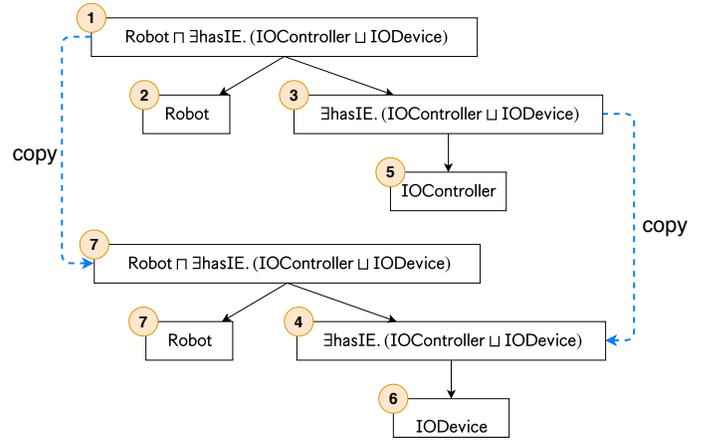


Fig. 3. The tree construction process of the OWL complex class $Robot \sqcap \exists hasIE.(IOController \sqcup IODevice)$. The numbers in the tree nodes show the sequence of node construction.

C2: R^- does not appear in the filler of cardinality restrictions, e.g. $\geq n R^-.C$.

C3: R^- does not appear in the filler of any restriction that has a different property $R' \neq R$, e.g. $\exists R'.(\exists R^-.C)$.

C4: $isEIOf$ does not appear in the filler of any restrictions that has an inverse property, e.g. $\exists R^-.(\exists isEIOf.C)$

The conditions C1 and C2 avoid modeling redundancies in OWL, since AML data has a tree structure, and each node in the tree has a unique predecessor. A class expression $\exists R.\exists R^-.C$ is therefore logically equivalent to C , and a cardinality restriction is redundant to an existential restriction. The condition C3 avoids modeling errors in OWL since the set of internal elements is disjoint with the set of external interfaces. The condition C4 holds since external interfaces have no child object in AML. We call an OWL class that meets the conditions C1-C4 as a *proper AML class*.

The inverse properties of a proper AML class always appear continuously at the outermost layer of the class expression. In other words, the AND-tree of a proper AML class has all inverse properties in the upper part of the tree. Therefore, Algorithm 3 iteratively removes the inverse properties from the root of an AND-tree. We call an AND-tree that contains no disjunctions nor inverse properties as an *AML concept tree*.

Figure 4 shows how the inverse property in the root of class D's AND-tree is removed. Since the original root node is an intersection, the algorithm first constructs a template node for the new root (line 11). Then a new child node is constructed for the previous child IOInterface by formulating an existential restriction in step 2 (line 14 to 15). To keep the consistency of the tree, the expression of the new child node is added to the new root node in the third step (line 16). For the previous child $\exists isEIOf.(\geq 3 hasEI.IOInterface)$ with the inverse property $isEIOf$, the filler $\geq 3 hasEI.IOInterface$ is added to the new root node as a conjunctive term in step 4 (line 18), and the corresponding grandchild with its sub-tree is added as a child to the new root in step 5 (line 19).

It is obvious that the inverse property $isEIOf$ is now re-

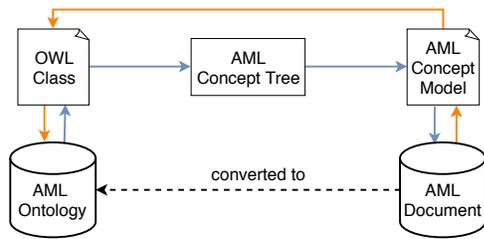


Fig. 5. The work flow for ontology engineering using bidirectional translation.

In the first use case (orange arrows in Figure 5), the required OWL class does not exist yet, and a user wants to create an AML concept model for the concept in mind:

- 1) The user generates the primary AML concept model for the target concept, i.e. a CAEX role class, system unit class, interface class, internal element or external interface with class reference and concept attributes.
- 2) The user adds CAEX attributes and sub-elements with sufficient constraints to the model. This process repeats recursively for nested attributes and sub-elements.
- 3) If the primary AML concept model shall be further restricted by the properties of its predecessor or siblings, a parent AML concept model is generated. This process repeats recursively for further predecessors and siblings.
- 4) The user generates the OWL class using the backward translation and adds it to the AML ontology.

The second use case (blue arrows in Figure 5) refers to the ontology evolution procedure in which OWL complex classes already exist in an AML ontology. These classes might be modeled by an ontology expert or created using the AML editor as described above. Now the user might want to inspect a particular OWL class and modify it by demand. First, the chosen OWL class is translated into AML concept models via its AML concept trees. Then, the user can open the generated AML concept models in the AML editor and inspect them by browsing their XML structure. If any modification is necessary, the user can edit the AML concept models as described above and export the new one to an OWL class.

