Software and System Modeling manuscript No.
(will be inserted by the editor)

Transforming XML Schemas into OWL ontologies using Formal Concept Analysis

Mokhtaria Hacherouf · Safia Nait-Bahloul · Christophe Cruz

Received: date / Accepted: date

Abstract Ontology Web Language (OWL) is considered as a data representation format exploited by the Extensible Markup Language (XML) format. OWL extends XML by providing properties to further express the semantics of data. To this effect, transforming XML data into OWL proves important and constitutes an added value for indexing XML documents and re-engineering ontologies. In this paper we propose a formal method to transform XSD schemas into OWL schemas using transformation patterns. To achieve this end, we extend at the beginning, a set of existing transformation patterns to allow the maximum transformation of XSD schema constructions. In addition, a formal method is presented to transform an XSD schema using the extended patterns. This method named PIXCO comprises several processes. The first process models both the transformation patterns and all the constructions of XSD schema to be transformed. The patterns are modeled using the context of Formal Concept Analysis. The XSD constructions are modeled using a proposed mathematical model. This modeling will be used in the design of the following process. The second process identifies the most appropriate patterns to transform each construction set of XSD schema. The third process generates for each XSD construction set an OWL model according to the pattern that is identified. Finally, it creates the OWL file encompassing the generated OWL models.

Keywords XML Schema · OWL ontology · Formal transformation · XSD Formalization · FCA · Transformation patterns

M. Hacherouf
Laboratoire LSSD, Faculté des mathématiques et informatique, Université des Sciences et de la Technologie d'Oran Mohamed Boudiaf, USTO-MB, BP 1505, El Mnaouer, 31000 Oran Algérie
E-mail: mokhtaria.hacherouf@univ-usto.dz

S. Nait-Bahloul
Laboratoire LITIO, Université d'Oran 1, Ahmed Ben Bella, BP 1524, El-M'Naouer, 31000 Oran, Algérie Nait-bahloul.safia@univ-oran.dz

C. Cruz
Le2i, CNRS FRE 2005, Arts et Métiers, Univ. Bourgogne Franche-Comté, Bâtiment I3M rue Sully, Dijon, France christophe.cruz@ubfc.fr

1 Introduction

The ontologies are widely used to express the semantics of data. They are based on a formal description in order to model the business knowledge. Nevertheless, the development of these ontologies from scratch is an expensive and difficult task [15]. Consequently, some approaches such as [2, 9, 18] propose to use ex-

isting data sources that containing the semantic definitions of entities. The existing data sources are usually modeled by various formats such as XML [7]. These sources are intended for applications that do not support the semantics as expressed in OWL [39]. The transformation of an XML document into an OWL document permits to generate ontologies by minimizing the cost and the time of the realization. In addition, this transformation is necessary and useful for other areas such as data integration systems ([5, 11, 27, 38]). To achieve this transformation, the literature proposes two processes. The first process which we are interested tries to find mapping rules between the XML schema (XSD) constructs and the OWL schema constructs in order to generate the ontology. The second process consists in populating the generated ontology from the XML instances. Existing approaches are often limited to the theoretical aspects of proposing a minimum number of mapping rules. It considers the most used XSD elements as *xsd:element*, *xsd:attribute*, *xsd: complexType*, *etc*. In addition, no approach provides a formal method transforming an XSD schema into OWL using predefined mapping rules.

We propose a formal method of XSD schema transformation into OWL ontology using a large number of transformation patterns. A transformation pattern links an XSD block to an equivalent OWL ontology model. An XSD block is formed by a sub-hierarchy of XSD constructs. An XSD construct is defined by an XSD element (*xsd:element*, *xsd: simpleType*, ...) with all its attributes (*name*, *type*, *ref*, ...). To achieve the proposed method, we define at the beginning, the set of transformation patterns, then, we provide the method that describes how to use these patterns for the transformation of an XSD schema into OWL ontology.

The set of transformation patterns defined is based on the Janus method [3]. We used this method to process the maximum number of XSD elements (see [22]). The XSD elements processed by Janus are relatively used in the B2B (Business to Business) domain. But this domain can be very evolutionary over time, and the use of certain elements grows significantly. For example, the *xsd:key* element is not processed by Janus because its usage percentage is 0% (see Figure 1 in [3]). Thus, it is necessary to consider the maximum of XSD elements. Actually, the more a transformation process considers the XSD elements, the more ontology generated is rich semantically and exhaustive. To this end, we extend the Janus patterns by proposing new patterns. These patterns address the XSD constructions which take care of coherence constraints. These constraints play a fundamental role in the design of XSD schemas.

The transformation method using the extended patterns draws on a set of processes. The first process models, on the one hand, the transformation patterns using the context of Formal Concept Analysis (FCA) [17], and on the other hand, the XSD schema to be transformed using a proposed mathematical model called $\mathcal{FS}$ ($\mathcal{XS}$). This modeling will be used to realize the following processes. The second process identifies among a set of extended patterns, the candidate patterns to transform each block of an XSD schema. For a block of XSD constructs this process can identify several patterns that are similar. The similar patterns are formed by the same XSD elements but differ in their attributes. The third process checks the result of the previous process, and selects the relevant pattern from the similar patterns. The fourth process creates for each XSD block the equivalent OWL blocks using the relevant pattern. Then, it generates the OWL file.

The remaining of this paper is organized as follows. Section 2 discusses the various related work. Section 3 presents the extension of the patterns proposed by the Janus method. Section 4 describes the transformation method of an XSD schema into an OWL ontology using the extended patterns. Section 5 deals with the PIXCO prototype that implements the proposed solution. Section 6 presents a discussion

on the PIXCO prototype. Finally, Section 7 concludes this paper.

2 Related work

OTM (Ontological Transformation Methods) are methods design to transform data from its original format into an ontological format. For example, some methods focus on transforming databases into OWL ontology such as [18, 25, 41]. Other methods focus on transforming data modeled using UML (Unified Modeling Language) such as [2, 4, 14]. As for the transformation of XML data, there are many approaches ([3, 6, 9, 15, 19, 26, 29, 32, 35, 37, 40]) on which our approach contributes.

In knowledge engineering, ontology development methods that are totally manual such as [8, 30, 34] generate ontologies controlled by engineers or domain experts. They produce ontologies of quality. That is to say, ontologies are adapted to their needs. However, achieving these methods is complex, costly and time-consuming. To make the ontology development process lighter and faster, the literature proposes to re-use existing data sources containing the semantic definitions of the entities. However, data sources can not be used directly. The transformation of data sources to an ontological format is required. This transformation is achieved using *OTM*.

In data integration systems, the main objective is to generate a global schema from a set of heterogeneous data sources. To achieve this objective some approaches propose to find correspondences between the elements of the schemas of the data sources to be integrated. Among those approaches which are concerned the XML format: [10, 12, 31, 28]. Other methods are discussed and evaluated with XML Matchers [1] and XBenchMatch [13]. Usually these approaches discover the mappings between the schema elements based on their names, data types, constraints, and schema structure. Nevertheless, in some cases, schema elements do not correspond structurally, but they can be linked by certain semantic properties. These properties make possible to establish the correspondences. As a result, other approaches such as [5, 11, 27, 38] propose in the first level to transform the data sources to be integrated using the *OTM*. Then, ontology alignment or fusion techniques ([20, 24, 33]) are used to find the correspondences. These techniques exploit the semantic properties by expressing the semantics of the data.

To this end, we found that both domains (knowledge engineering and data integration) require the design of *OTM*. In Table 1 we summarize the strong (+) and weak (-) points of these methods in both domains.

The proposed method applies *OTM* on the XML format. Different approaches of transformation of XML documents into OWL ontology is presented in [22]. These approaches are classified into two main classes according to the types of schema from which the ontology is generated. The first class ([29, 32]) generates ontologies from XML instances. The second class ([3, 6, 9, 15, 19, 26, 35, 37, 40]) generates ontologies from XML schema or DTD schema. The proposed method belongs to the second class based on mapping rules between ontology constructors (*owl:Class*, *owl:ObjectProperty*, *owl:DataProperty*, ...) and XML schema constructors (*xs:element*, *xs:attribute*, *xs:complexType*, ...). The ontologies generated by the second class are richer in concepts and well-structured compared to those generated from the instances. This is due to XSD schemas containing expressions such as descriptions of element contents, occurrences, type definitions, *etc.* These expressions do not exist in an XML instance. The first class generates a different ontology for each XML instance even if these XML instances have been validated by the same schema. In contrast, the second class can generate only a single ontology for several XML documents validated by the same schema. This avoids the rerun of the transformation process.

Although the class that generates the ontology from XSD schema is more advantageous. Unfortunately, it remains limited by problems

Table 1 The advantages and disadvantages of using *OTM*.

	Knowledge engineering	**Data integration systems**
OTM: – Data Bases/OWL : [18, 25, 41] – UML/OWL : [2, 4, 14] – XML/OWL : [3, 6, 9, 15, 19, 26, 29, 32, 35, 37, 40]	+ Fast and light ontology development process. + Reduction of human interventions. - Ontology resulting from processing less quality.	+ Identification of the correspondences between the elements of sources by introducing a semantic level. + Use of ontology alignment techniques to facilitate matching. - Length of the integration process by adding the data transformation phase.

that we hold to resolve. Table 2 summarizes the differences between the methods of this class. The aspects taken into account in the comparison are the automation of the transformation, the consideration of the consistency constraints in the transformation process, the provision of the formal method of transformation, and the number of XSD elements processed. We obtained this number from the article [22] which accounted the number of XSD elements processed by each approach. It cited in a table (see Table 9 in [22]) a set of XSD elements and verified for each element if it processed by each approach. Afterwards, the final number of XSD elements processed by each approach is determined.

