Interoperability support between OWL 2.0 and XML environments

Ioannis N. Stavrakantonakis

Department of Electronic & Computer Engineering

Technical University of Crete

Chania 2010

Interoperability support between OWL 2.0 and XML environments

This thesis submitted to the Department of Electronic & Computer Engineering of the Technical University of Crete in partial fulfillment of the requirements for the Diploma of Electronic Engineer and Computer Engineer.

Author: Ioannis N. Stavrakantonakis

Supervisor: Prof. Stavros Christodoulakis

Lab. Of Distributed Multimedia Information Systems and Applications

Department of Electronic & Computer Engineering

Technical University of Crete

Greece

Chania 2010

Visit the I.Stavrakantonakis Web site at
http://www.istavrak.com

and the Lab. MUSIC/TUC at
http://www.music.tuc.gr

This thesis is dedicated to my family.

Abstract

Since the dawn of the Semantic Web technologies, a massive amount of possibilities and new semantics appeared in the field of the web of data. The Semantic Web is developing in a completely different environment compared to the one that we have already get used to and the software industry has worked on during the very recent years.

Since different communities in the industry and the academia work and are familiar with either the Semantic Web environment or the XML environment, a Schema mapping framework and a Query mapping framework are needed that will allow querying data repositories, multimedia content and service descriptions in a uniform way in both the XML and Semantic Web environments.

This thesis presents the SPARQL2XQuery 2.0 framework, which allows expressing semantic queries on top of XML data through the translation of SPARQL queries in XQuery syntax and the XS2OWL 2.0 framework for uplifting XML Schemas to OWL 2 ontologies. SPARQL2XQuery 2.0 may work with both existing ontologies and with automatically produced ones, formed according to the XS2OWL 2.0 transformation model. XS2OWL 2.0 exploits the OWL 2.0 semantics and supports the new XML constructs introduced by XML Schema 1.1.

The un-queried life is not worth living
Socrates (Plato, The Apology, 38a)

Acknowledgements

I would like to thank my supervisor, Prof. Stavros Christodoulakis, for his encouragement and his continuous guidance and support throughout my research. I would like also to thank him for the important experiences he offered me during my stay at the Laboratory of Distributed Multimedia Information Systems and Applications (MUSIC).

I would like to thank my co-supervisor, Dr. Chrisa Tsinaraki, for her continuous guidance and solid collaboration throughout my research, as well as the long lasting discussions regarding this thesis. I merited her exemplar teamwork spirit and the detailed corrections and comments during the writing session of this thesis.

I would like to express my gratitude to the readers of this thesis Mr. Antonios Deligiannakis and Mr. Michail G. Lagoudakis for the time they devoted and their critical evaluation.

My appreciation goes to Nektarios Gioldasis for his valuable help regarding this thesis. I am also grateful to Nikos Bikakis for being always ready to offer his help whenever needed.

I would like to thank all my colleagues in Lab so much for our excellent cooperation, as for the pleasant environment in the Laboratory.

Finally, I wish to thank my family and Giasemi for helping me get through the difficult times, and for all the emotional support, as well as all of those people that were close to me during this period of time for the entertainment, and caring they provided.

Ioannis N. Stavrakantonakis
Technical University of Crete
Chania 2010

Contents

List of Figures

List of Tables

1. Introduction

The availability of internet based communication opportunities and the massive usage of the web during the last decade has led the volume of the information and data available on the web to grow at an explosive pace. At the same time, organizations all around the world have developed systems relying on different standards in order to serve soundly different needs. These parameters add complexity and create heterogeneity inside the ecosystem of the computer networks, the World Wide Web and the data transactions. These challenges are the motivation of the Semantic Web. The Semantic Web is the greatest attempt to bridge the different gaps among systems in order to function within interoperability in global scope.

The Semantic Web was introduced by the president and founder of the World Wide Web, Tim Berners-Lee. In 1998, Tim Berners-Lee presented his Semantic Web vision in his book “Weaving the Web” [TimWW]. The Semantic Web was described there as a web of data automatically analyzed and used by applications based on the semantics and the meaning of data.