In conclusion, AML concept models can be inspected and modified using the AML editor, while the forward and backward translation are transparent to the user. By comparing the OWL complex classes in Formula 1 and their corresponding AML concept models in Figure 1, we believe that the two use cases demonstrate an intuitive and efficient interaction with OWL complex classes. Because the forward and backward translations are inverse functions of each other (see Formula 2 in section V-D), Figure 5 also illustrates that a round-trip engineering of OWL complex classes is possible by following the work flow of both use cases successively.

VII. CONCLUSION

In this paper, we studied the problem of interpreting OWL complex classes from an AML ontology. We identified the inadequacy of existing approaches and introduced a native AML based approach for visualizing, editing and creating

OWL complex classes. More specifically, we presented the AML concept model that is capable of carrying ontological semantics, and a bidirectional translation procedure for the conversion between OWL complex classes and AML concept models. With two typical use cases in ontology engineering, we demonstrated the utility of the proposed approach.

Future works are considered in two aspects. First, the semantic expressivity of the AML concept model is restricted by the object properties `hasIE`, `hasEI` and can be extended to cover further modeling facilities in AML, e.g. connections between objects. Second, the current implementation does not provide a friendly user interface and can be improved by integrating the translation procedure into the AML Editor.

ACKNOWLEDGMENT

This work has been supported from the European Unions Horizon 2020 research and innovation programme under grant agreement No 688117 Safe human-robot interaction in logistic applications for highly flexible warehouses (SafeLog).

REFERENCES

- [1] R. Drath, A. Lüder, J. Peschke, and L. Hundt, "Automationml - the glue for seamless automation engineering," in *2008 IEEE International Conference on Emerging Technologies and Factory Automation*, Sept 2008, pp. 616–623.
- [2] Y. Hua and B. Hein, "Concept Learning in AutomationML with Formal Semantics and Inductive Logic Programming," in *2018 IEEE International Conference on Automation Science and Engineering (CASE)*, 2018.
- [3] L. Abele, C. Legat, S. Grimm, and A. W. Müller, "Ontology-based Validation of Plant Models," in *2013 11th IEEE International Conference on Industrial Informatics (INDIN)*, July 2013, pp. 236–241.
- [4] Y. Hua, S. Zander, M. Bordignon, and B. Hein, "From Automationml to ROS: A Model-driven Approach for Software Engineering of Industrial Robotics using Ontological Reasoning," in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, Sep. 2016, pp. 1–8.
- [5] C. Hildebrandt, S. Trsleff, B. Caesar, and A. Fay, "Ontology Building for Cyber-Physical Systems: A domain expert-centric approach," in *2018 IEEE 14th International Conference on Automation Science and Engineering (CASE)*, Aug 2018, pp. 1079–1086.
- [6] S. Runde, K. Güttel, and A. Fay, "Transformation von CAEX-Anlagenplanungsdaten in OWL: Eine Anwendung von Technologien des Semantic Web," in *Automation 2009, Der Automatisierungskongress in Deutschland, VDI/VDE-Gesellschaft Mess- und Automatisierungstechnik (GMA)*, Jun 2009, pp. 175–178.
- [7] O. Kovalenko, M. Wimmer, M. Sabou, A. Lder, F. J. Ekapatra, and S. Biffl, "Modeling AutomationML: Semantic Web technologies vs. Model-Driven Engineering," in *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*, Sep. 2015, pp. 1–4.
- [8] S. Runde, A. Fay, and S. Böhm, "Konvertierung von OWL-Planungsergebnissen nach CAEX," in *Automation 2010, Der 11. Branchentreff der Mess- und Automatisierungstechnik, VDI/VDE-Gesellschaft Mess- und Automatisierungstechnik (GMA)*, Jun 2010, pp. 1–12.
- [9] B. Motik, P. F. Patel-Schneider, and B. C. Grau, "OWL 2 Web Ontology Language Direct Semantics (Second Edition)," 11.12.2012. [Online]. Available: <https://www.w3.org/TR/owl2-direct-semantics/>
- [10] M. Krötzsch, F. Simancik, and I. Horrocks, "A Description Logic Primer," *CoRR*, vol. abs/1201.4089, 2012. [Online]. Available: <http://arxiv.org/abs/1201.4089>
- [11] B. Motik, P. F. Patel-Schneider, and B. Parsia, "OWL 2 Web Ontology Language Direct Semantics (Second Edition)," 11.12.2012. [Online]. Available: <https://www.w3.org/TR/owl2-syntax>
- [12] P. Hitzler, M. Krtzsch, and S. Rudolph, *Foundations of Semantic Web Technologies*, 1st ed. Chapman & Hall/CRC, 2009.