We discuss the comparison criteria summarized in Table 2:

- The semi-automatic transformation gives more correct and more reliable ontology since it is controlled by a human. However, it is preferable to automate the process of transformation in the context of Data Mining. Moreover, our approach has identified a large number of patterns and has provided a formal method of transformation using these patterns. This encourages more the use of automated transformation process.
- Our approach is the only one that considers the constraints of consistency (*xsd:key*, *xsd:unique*, *xsd:keyref*). This keeps information about the structure of the data and generates a richer ontology from a logical constraint point of view.
- Our approach is based on a formal transformation method using a mathematical formalizes as FCA context. This context was previously used for the generation of ontologies. For example, the approach [21] uses FCA for the generation of ontologies from texts. The approach [36] uses FCA for ontology generations in a Data Mining context. The work [16] provides an approach for the generation of FCA-based ontology for the integration of data sources containing implicit and ambiguous information. Generally, these approaches ([21, 36, 16]) have used FCA as techniques for classifying concepts. While the purpose of using the FCA context in this method is to model the mapping rules between XML schema constructs and OWL schema constructs.
- Our approach considers a wide set of XSD elements which makes it possible to transform more constructions of an XSD schema. Consequently, The resulting ontology is a richer ontology compared to that generated by other methods.

3 Janus Patterns extension

In this section the Janus method to be extended is briefly presented, then, the method relies on the definition of the proposed patterns that are not considered by other methods.

Table 2 A comparative analysis of the XSD/OWL transformation approaches.

Approach	Automation transformation	Consistency constraints	Formal method	Number of XSD elements processed
OWLMAP [15]	Automatic	No	No	14
XML2OWL [6]	Automatic	No	No	8
XS2OWL [37]	Automatic	No	No	7
XSD2OWL [9]	Semi-automatic	No	No	-
X2OWL [19]	Automatic	No	No	9
Janus [3]	Automatic	No	No	19
EXCO [26]	Automatic	No	No	10
Approche [40]	Automatic	No	No	9
Our method	**Automatic**	**Yes**	**Yes**	**24**

3.1 Janus Overview

The method [3] proposed a tool named Janus which allows to transform an XSD schema in OWL 2-RL ontology. This tool is based on forty transformation patterns considering a wide number of XSD elements (19 XSD elements) compared to the similar approaches cited in [22]. A transformation pattern is an XSD structure corresponding to an equivalent OWL ontology model. For example, Table 3 presents two patterns proposed by the Janus method. The first column shows the pattern number, the second column shows the schema of the XSD structure, and the third column shows the OWL model corresponding to the XSD structure.

3.2 The proposed transformation patterns

Although, the Janus method has process a wide number of XSD schema elements, there are still other untreated elements. However, we enrich this method by addressing with new elements. These elements mainly concern coherence constraints, which fall into two types: constraints of uniqueness and constraints of existence. Example 1 shows an XSD schema in which we present the use of coherence constraints. This schema describes a publication article. It imposes three following constraints:

1. Two authors do not have the same name.
2. The DOI value is unique.
3. An article DOI must be existed in the DOI list of the journal.

Example 1:

```
<?xml version="1.0" encoding="iso-8859-1"
    ?>
<xs:schema xmlns:xs="http://www.w3.org
    /2001/XMLSchema">
 <xs:element name="article">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="authors" type="
        authorsType"/>
    <xs:element name="journal" type="
        journalType"/>
   </xs:sequence>
   <xs:attribute name="doi" type="
       xs:integer"/>
  </xs:complexType>
  <!-- Uniqueness of the authors names --
      >
  <xs:unique name="uniqueAuthors">
   <xs:selector xpath="authors/author"/>
   <xs:field xpath="@nameAuthor"/>
  </xs:unique>
  <!-- Unity of the DOI of the journal --
      >
  <xs:key name="doiKey">
   <xs:selector xpath="journal/listOfDOI"
       />
   <xs:field xpath="@doi"/>
  </xs:key>
  <!-- Existence of DOI of the article in
       the list of DOI of the journal -->
  <xs:keyref name="DOIrefer" refer="
      doiKey">
   <xs:selector xpath="article"/>
   <xs:field xpath="@doi"/>
  </xs:keyref>
 </xs:element>
</xs:schema>
```

The specific patterns of unique constraints that precess the elements *xsd:key* and *xsd:unique* will presented in Section 3.2.1. The specific patterns of existence constraints that process the element*xsd:keyref* will be introduced in Section 3.2.2.

Table 3 A transformation patterns proposed by Janus method [3].

N	XSD schema	OWL 2 (Turtel Syntax)
1	<xsd:simpleType name="st_name">	:st_name rdf:type rdfs:Datatype.
2	<xsd:simpleType name="st_name1"> <xsd:union memberTypes="st_name xsd:nativDataType"/> </xsd:simpleType>	:st_name1 owl:equivalentClass :st_name; owl:equivalentClass xsd:nativeDataType; owl:equivalentClass ...

3.2.1 Specific transformation patterns of unique constraints

In an XSD schema, a unique constraint is expressed using the syntax *xsd:key* or *xsd:unique* in the contents of a given element. Only one element composed of sub-elements or attributes has a unique value. These schema constraints are specified in the Table. 4 column *"XSD schema"*. An element *xsd:key* or *xsd:unique* has only one attribute *name* whose value *"key_name"* represents the constraint name. The two elements *xsd:key* and *xsd:unique* contains an element *xsd:selector* and an elements *xsd:field* having each one an attribute *xpath*. The element *xsd:selector* that determines the constraint elements has an attribute *xpath* whose value *"elt_name"* is an XPath[1] expression which selects the element with the constraint. Each of the elements *xsd:field* has an attribute *xpath* whose value *"attr_key_name"* specify a value field that must be unique. *xsd:key* element requires that each of the fields determined by the elements *xsd:field* is present and the value thus constituted is unique. Unlike the constraint given by a *xsd:unique* that does not require the presence of each field determined by the elements *xsd:field*.

However, even in OWL 2 [23] a class can have a data property collection having unique instances. In this case, the collection of data properties *owl:DatatypeProperty* is assigned as a key of their superclasses. The OWL expression that expresses this assignment is *owl:haskey*. To this end, we find that the elements *xsd:key* and *xsd:unique* have the same semantics as the owl element *owl:haskey*. Two transformation patterns 41 and 42 are defined and presented in Table 4.

The XSD element *"elt_name"* that carries the unique constraint is always a complex element since it contains sub-elements or attributes. According to the Janus pattern number 3 (see Table II in [3]) a complex element is represented by a class in OWL. The sub-elements or attributes *"attr_key_name"* of this element are represented by the data properties according to Janus patterns 15 and 26 (see Table IV and V in [3]). To express the unique constraint of XSD schema in OWL, we assign the property *owl:haskey* to the class *"elt_name"* and the key is *"attr_key_name"*. For the pattern 41, we add the construction *owl:minCardinality* to impose the presence of a data property.

3.2.2 Specific transformation patterns for existence constraints

An existence constraint requires that for each given element, there is an element with a unique constraint associated with the same value. We specify the constraint schema in the Table 5. column *"XSD schema"*. The element *xsd:keyref* has two attributes: *name* and *refer*. The value *"keyref_name"* of the attribute *name* represents the name of the constraint. The value *"key_name"* of the attribute *refer* must be the name of a unique constraint that is associated with this existence constraint. The element *xsd:keyref* contents an element *xsd:selector* and elements *xsd:field*. The element *xsd:selector* selects the element *"elt_name"* on which the constraint is carried. The elements *xsd:field* determine the field *"attr_keyref_name"* (or different field) of the value to be a value of a field that is an existing unique key.

[1] http://www.w3.org/TR/xpath/

Table 4 Specific patterns for unique constraints.

N	XSD schema	OWL schema
41	<xsd:key name="key_name"> <xsd:selector xpath="elt_name"/> <xsd:field xpath="@attr_key_name"/> ... </xsd:key>	<owl:Class rdf:about="elt_name"> <owl:hasKey rdf:parseType="Collection"> <owl:DatatypeProperty rdf:about="attr_key_name"/> ... </owl:hasKey> <rdfs:equivalentClass> <owl:Restriction> <owl:onProperty rdf:resource="attr_key_name"/> <owl:minCardinality rdf:datatype= "&xsd;nonNegativeInteger">1 </owl:minCardinality> </owl:Restriction> </rdfs:equivalentClass> </owl:Class>
42	<xsd:unique name="key_name"> <xsd:selector xpath="elt_name"/> <xsd:field xpath="@attr_key_name"/> ... </xsd:unique>	<owl:Class rdf:about="elt_name"> <owl:hasKey rdf:parseType="Collection"> <owl:DataProperty rdf:about="attr_key_name"/> ... </owl:hasKey> </owl:Class>

However, even in OWL 2 a class can have a data property that has only the values to be instances of a specific class. To this effect, we notice that the elements *xsd:keyref* and *owl:allValuesFrom* have the same semantics. Consequently, we define the transformation pattern 43 described in the Table 5. The XSD element *"elt_name"* that is bearing the existence constraint is represented by a class in OWL. The attribute (or sub-element) *"attr_key_name"* of this element is represented by a data property *owl:datatypeProperty*. To express the existence constraint, we added the existential quantifier *owl:allValuesFrom* to the class *"elt_name"*.