The Semantic Web technologies and languages support the Semantic Interoperability by transforming the web into semantically rich structures. In the core of the semantic interoperability

is **ontology**[1]. An **ontology** is a set of precise descriptive statements about some part of the world (usually referred as the domain of knowledge or the subject matter of the ontology). Precise descriptions satisfy several purposes: most notably, they prevent misunderstandings in human communication and they ensure that the software behaves in a uniform, predictable way and works well with other software. The dominant language for describing ontologies is OWL (Web Ontology Language)[W3C/OWL]. It is worthwhile to mention that OWL is not a programming language: OWL is declarative, i.e. it describes a state of affairs in a logical way. Appropriate tools (so-called reasoners) can then be used to infer further information about that state of affairs.

Information about resources in the Semantic Web is represented using the RDF language (Resource Description Framework)[W3C/RDF]. RDF data can be queried with a great set of querying languages -e.g. RQL, RDFQ, DQL etc.- developed for this purpose with the most well known and broadly used, the SPARQL W3C standard (Simple Protocol and RDF Query Language)[W3C/SPARQL].

The great interest of the academia and the industry in semantic web has led to new W3C recommendations of existing languages and technologies in order to cover the needs of different domains. OWL has evolved in OWL 2.0 by adding even more expressiveness and potential of reasoning over the axioms of the ontology.

On the other hand, the dominant standard in data exchange among systems and web services is the XML (Extensible Markup Language)[W3C/XML]. The XML documents structure is described with XML Schema, which supports the definition of complex structures and datatypes in order to cover all the needs. XML data are queried with XQuery (XML Query Language)[W3C/XQUERY]. The flexibility and the advanced structure-description capabilities of the XML Schema have made it a de facto standard in data and metadata description in several application domains like, for example, the multimedia domain, e-learning domain and digital libraries. Thus, XML is the base of Syntactic Interoperability and XML Schema allows for the Structural Interoperability.

Motivation

Since these environments (Semantic Web and XML) and their developed infrastructures are based on different data models and use different query languages to access them, it is crucial to develop mechanisms that allow bridging the two environments and exploiting the information stored in both sides. Legacy systems and Semantic web must form one solid and unified web. Furthermore, it is unrealistic to expect that all the existing XML data will be transformed to RDF data. Thus, it is crucial for the Semantic Web to make feasible the retrieval of data from legacy data sources like XML repositories via the Semantic Web query language, the SPARQL. The translation of SPARQL queries into the query language of XML data, in order to bridge the gap between these environments, constitutes the motivation of this thesis.

[1] *In the domain of philosophy, ontology is the study of being, existence or reality in general, as well as the basic categories of being and their relations.*

Consider for example, a legacy digital library of a cultural heritage institution that manages multimedia documents referring to cultural objects, using the MPEG-7 standard. Also, consider a digital library of a cultural heritage institution that exploits the SW technologies and encodes their cultural objects using the CIDOC/CRM standard [CIDOC/CRM]. It is crucial for Semantic Web applications and users to be able to query and retrieve the XML data stored as MPEG-7 descriptions by using Semantic Web technologies like the SPARQL query language, utilizing the CIDOC/CRM ontology that they are familiar with. In this case, the SW users should not have to interface with more than one model or query languages.

Related work and contribution

This thesis aims to implement a robust system able to support interoperability between OWL 2.0 and XML environments by allowing the users to query XML Data repositories with using the SPARQL query language, which allows accessing OWL/RDF repositories.

This is the sequel of two individually developed frameworks in the MUSIC laboratory, namely XS2OWL and SPARQL2XQuery. These two frameworks can be regarded as the main components of one integrated system, which allows the expression of semantic queries on top of XML data through the translation of SPARQL queries in XQuery syntax. This integrated system has been developed in the context of this thesis.

In this thesis, the new features of XML Schema 1.1 are also described, as well as the new features of OWL 2.0 but only the subset of the latter that are relevant with architecture. Furthermore, this thesis describes the set of transformations that are applied in the XML Schema constructs in order to generate equivalent OWL syntax. These are illustrated using a broad set of examples in order to cover every single condition.