4 Formal method of transformation

An XSD schema is formed by a set of XSD construct that are hierarchically related between them and each constructions block is transformed using a specific pattern. The identification of each pattern for each XSD block is a complex process. The complexity stems from the fact that the XSD block can have several cases of representation and each XSD block is transformed using a particular pattern. This pattern is selected from a high number of patterns. However, the generation of ontology from an XSD schema is accomplished by several steps (Figure 1).

The step 1 deals with the formalization of XSD schemas and the 43 transformation patterns. The formalization of the transformation patterns is done using the FCA context [17] by considering that the patterns are the objects and the XSD elements are the attributes. The concept lattice that is designed permit to easily extract the pattern linked with a set of XSD elements and vice versa. For the formalization of the XSD schema to be transformed, a formal model named $\mathcal{FS}(\mathcal{XS})$ is proposed. This model allows to manipulate the XSD constructions by representing these elements using sets, functions and applications. To this end, the formalization of both the XSD schema ($\mathcal{FS}(\mathcal{XS})$) and the transformation patterns is the basis of the conception of the next steps. The step 2 consists in automatically identifying the patterns corresponding to the XSD blocks of an XSD schema. This step is performed by proposing two algorithms: 1) *PatternsIdentification* and 2) *GetIdentifiedPatterns*.

Table 5 Specific patterns for existence constraints.

N	XSD schema	OWL schema
43	`<xsd:keyref name="keyref_name" refer="key_name">` `<xsd:selector xpath="elt_name"/>` `<xsd:field xpath="@attr_keyref_name"/>` `...` `</xsd:keyref>`	`<owl:Class rdf:about="elt_name">` `<rdfs:equivalentClass>` `<owl:Restriction>` `<owl:allValuesFrom rdf:resource="attr_key_name"/>` `<owl:onProperty rdf:resource="attr_keyref_name"/>` `</owl:Restriction>` `...` `</rdfs:equivalentClass>` `</owl:Class>`

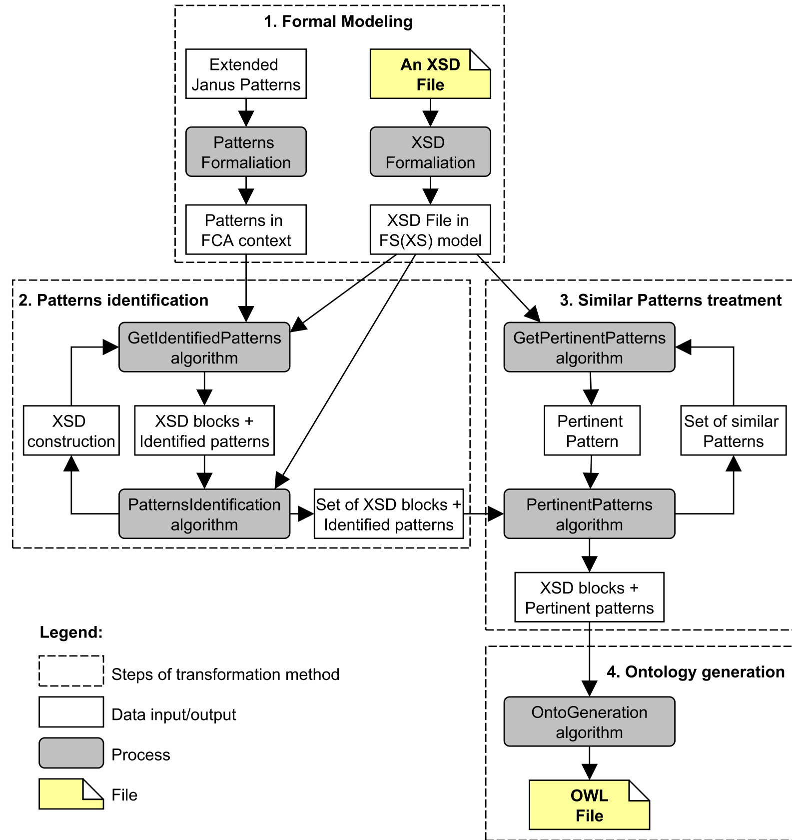

Fig. 1 The main steps of the proposed transformation method.

The first algorithm travels the XSD constructs set and calling the second algorithm for each XSD construct not yet processed. The latter identifies the XSD block concerned by this construction and the patterns that can be used to transform this block. The step 3 processes the similar patterns identified during the step 2. These patterns are constituted by XSD constructions whose XSD elements are identical, and the attributes are different. The last step takes each XSD block and generates the OWL block according to the pattern identified for

this block. Finally, the OWL file that encompasses all OWL blocks is created.

4.1 Formal modeling

For the conception of the transformation method, two types of modeling are necessary at the beginning. The transformation patterns are represented using FCA context, then, the XSD schema to be transformed is formalized using a proposed mathematical model.

4.1.1 Patterns Formalization

An FCA context is a triple (G, M, I) containing a set of objects G, a set of attributes M, and a relationship noted I defined by $I \subseteq G \times M$. This context is used to represent the transformation patterns by considering that:

- G: means the set of patterns such as $G = \{1, 2, 3 \ldots 43\}$.
- M: means the set of XSD schema elements considered by transformation patterns such as:
  $M = \{simpleType, union, \ldots\}$.
- gIm: indicates that the pattern g uses the element m. For example,
  $1IsimpleType$ mean the pattern 1 use the element *simpleType*.
- For $B \subseteq M$ the formal operator of concepts $^{\downarrow}$ is defined as following:
  $^{\downarrow} : 2^M \longrightarrow 2^G$
  $B^{\downarrow} = \{g \in G \mid foreach m \in B :< g, m >\in I\}$
  For instance, **if**
  $B = \{simpleType, restriction, enumeration\}$
  then $B^{\downarrow} = \{4\}$.

The patterns and XSD elements are in large numbers: 43 transformation patterns and 25 XSD elements. As a result, the FCA table consists of 43 rows and 25 columns. Due to the large space that this table takes, we only consider the first six Janus patterns (see tables II and III in [3]) on which we create the FCA context table. Table 6 presents this partial context. The position (i, j) is 1 if and only if the pattern i uses the element j, else 0.

Table 6 The partial formal binary context of patterns.

	simpleType	**union**	**restriction**	**enumeration**	**minInclusive**	**maxInclusive**	**complexType**	**simpleContent**	**extension**
1	1	0	0	0	0	0	0	0	0
2	1	1	0	0	0	0	0	0	0
3	0	0	0	0	0	0	1	0	0
4	1	0	1	1	0	0	0	0	0
5	1	0	1	0	1	1	0	0	0
6	0	0	0	0	0	0	1	1	1

From the full table of the formal context, we trace the corresponding full concept lattice as shown in Figure 2 using the conexp-1.3[2] tool. This lattice represents the classes linking objects (patterns) and attributes (XSD elements). For a given class, we have a set of patterns linking XSD elements linked in the lattice. For example, the node *element*, if there is only one exploits the patterns 15, 16, 21, 24, 25. In contrast, if it is linked directly with the *annotation* node, then there are two paths: 1) the path goes down to the element *appinfo* and the pattern 40 is mapped. 2) the path goes down to the element *documentation* and the pattern 39 is mapped. To this effect, the element *element* is indirectly linked with the elements *appinfo* and *documentation*. It may also be linked indirectly with other elements such as *complexType* and *sequence*.

4.1.2 XSD Formalization

To formalize the set of XSD schema constructions, a new model named $\mathcal{FS}(\mathcal{XS})$ (**F**ormal **S**tructure of **X**ML **S**chema) is proposed. This model is defined by 6-tuple $S = (E, L_E, A, V_A, C, H_C)$ formed by:

[2] conexp.sourceforge.net/.

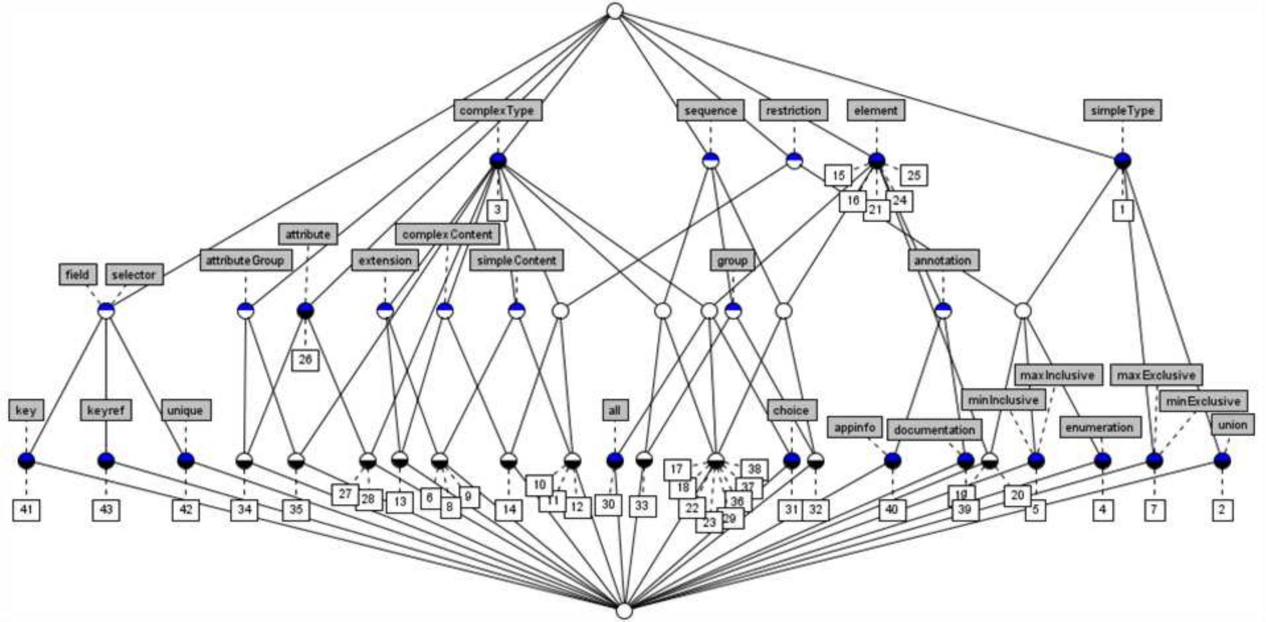

Fig. 2 The full concept lattice.

1. A set E containing all the **elements** of an XSD schema:
   $E = \{all, annotation, attribute, choice, element, \ldots\}$.
2. A set L_E containing all the **literals** contained in the elements of an XSD schema.
3. A set A containing all the **attributes** of an XSD schema:
   $A = \{targetNamespace, xml : lang, name, type, minOccurs, \ldots\}$.
4. A set V_A containing all the **attributes values** of an XSD schema.
5. A **collection of functions** C representing all the **constructions** of an XSD schema. A construction is a representation modeled as following:
   $C = \{C_1, \ldots C_n\}$ such as $n \in \mathbb{N}$ and $C_i : E \longrightarrow 2^{A \times V_A} \times L_E$.
6. A **function** H_C which expresses a **hierarchy of constructions**:
   $H_C : C \longrightarrow 2^C$.

To illustrate the model $\mathcal{FS}(\mathcal{XS})$, we use example 2 which describes the characteristics of a vehicle. The example consists of 22 numbered lines. Lines expressing the closing of a tag are ignored in modeling. These lines are 6, 7, 12, 13, 18, 19, 20 and 22. The lines that will be considered in modeling are: 1, 2, 3, 4, 5, 8, 10, 11, 14, 15, 16, 17 and 21. An XSD construct is referenced by $C_{ligneNumber}$. For example C_1 refers to the construction of line number 1.

Example 2:

```
1<xs:schema xmlns:xs="http://www.w3.org
      /2001/XMLSchema" targetNamespace="
      http://mynamespace/vehicleSchema">
2 <xs:simpleType name="categoryType">
3  <xs:restriction base="xsd:string">
4   <xs:enumeration value="heavy"/>
5   <xs:enumeration value="light"/>
6  </xs:restriction>
7 </xs:simpleType>
8 <xs:complexType name="vehicle">
9  <xs:sequence>
10   <xs:element name="mark" type="
        xsd:string" maxOccurs="1"/>
11   <xs:element name="color" type="
        xsd:string"/>
12  </xs:sequence>
13 </xs:complexType>
14 <xs:complexType name="truck">
15  <xs:complexContent>
16   <xs:extension base="vehicle">
17    <xs:attribute name="category" type=
      "categoryType"/>
18   </xs:extension>
19  </xs:complexContent>
20 </xs:complexType>
21 <xs:element name="vehicleTransport"
      type="truck"/>
22</xs:schema>
```

The formalization of Example 2 using the $\mathcal{FS}(\mathcal{XS})$ model takes the following form:
$E = \{xs : schema, xs : simpleType, xs : restriction,$
$xs : enumeration, xs : complexType, xs : sequence,$
$xs : element, xs : complexContent, xs : extension,$
$xs : attribute, xs : annotation, xs : documentation\}.$
$T_E = \{"Transportsociety"\}.$
$A = \{xmlns : xs, targetNamespace, name, base, value,$
$type, maxOccurs, xml : lang, source\}.$
$V_A = \{"http : //www.w3.org/2001/XMLSchema",$
$"http : //mynamespace/vehicleShema",$
$"categoryType", "xsd : string", "heavy", "light",$
$"vehicle", "mark", "1", "color", "truck",$
$"category", "vehicleTransport", "Ang",$
$"http : //www.transport.com"\}.$
$C = \{C_1, C_2, C_3, C_4, C_5, C_8, C_9, C_{10}, C_{11}, C_{14}, C_{15},$
$C_{16}, C_{17}, C_{21}, C_{22}, C_{23}\}.$
$C_1 = \langle xs : schema, \{(xmlns : xsd,$
$"http : //www.w3.org/2001/XMLSchema"),$
$(targetNamespace,$
$"http : //mynamespace/vehicleSchema")\}\rangle.$
$C_2 = \langle xs : simpleType, (name, "categoryType")\rangle.$
$C_3 = \langle xs : restriction, (base, "xsd : string")\rangle.$
$C_4 = \langle xs : enumeration, (value, "heavy")\rangle.$
$C_5 = \langle xs : enumeration, (value, "light")\rangle$
$C_8 = \langle xs : complexType, (name, "vehicle")\rangle.$
$C_9 = \langle xs : sequence\rangle.$
$C_{10} = \langle xs : element, \{(name, "mark"), (type,$
$"xsd : string"), (maxOccurs, "1")\}\rangle.$
$C_11 = \langle xs : element, \{(name, "color"),$
$(type, "xsd : string")\}\rangle.$
$C_{14} = \langle xs : complexType, (name, "truck")\rangle.$
$C_{15} = \langle xs : complexContent\rangle.$
$C_{16} = \langle xs : extension, (base, "vehicle")\rangle.$
$C_{17} = \langle xs : attribute, \{(name, "category"),$
$(type, "categoryType")\}\rangle.$
$C_{21} = \langle xs : element, \{(name, "vehicleTransport"),$
$(type, "truck")\}\rangle.$
$H_C = \{(C_1, C_2)(C_2, C_3), (C_3, C_4), (C_3, C_5), (C_1, C_8),$
$(C_8, C_9), (C_9, C_{10}), (C_9, C_{11}), (C_1, C_{14}), (C_{14}, C_{15}),$
$(C_{15}, C_{16}), (C_{16}, C_{17}), (C_1, C_{21})\}.$

The model $\mathcal{FS}(\mathcal{XS})$ can be freely used since it allows to formally manipulate all the constructions of an XSD schema. In addition, it can be implemented in temporary or persistent mode. For the temporary mode, vectors or matrices can be used. For the persistent mode, the relational tables are convenient. In the implementation of the proposed method, we use the persistent mode because we have several methods that manipulate the XSD constructs. And these methods are executed in different instances.

4.2 Patterns identification

In this section the most appropriate patterns will be identified for each XSD block. Identifying one or two patterns among 43 patterns is a complex task. However, two algorithms are proposed (see Fig.1 step 2).

The algorithm *PatternsIdentification* (Algorithm 1) takes as input an XSD schema formalized using the $\mathcal{FS}(\mathcal{XS})$ model. At the exit, it gives a set of XSD blocks where each block is accompanied by a set of identified patterns. The principle consists in browsing through the XSD schema constructions sequentially in the order $(C_1, C_2, C_3, \ldots)$. The algorithm takes a construction that is not yet marked and transmits it to the *getIdentifiedPatterns* algorithm (Algorithm 2). This latter returns the block that starts with this construction and the patterns identified for this block. The main algorithm records in a matrix of two columns the construction blocks and the identified patterns. At this stage, constructions of the block are marked. Thus, the algorithm 1 yet returns the next construction which is not marked, then repeated the process for it. The loop runs until having marked all constructions.

The algorithm *getIdentifiedPatterns* (Algorithm 2) takes as input a construction C_i to identify the block $block_i$ which begins with this construction and the patterns used to transform this block. First, the algorithm constitutes initially the block by the construction C_i, i.e., $block_i = \{C_i\}$. Second, it traverses the list of constructions C starting from the C_i. And it gradually adds constructions to $block_i$. The added construction C_{next} must verify two conditions: The first condition is that C_{next} is connected to the input construction in H_c.

The second condition is that the XSD element of C_{next} is linked with the XSD elements of the XSD block in the lattice. If the first condition is satisfied and the second is not checked, then a new block is initially identified by the construction C_{next}. This C_{next} is accompanied by a XSD block where the XSD elements are associated with the XSD element of C_{next} in the lattice. Thus, this new block will be processed in the same way.

In order to show the principle of both algorithms, the example 2 formalized with the $\mathcal{FS}(\mathcal{XS})$ model is used. Table 7 show the states of the principal variables using by the algorithms. These variables are the identified XSD blocks, construction C_i and the pattern set P that are identified for each XSD block. White cells mean that the state of the variables has not changed. Gray cells mean the final states.

The final result of the pattern identification is presented using a matrix ***PI***. Table 8 shows this matrix. The first column contains the XSD blocks and the second column contains the patterns identified for each block.

From the Table 8, it can be noted that for a block, the pattern identification algorithm can identify several patterns. For block $\{C_{10}, C_{13}\}$, it identifies two patterns 27 and 28. This is the case when the patterns are the same XSD elements and the algorithm does not distinguish between them. Section 4.3 details the process of the distinction method.

Both algorithms *PatternsIdentification* and *getIdentifierPatterns* use a set of methods. Table 9 provides a description of each method.

Algorithm 1: PatternsIdentification

```
input  : schema_FS(XS)    /* FS(XS) model of an XSD schema */
output: PI [n] [2]
C_i ∈ C ;                  /* XSD construction */
Block ⊂ C ;        /* sub-set of constructions */
P ⊂ G ;                 /* sub-set of patterns */
j = 0;
repeat
    C_i = getConstructionNotMarked ;
    {Block, P} = getIdentifiedPatterns (C_i);
    PI [j] [0] = Block;
    PI [j] [1] = P;
    j++;
until hasConstructionNotMarked = false;
```

Algorithm 2: getIdentifiedPatterns

```
input  : C_i ∈ C
output: {Block, P}
sG ⊂ G;
sM ⊂ M;
P = {P_0, ... P_n}; P_i ⊂ G;
Block = {Block_0, ... Block_n}; Block_i ⊂ C;
C_next ∈ C;
i=0;
markConstruction (C_i) ;     /* marking the entry
  construction */
Block_i = {C_i} ; /* initialize the block with the
  entry construction */
while hasNextConstruction (C_i) do
    C_next = getNextConstruction (C_i) ;
      /* return the next construction of C_i */
    markConstruction(C_next) ;
    Block_i = Block_i ∪ {C_next} ;       /* add the
      following construction to the block that
      is in processing runs */
    sM=getElements (Block_i);
    sG = sM^↓;
    if sG = ∅ then
        Block_i = Block_i − {C_next} ;  /* removes
          the construction since it does not
          form a pattern */
        i =i+1 ;
        Block_i =
          getIntersection(Block_{i−1},C_next);
    end
end
/* recover blocks and sets of identified
   patterns                                   */
for k = 0 to i do
    Block = Block ∪ Block_k;
    sM = getElements(block_k);
    P_k = getPatterns (sM);
    P = P ∪ P_k;
end
```

4.3 Similar patterns treatment

Patterns consisting of the same XSD elements but they possibly differ from their attributes or their attribute values, are called **similar patterns**. Table 10 shows an example of two similar patterns. The constructions of the two patterns 27 and 28 contain the same set of XSD elements $\{complexType, attribute\}$. The difference is that the element *attribute* in the pattern 27 contains the attributes *name* and type, whereas in the pattern 28 it has the attribute *ref*.

Figure 3 shows the set of similar patterns found in 43 patterns. These sets are extracted from the concept lattice as shown in Figure 2.

We found that there are 6 sets of similar patterns. The result of patterns identification

Table 7 Identification of patterns for Example 2 using algorithms 1 and 2.

C_i	**States of Blocks**	**Blocks of construction elements**	**States of P**	**Explanations**
C_2	$\{C_2\}$	$\{\emptyset\}$	$\{\emptyset\}$	
	$\{C_2, C_3\}$	$\{simpleType, restriction\}$	$\{49, 4, 5, 7, 19, 20\}$	C_8 is not taken because the first condition is not satisfied.
	$\{C_2, C_3, C_4\}$	$\{simpleType, restriction, enumeration\}$	$\{4\}$	
	$\{C_2, C_3, C_4, C_5\}$		$\{4\}$	
C_8	$\{C_8\}$	$\{\emptyset\}$	$\{\emptyset\}$	
	$\{C_8, C_9\}$	$\{complexType, sequence\}$	$\{17, 18, 22, 23, 29, 33, 36, 37, 38\}$	C_{14} is not taken because the first condition is not satisfied.
	$\{C_8, C_9, C_{10}\}$	$\{complexType, sequence, element\}$	$\{17, 18, 22, 23, 29, 36, 37, 38\}$	
	$\{C_8, C_9, C_{10}, C_{11}\}$		$\{17, 18, 22, 23, 29, 36, 37, 38\}$	
C_{14}	$\{\emptyset\}$	$\{\emptyset\}$	$\{\emptyset\}$	C_{17} will be added to a new block because the 1st condition is verified and the 2nd condition is not verified.
	$\{C_{14}, C_{15}\}$	$\{complexType, complexContent\}$	$\{13, 14\}$	
	$\{C_{14}, C_{15}, C_{16}\}$	$\{complexType, complexContent, extension\}$	$\{13\}$	
	$\{C_{14}, C_{15}, C_{16}, C_{17}\}$	$\{complexType, complexContent, extension, attribute\}$	$\{\emptyset\}$	
	$\{C_{14}, C_{15}, C_{16}\}$	$\{complexType, complexContent, extension\}$	$\{13\}$	
	$\{C_{14}, C_{17}\}$	$\{complexType, attribute\}$	$\{27, 28\}$	C_{14} and C_{17} constitute a new block because its XSD elements are connected in the lattice.
C_{21}	$\{C_{21}\}$	$\{element\}$	$\{15, 16, 21, 24, 25\}$	All constructions are marqued.

Table 8 The resulting matrix PI of the pattern identification algorithm.

Block	**P**
$\{C_2, C_3, C_4, C_5\}$	$\{4\}$
$\{C_8, C_9, C_{10}, C_{11}\}$	$\{17, 18, 22, 23, 29, 36, 37, 38\}$
$\{C_{14}, C_{15}, C_{16}\}$	$\{13\}$
$\{C_{14}, C_{17}\}$	$\{27, 28\}$
$\{C_{21}, C_{22}, C_{23}\}$	$\{15, 16, 21, 24, 25\}$

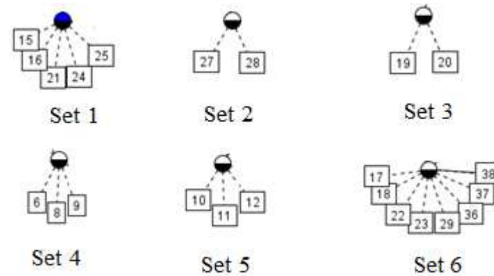

Fig. 3 Sets of similar patterns.

process may contain one of similar patterns set for an XSD block which requires to select the **pertinent pattern** that is the most suitable. To solve this issue, we propose a pattern selection algorithm named *PertinentPatterns* (Algorithm 3). The principle of this algorithm 3 is to browse all the pattern set column. If a set is identified as similar patterns, then this set and all the associated constructions will be

Table 9 Methods used by the pattern identification algorithms.

Method	Parameters	Return	Description
markConstruction	$C_i \in C$		Marks the construction C_i.
hasConstructionNotMarked		True/false	Checks if there is again a construction that is not marked.
getConstructionNotMarked		$C_i \in C$	Returns the first construction that is not yet marked.
hasNextConstruction	$C_i \in C$	True/false	Checks in H_c if there is still an unmarked construction gathered with C_i or with its combination (couple of C_i).
getNextConstruction	$C_i \in C$	$C_{next} \in C$	Returns from H_c the construction C_{next} that is not yet marked and it connected with the construction C_i or with its couple.
getElements	$sC \subset C$	$sE \subset E$	Returns a set of elements sE contained in a set of constructions C. One element for double elements.
getIntersection	$sC \subset C$ and $C_{next} \in C$	$sC_1 \subset sC$	Returns the set of constructions sC whose elements are linked with the element of the construction C_{next} in the lattice as well as the construction C_{next}.
getPatterns	$B \subset M$	$A \subset G$	Returns a set of patterns A of first level nodes that exploit the elements of the set B from lattice.

Table 10 Two similar patterns

Pattern	XSD constructions
27	$C_1 = complexType, (name, "ct_name")$. $C_2 = attribute, \{(name, "attr_name"), (type, "st_name")\}$.
28	$C_1 = complexType, (name, "ct_name")$ $C_2 = attribute, (ref, "attr_name")$

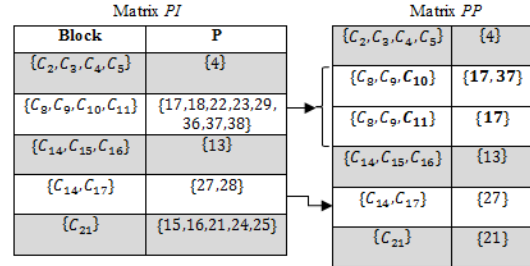

Fig. 4 Example of pertinent pattern selection results.

processed by the *getPertinentPatterns* procedure.

The procedure *getPertinentPatterns* (Algorithm 4) selects the pertinent pattern among the similar patterns set. Firstly, the procedure identifies among the six sets similar patterns the set which together will be concerned with finding the pertinent pattern. Secondly, it assigns to this set a specific process because each set has a particular difference. For instance, the difference in the set 4 (see Figure 3) lies in the element *extension*. By contrast, for the set 2 the difference lies in the element *xsd:attribute* which admits a *name* or *ref* attribute.

With the exception of the set 6 the procedure selects a single pertinent pattern and does not require a redistribution of the block. For example, from the PI matrix presented in the Table 8, the set of identified patterns for the block $\{C_8, C_9, C_{10}, C_{11}\}$ is $P_6 = \{17, 18, 22, 23, 29, 36, 37, 38\}$. However, the main algorithm (Algorithm 3) re-districts this block into sub-blocks according to the number of constructions which the XSD element is *element*. Afterwards, for each sub-blocks the method identifies the pattern 17, 18, 22 or 23. Once one of these patterns is identified, the method seeks if the element *element* containing a *maxOccurs* or *minOccurs* attribute. If so, then it yet identifies one or two patterns at a time: 36, 37, (36 and 37), or 38.

Figure 4 shows the result of the pertinent patterns selection algorithm applied on the matrix PI that is presented in Table 8.

The pertinent pattern selection algorithm uses some methods that are mentioned in Table 11.

Table 11 Description of the methods used by the pertinent pattern selection algorithm.

Method	Parameters	Return	Description
getTypeOfAttributeValue	$C_i \in C$, $a \in A$	Type	Returns the type of the value of attribute a in the construction c_i.
hasAttribute	$a \in A$, $C_i \in C$	True/ false	Returns true if the attribute a exists in the construction C_i otherwise false.
getValueOfAttribute	$a \in A$, $C_i \in C$	$v \in V_A$	Returns the value of attribute a in the construction C_i.

Algorithm 3: PertinentPatterns

```
input  : PI[n][2]          /* matrix of identified patterns */
output: PI[m][2]           /* matrix of pertinent patterns */
P_sim = {{15,16,24,25,21},{19,20},{6,8,9},
  {10,11,12},{27,28}} ;    /* sets of sub-set of similar patterns */
P_6 = {17,18,22,23,29,36,37,38} ;   /* set of similar patterns to be processed differently */
sP ⊂ G ;                   /* sub-set of patterns */
Block = {C_1,...,C_n} ; C_i ∈ C;
Block' ⊂ C;
P_pert ⊂ G ;               /* pertinent patterns */
n : lines number of the matrix PI;
isImbriked : false;
for (i=0 to n-1) do
    sP = PI[i][1] ;        /* return the i^th set of identified patterns */
    Block = PI[i][0] ;     /* return the i^th block of constructions */
    if sP ∈ P_sim then
        P_pert = getPertinentPatterns
          (Block, sP);
        PI[i][1] = P_pert;
    end
    if sP = P_6 then
        isImbriked = true;
        k = 0;
              /* redistribution of the block
        depending on the number of
        constructions starting from the 3^rd
        construction */
        for j=3 to size(block) do
            Block' = {C_1, C_2, C_j};
            P_pert = getPertinentPatterns
              (Block', sP);
            PI[i][1] = PI[i][1] ⋃ P_pert;
            PI[i][0] = PI[i][0] ⋃ Block';
        end
    end
end
if isImbriked then
    PP = addAll (PI);
end
PP = PI;
```

4.4 Ontology generation

As a result, the previous steps return a matrix of two columns where the first column includes construction blocks and the second column is composed of one or two identified patterns for each construction block. The ***OntoGeneration*** algorithm (Algorithm 5) takes as input this matrix, and then it calls for each XSD block the appropriate method that generates the OWL block corresponding to the identified pattern. To generate an OWL block, the extraction of data from the XSD block is required in the first place. Then, the extracted data is sent as parameters to the appropriate method that handles the generation of OWL blocks such as *writePatterns_1*, *writePattern_2*, *etc.* The creating methods of OWL blocks are parametrized according to the information necessary for each pattern. For example, the *writePattern_1* method that creates the pattern 1 requires only the name of the data type to create. We have not created 43 methods since there are methods that do the same processing. For example, the method *writePattern_14_21* (*class_name1*, *class_name2*) creates the class *class_name1* which is a subclass of *class_name2*. This process is valid for both patterns 14 and 21 (see Table III and IV in [3]). Figure 5 illustrates the principle of generating an OWL block corresponding to an identified pattern. The XSD block and the identified pattern are extracted from the *PP* matrix that presented in Figure 4.

Once all the OWL blocks are created, the *OntoGeneration* algorithm (Algorithm 5) uses *writeOWLFile* method to create the OWL file

Algorithm 4: getPertinentPatterns

```
input  : Block ⊂ C, P ⊂ G
output: {P_pert, P_pert2, P_pert3}
type : String; Block = {C_1, ..., C_n};
switch P do
    case {15, 16, 24, 25, 21} do
        type =
         getTypeOfAttributeValue(C_1,"type");
        if type="DataType" then P_pert = 24;
        else if type="simpleType" then
         P_pert=16;
        else if type="complexType" then
         P_pert=21;
        else P_pert=25;
    end
    case {27, 28} do
        if hasAttribute("name", C_2) then
         P_pert=27;
        else if hasAttribute("ref", C_2) then
         P_pert=28;
    end
    case {19, 20} do
        type =
         getTypeOfAttributeValue(C_3,"base");
        if type = "DataType" then P_pert = 19;
        else if type = "simpleType" then P_pert
         = 20;
    end
    case {6, 8, 9} do
        type=
         getTypeOfAttributeValue(C_3,"base");
        if type = "DataType" then P_pert = 6;
        else if type = "simpleType" then P_pert
         = 8;
        else if type ="complexType" then P_pert
         = 9;
    end
    case {10, 11, 12} do
        type =
         getTypeOfAttributeValue(C_3,"base");
        if type = "DataType" then P_pert = 10;
        else if type ="simpleType" then P_pert
         =11;
        else if type = "complexType" then
         P_pert =12;
    end
    case {17, 18, 22, 23, 29, 36, 37, 38} do
        if hasAttribute("name", C_3) then
            type =
             getTypeOfAttributeValue(C_3,"type");

            if type="DataType" or
             type="simpleType" then P_pert
             =17;
            else if type="complexType" then
             P_pert =22;
        else if hasAttribute("ref", C_3) then
            type =
             getTypeOfAttributeValue(C_3,
             "ref");
            if type ="DataType" or
             type="simpleType" then
             P_pert=18;
            else if type="complexType" then
             P_pert = 23;
        end
        if hasAttribute("minOccurs", C_3) and
         hasAttribute("maxOccurs", C_3) then
            x = getValueOfAttribute
             ("minOccurs",C_3);
            y = getValueOfAttribute
             ("maxOccurs", C_3);
            if x = y then P_pert2=38;
            else
                P_pert2=36;
                P_pert3=37;
            end
        else if hasAttribute("minOccurs", C_3)
         then P_pert2=36;
        else if hasAttribute("maxOccurs", C_3)
         then P_pert2=37;
    end
end
```

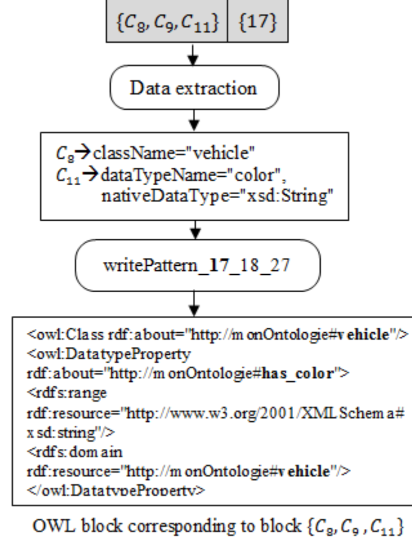

Fig. 5 Generation of an OWL block corresponding to an XSD block.

encompassing all OWL blocks. For this reason, to the large space that the methods of creating XSD blocks take, we can not give them in this paper. So, we give as examples some methods. The method ***writePattern_1*** (Algorithm 6) is used to create a datatype, the method ***writePattern_2*** (Algorithm 7) is used to process name spaces and several resources that are equivalent to a given resource. The method ***writePattern_4*** (Algorithm 8) implements the construction *owl:oneOf*. The method ***writePattern_6*** (Algorithm 9) allows the creation of a class and a data type property with its domain and range. In the following, we give the general algorithm *OntoGeneration* which will make calls of the different algorithms. These algorithms define the method *concat* that allows to concatenate two strings and the variables *ontoNameSpace* and *xmlNameSpace* by *"name_space_of_ontology"* and *"http://www.w3.org/2001/XMLSchema#"*, respectively.

We take again the example 2 and the identified patterns that are presented in Figure 4. The OWL ontology generated from this example is shown in Figure 6. We have divided this

Algorithm 5: OntoGeneration

```
input  : PP[n][2]            /* resulting matrix of algorithm 3 */
output: owlFile: file
type : String
Block ⊂ C
P_pert ⊂ G                   /* pertinent patterns */
n: line number of the matrix PP
st_name, data_restriction, member_type: String
enumeration_value_list, member_type_list: list
for j=0 to n-1 do
    Block = PP[j][0]         /* return a block's constructions */
    P_pert = PP[j][1]        /* return the pertinent pattern for the block's constructions */
    switch P_pert do
        case 1 do
            st_name = getValueOfAttribute(C1, "name")    /* return the name of simpleType element */
            writePattern1(st_name)    /* calls of algorithm 6 */
        end
        case 2 do
            st_name = getValueOfAttribute (C1, "name")    /* return the name of simpleType element of the first construction C1 */
            member_type_list = getValueOfAttribute (C2, "memberType")    /* returns the value of the attribute memberType of the union elements of the second construction C2 */
            member_type_list = split (member_type)    /* separation of the types combined */
            writePattern2(st_name, member_type_list)    /* calls of algorithm 7 */
        end
        case 4 do
            st_name = getValueOfAttribute (C1, "name")    /* return the name of simpleType element */
            base_restriction = getValueOfAttribute (C2, "base")
            /* returns the datatype that is the value of "base" attribute and restriction element */
            /* put the enumeration values in a list */
            for k = 3 to size(Block) do
                enumeration_value_list = enumeration_value_list ∪ getValueOfAttribute(Ck, "value")
            end
            writePattern4(st_name, base_restriction, enumeration_value_list) /* calls of algorithm 8 */
        end
        case 6 do
            ct_name = getValueOfAttribute(C1, "name")    /* return the name of complexType element */
            base_extension = getValueOfAttribute(C3, "base")
            /* returns the datatype that is the value of the "base" attribute and the extension element */
            writePattern6(ct_name, base_extension)    /* calls of algorithm 9 */
        end
        case 43 do
            ...
        end
    end
end
owlFile = writeOWLFile() /* creation of the OWL file */
```

Algorithm 6: writePatterns_1

```
input  : dataTypeName      /* data type name */
output: ontology model
create Resource that type is RDFS.Datatype and uri is concat(ontoNameSpace, dataTypeName)
```

Algorithm 7: writePatterns_2

```
input  : dataTypeName: String, equivalent DataTypeList: List
output: ontology model
namespace: String
create Resource that name is resource1, type is RDFS.Datatype and uri is concat(ontoNameSpace, dataTypeName)
for each equivalentDataTypeName in equivalentDataTypeList do
    if equivalentDataTypeName contain ":" then nameSpace = xmlNameSpace;
    else
        nameSpace = ontoNameSpace
    end
    create Resource that name is resource2, type is RDFS.Datatype and uri is concat (namespace, equivalentDataTypeName)
    resource1 has equivalent class resource2
    /* creation of the equivalence relation between the data types resource1 and resource2 */
end
```

Algorithm 8: writePatterns_4

```
input  : dataTypeName: String, nativeDataType: String, oneOfValueList: List
output: ontology model
create Resource that name is resource1, type is RDFS.Datatype and uri is concat(ontoNameSpace, dataTypeName)
create RDFList that name is rdfList1
for each oneOfValue in oneOfValueList do
    create Literal that name is literal1, object is oneOfValue and type is concat(xmlNameSpace, nativeDataType)
    rdfList1 = cons(literal1)    /* assign literal1 to the rdfList1 */
end
create DataRange that name is dataRange1, uri is rdfList1
add property witch property type is equivalentClass and RDFNode is dataRange1 to resource1
```

ontology using circles with modified points. Each circle represents an ontology fragment resulting from the transformation of an XSD block using the identified patterns. The identified XSD blocks and patterns are presented by vectors fused with the circles.

As can be seen from the ontology presented in Figure 6, the proposed method provides more semantics for existing data by adding more

Algorithm 9: writePatterns_6

```
input  : className: String, nativeDataType:
         String
output: ontology model
dataTypePropertyName: String
create OntClass that name is class, uri is
  concat(ontoNameSpace, className)
dataTypePropertyName = concat("has_",
  className)
create DataTypeProperty that name is dtp, uri is
  concat(ontoNameSpace, dataTypePropertyName)
domain of dtp is class1
range of dtp is concat(xmlNameSpace,
  nativeDataType)
```

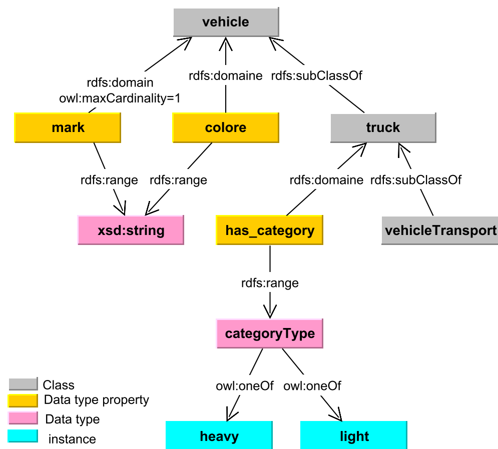

Fig. 6 Ontology generated from the example 2.

vocabularies to describe the relationship between classes and between the class and its properties. Since our method considers all XSD elements in the transformation process, it does not need user intervention to complete the transformation.

5 Implementation of PIXCO prototype

The prototype of the transformation method is called **PIXCO** (**P**atterns **I**dentification for **X**SD **C**onversion to **O**WL). It takes as input an XSD schema and gives in output two results: the set of XSD construction blocks where each block is accompanied by the pertinent pattern, and the OWL file containing the resulting ontology. Figure 7 presents the general architecture of PIXCO. The architecture is based

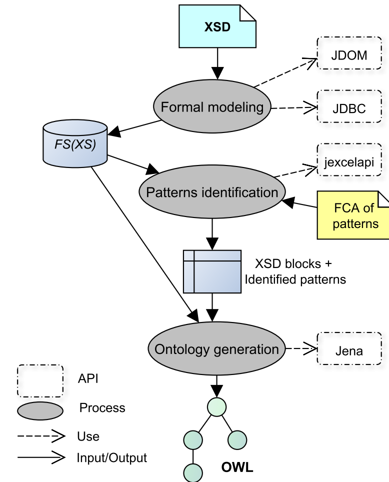

Fig. 7 PIXCO Architecture

on three processes which are the main treatments of the proposed method.

The first process of modeling uses the JDOM[3] API to manipulate the XSD document. It traverses the XSD tree from the root and extracts the information from each node (element name, list of attributes). This information will be stored in a database using the JDBC[4] API. This database saves XSD constructs in $\mathcal{FS}(\mathcal{XS})$ format. It used by methods manipulating XSD constructs such as *markConstruction*, *hasConstructionNotMarked*, *hasAttribute*, *getTypeOfAttributeValue*, etc. (Table 9 and 11).

The second process of patterns identification uses the jexcelapi[5] API to manipulate the FCA context of forty-three patterns. This process takes care of all methods manipulating patterns such as *getIntersection* and *getPatterns* (Table 11), and thus the implementation of algorithms 1, 2, 3 and 4.

The last process generates the OWL file containing the ontology resulting from the transformation. It implements OWL ontology generation algorithms such as algorithms 5, 6, 7,

[3] http://www.jdom.org/.
[4] http://docs.oracle.com/javase/7/docs/technotes/guides/jdbc/jdbc_41.html.
[5] http://jexcelapi.sourceforge.net/.

Table 12 Performance Measures of PIXCO for UBL Dataset.

N	Dataset	XSD Const. nb	Patt. Ident. (sec.)	OWL Gen. (sec.)
1	UBL-CodeList-SubstitutionStatusCode-1.0	19	97	33
2	UBL-CoreComponent-Parameters-1.0	86	7	1
3	UBL-CodeList-Country-IdentificationCode-1.0	256	96	1
4	UBL-CommonBasic-Components-1	356	26	8

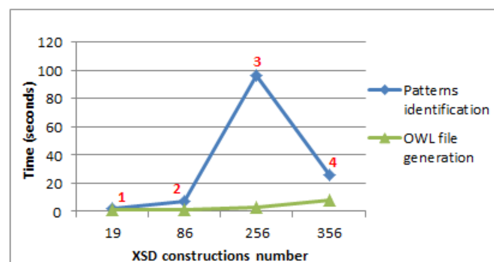

Fig. 8 Graph Comparing the execution time of the three processes of PIXCO with the XSD constructions number of UBL dataset.

etc. It uses the Jena[6] API to create Ontology properties (Class, DataTypeProperty, ...).

6 PIXCO discussion

For the purpose of evaluating our approach on the use of cases, we use the datasets available on the W3C website[7]. The evaluation is performed for UBL schemas which are in different XSD construct numbers. Table 12 presents the sets of schemas to be tested. These schemas are referenced by numbers because they have long names.

For each schema we have executed, at first, the process of patterns identification (Algorithms 1, 2, 3 and 4), then, the OWL ontology generation process (algorithm 5) is launched. For both processes we have marked the execution time in table 12. The graph given in Figure 8 shows that the time taken for the identification of the transformation patterns does not increase linearly with respect to the number of schema constructs given in input. On the other hand, the time taken for the generation of ontology increases linearly with respect to the number of schema constructs given in input.

We have found that the time required to identify the patterns for scheme 3 is greater than that for Scheme 4, even though the number of constructs of schema 3 is larger than that in scheme 4. This is because the process of patterns identification has identified an XSD block consisting of 241 constructs for schema 3. This block concerns a simple type enumerated by a large number of values. In addition, the maximum time required for the pattern identification process is 96 seconds. While the maximum time required for the ontology generation process is 8 seconds. To this end, the time required for the transformation depends not only on the number of XSD schema constructs, but also on the structure of the XSD blocks to be identified. As well as the patterns identification process requires a lot of time compared to the ontology generation process.

Other approaches discussed in the related works do not have working prototypes readily available, on which such experiments for evaluation could be done. Due to this, we were unable to give any comparative performance measures between our approach and others.

Although the PIXCO prototype processes the maximum of XSD constructs, it will be a little cumbersome for simple XSD files that contain the most used XSD elements such as elements, attributes, and types (simple and complex). For example, for a simple XSD file, the prototype must seek the transformation patterns by traversing all forty-three patterns for each XSD construct. It may be necessary for future research to add a configuration before starting the transformation. This configuration consists of checking the XSD elements consid-

[6] http://jena.sourceforge.net/.
[7] https://www.w3.org/XML/Binary/2005/03/test-data/

ered in the transformation. This reduces the number of patterns to be traveled for the identification of the transformation patterns.

Concerning the complexity of the implementation algorithms of PIXCO is always polynomial. It depends on the number of XSD schema constructs to be transformed. If we note that n is the number of XSD constructs, then the complexity of the algorithms is $O(n^k)$ where $1 < k \leq 5$, The constant k is dependent on each algorithm.

7 Conclusion

A formal method to transform XSD schema into OWL schema using patterns is presented in this paper. The transformation patterns proposed by the Janus method are first expanded to allow the maximum transformation of XSD schema constructions. However, the high number of patterns makes the transformation process more complex. Thus, a formal model was proposed to transform an XSD schema construction into OWL ontology. Concerning the modeling of patterns, we proposed the FCA context to represent all of the 43 patterns by objects and the XSD elements (*xsd:element*, *xsd:attribute*, *xsd:all*, *etc.*) by attributes allowing the evaluation of all possible sequences of XSD constructions blocks. For the formalization of the XSD schemas, we proposed a mathematical model called, $\mathcal{FS}(\mathcal{XS})$ that provides a formal manipulation of the XSD schema constructions using sets, functions and applications. This model can also be used freely in different contexts. Based on this modeling, we proposed an algorithm for the identification of the most appropriate patterns to transform each XSD construction block. For a construction block, the algorithm can identify a set of patterns that are similar. These patterns are formed by the same XSD elements. But each pattern has a particular difference on XSD attributes. To process this problem, we proposed another algorithm that selects the appropriate pattern for each XSD block. For ontology generation process, we proposed an algorithm that generates for each XSD construction block an OWL block according to the pattern that are previously identified. And it finally creates the OWL file encompassing OWL blocks generated. To validate our method of transformation, we proposed a prototype named PIXCO. This prototype takes as input an XSD schema and provides in output three results: the $\mathcal{FS}(\mathcal{XS})$ model corresponding to the XSD schema, all blocks of XSD schema where each block is accompanied by relevant patterns, and the OWL file containing the resulting ontology. In order to assess the main features of the principal proposed algorithms of PIXCO, we gave some experimental tests using datasets.

The proposed method finds its interest in virtual integration systems and the generation of ontology models or terminology without the generation of the assertion level. However, we envisage in our future work to process the XML instances in order to use our approach in a broader context such as the automatic construction of data warehouses.

References

1. Agreste, S., De Meo, P., Ferrara, E., Ursino, D.: Xml matchers: approaches and challenges. Knowledge-Based Systems **66**, 190–209 (2014)
2. Baclawski, K., Kokar, M.K., Kogut, P.A., Hart, L., Smith, J., Letkowski, J., Emery, P.: Extending the unified modeling language for ontology development. Software and Systems Modeling **1**(2), 142–156 (2002)
3. Bedini, I., Matheus, C., Patel-Schneider, P.F., Boran, A., Nguyen, B.: Transforming xml schema to owl using patterns. In: Semantic Computing (ICSC), 2011 Fifth IEEE International Conference on, pp. 102–109. IEEE (2011)
4. Belghiat, A., Bourahla, M.: Transformation of uml models towards owl ontologies. In: Sciences of Electronics, Technologies of Information and Telecommu-

nications (SETIT), 2012 6th International Conference on, pp. 840–846. IEEE (2012)
5. Bian, J., Zhang, H., Peng, X.: The research and implementation of heterogeneous data integration under ontology mapping mechanism. In: International Conference on Web Information Systems and Mining, pp. 87–94. Springer (2011)
6. Bohring, H., Auer, S.: Mapping xml to owl ontologies. Leipziger Informatik-Tage **72**, 147–156 (2005)
7. Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F.: Extensible markup language (xml). World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210 **16**, 16 (1998)
8. Brockmans, S., Colomb, R.M., Haase, P., Kendall, E.F., Wallace, E.K., Welty, C., Xie, G.T.: A model driven approach for building owl dl and owl full ontologies. In: International Semantic Web Conference, pp. 187–200. Springer (2006)
9. Cruz, C., Nicolle, C.: Ontology enrichment and automatic population from xml data. ODBIS **2008**, 17–20 (2008)
10. De Meo, P., Quattrone, G., Terracina, G., Ursino, D.: Integration of xml schemas at various 'severity' levels. Information Systems **31**(6), 397–434 (2006)
11. Doan, A., Noy, N.F., Halevy, A.Y.: Introduction to the special issue on semantic integration. ACM Sigmod Record **33**(4), 11–13 (2004)
12. Duchateau, F., Bellahsene, Z.: Yam: A step forward for generating a dedicated schema matcher. In: Transactions on Large-Scale Data-and Knowledge-Centered Systems XXV, pp. 150–185. Springer (2016)
13. Duchateau, F., Bellahsene, Z., Hunt, E.: Xbenchmatch: a benchmark for xml schema matching tools. In: Proceedings of the 33rd international conference on Very large data bases, pp. 1318–1321. VLDB Endowment (2007)
14. El Hajjamy, O., Alaoui, K., Alaoui, L., Bahaj, M.: Mapping uml to owl2 ontology. Journal of Theoretical and Applied Information Technology **90**(1), 126 (2016)
15. Ferdinand, M., Zirpins, C., Trastour, D.: Lifting xml schema to owl. In: Web Engineering, pp. 354–358. Springer (2004)
16. Fu, G.: Fca based ontology development for data integration. Information Processing & Management (2016)
17. Ganter, B., Wille, R.: Formal concept analysis: mathematical foundations. Springer Science & Business Media (2012)
18. Ghawi, R., Cullot, N.: Database-to-ontology mapping generation for semantic interoperability. In: VDBL07 conference, VLDB Endowment ACM, pp. 1–8 (2007)
19. Ghawi, R., Cullot, N.: Building ontologies from xml data sources. In: DEXA Workshops, pp. 480–484 (2009)
20. Gu, J., Zhou, Y.: Ontology fusion with complex mapping patterns. In: International Conference on Knowledge-Based and Intelligent Information and Engineering Systems, pp. 738–745. Springer (2006)
21. Haav, H.M.: A semi-automatic method to ontology design by using fca. In: CLA. Citeseer (2004)
22. Hacherouf, M., Nait-Bahloul, S., Cruz, C.: Transforming xml documents to owl ontologies: A survey. Journal of Information Science **41**(2), 242–259 (2015)
23. Hitzler, P., Krötzsch, M., Parsia, B., Patel-Schneider, P.F., Rudolph, S.: Owl 2 web ontology language primer. W3C recommendation **27**(1), 123 (2009)
24. Kalfoglou, Y., Schorlemmer, M.: Ontology mapping: the state of the art. The knowledge engineering review **18**(01), 1–31 (2003)
25. de Laborda, C.P., Conrad, S.: Relational. owl: a data and schema representation format based on owl. In: Proceedings of the 2nd Asia-Pacific conference on Conceptual modelling-Volume 43, pp. 89–96. Australian Computer Society, Inc. (2005)

26. Lacoste, D., Sawant, K.P., Roy, S.: An efficient xml to owl converter. In: Proceedings of the 4th India software engineering conference, pp. 145–154. ACM (2011)
27. Liu, Q., Huang, T., Liu, S.H., Zhong, H.: An ontology-based approach for semantic conflict resolution in database integration. Journal of Computer Science and Technology **22**(2), 218–227 (2007)
28. Madhavan, J., Bernstein, P.A., Rahm, E.: Generic schema matching with cupid. In: VLDB, vol. 1, pp. 49–58 (2001)
29. O'Connor, M.J., Das, A.: Acquiring owl ontologies from xml documents. In: Proceedings of the sixth international conference on Knowledge capture, pp. 17–24. ACM (2011)
30. Pinto, H.S., Martins, J.P.: Ontologies: how can they be built? Knowledge and information systems **6**(4), 441–464 (2004)
31. Rahm, E., Do, H.H., Maßmann, S.: Matching large xml schemas. ACM SIGMOD Record **33**(4), 26–31 (2004)
32. Rodrigues, T., Rosa, P., Cardoso, J.: Mapping xml to existing owl ontologies. In: International Conference WWW/Internet, pp. 72–77. Citeseer (2006)
33. Shvaiko, P., Euzenat, J.: Ontology matching: state of the art and future challenges. IEEE Transactions on knowledge and data engineering **25**(1), 158–176 (2013)
34. Sure, Y., Tempich, C., Vrandecic, D.: Ontology engineering methodologies. Semantic Web Technologies: Trends and Research in Ontology-based Systems pp. 171–190 (2006)
35. Thuy, P.T.T., Lee, Y.K., Lee, S.: Dtd2owl: automatic transforming xml documents into owl ontology. In: Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human, pp. 125–131. ACM (2009)
36. Touzi, A.G., Massoud, H.B., Ayadi, A.: Automatic ontology generation for data mining using fca and clustering. arXiv preprint arXiv:1311.1764 (2013)
37. Tsinaraki, C., Christodoulakis, S.: Interoperability of xml schema applications with owl domain knowledge and semantic web tools. In: On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS, pp. 850–869. Springer (2007)
38. Wache, H., Voegele, T., Visser, U., Stuckenschmidt, H., Schuster, G., Neumann, H., Hübner, S.: Ontology-based integration of information-a survey of existing approaches. In: IJCAI-01 workshop: ontologies and information sharing, vol. 2001, pp. 108–117. Citeseer (2001)
39. Welty, C., McGuinness, D.L., Smith, M.K.: Owl web ontology language guide. W3C recommendation, W3C (February 2004) http://www. w3. org/TR/2004/REC-owl-guide-20040210 (2004)
40. Yahia, N., Mokhtar, S.A., Ahmed, A.: Automatic generation of owl ontology from xml data source. arXiv preprint arXiv:1206.0570 (2012)
41. Zhang, L., Li, J.: Automatic generation of ontology based on database. Journal of Computational Information Systems **7**(4), 1148–1154 (2011)