Some work has been done on the translation of XML Schemas into OWL syntax. Some of them use the XSLT for the transformation from XML Schema to OWL syntax, like [BoAu05] and [RoRoCar06] in contrast to [FeZiTr04], [GaCe05], [CrNiChr08], [AnIvMar07] that do not prefer XSLT-based Transformation. All these solutions, including XS2OWL 1.0, are based on OWL 1.0. OWL 1.0 does not offer the means to explicitly define a new datatype. This restriction of the OWL 1.0 syntax led the authors of the existing research work to ignore Simple Types, to partially support them [AnIvMar07] or to find a way to support them with extra information out of the main ontology like the previous version of the XS2OWL framework [TsCh07].

Taking all these drawbacks and restrictions in consideration, the XS2OWL 2.0 framework was developed on OWL 2.0 and ready to support XML Schema 1.1[(1)] ([W3C/XSD1.1a], [W3C/XSD1.1b]).

---

[1] *In the owl 2.0 Recommendation it is suggested to the developers and users to follow the XSD 1.1 Candidate Recommendation [W3C/XSD1.1a]. It is not expected any implementation changes will be necessary as XSD 1.1 advances to Recommendation.*

The XS2OWL 2.0 Framework, one of the components of the integrated system, aims to support semantic interoperability between OWL and XML. XS2OWL 2.0 is based on the transformation model XS2OWL and comprises the new version of previously developed XS2OWL [TsCh07]. The first version of XS2OWL supported the XML Schema 1.0 and OWL 1.0 W3C Recommendations. Both of these two technologies have advanced to newer versions since then. XML Schema 1.0 has evolved in XML Schema 1.1[1] and OWL 1.0 in OWL 2.0[2]. The latter is a recommendation and the former is a working draft at the time of writing this diploma thesis.

In the new version of XS2OWL, the representation of any XML construct in OWL was redesigned in order to be represented in a better way, more semantically consistent by exploiting the new features of OWL 2.0. Moreover, this thesis marks all the changes in XML Schema 1.1 and reforms the transformation model in a way to support the new functionality of XML Schema. The XS2OWL architecture is trying to transform every XML Schema construct into a semantically equivalent OWL construct.

Moreover, XS2OWL 2.0 allows the representation of identity constraints like key, keyref and unique. In order to achieve this, an algorithm was created to evaluate a simple[3] XPath ([W3C/SelXSD], [W3C/FiXSD]) over an XML Schema. Furthermore, XPath evaluator can be developed into a separate XSLT library useful for discovering the definition, in an XML Schema, of the XML element or attribute indicated by XPath expressions. In conclusion, XS2OWL 2.0 exploits the OWL 2.0 semantics and supports the new XML constructs introduced by XML Schema 1.1.

Finally, to the best of our knowledge there is no work addressing the SPARQL to XQuery translation. Given the very high importance of XML and the related standards in the web, this is a major limitation in the existing research. This thesis introduces the SPARQL2XQuery 2.0 framework in order to bridge this interoperability gap. The SPARQL2XQuery 2.0 framework may work with both existing ontologies and with automatically produced ones, formed according to our XS2OWL 2.0 transformation model. SPARQL2XQuery 2.0 exploits the OWL 2.0 semantics and supports the new XML constructs introduced by XML Schema 1.1. In particular, the XML Schema identity constraints can now be accurately represented in OWL 2.0 syntax (which was not possible in OWL 1.0 syntax), thus overcoming the most important limitation of the XS2OWL 1.0 framework [TsCh07][TsChODB07] and, consequently, of the SPARQL2XQuery 1.0 framework [Bik08]. Moreover, XS2OWL 2.0 allows defining datatypes in OWL 2.0 syntax and consequently represents accurately XML Schema simple types in OWL 2.0 syntax, thus overcoming the limitation in the query translation model of SPARQL2XQuery 1.0, which did not treat simple types.

Thesis structure

The rest of this thesis is structured as follows:

---

[1] XML Schema 1.1 is a W3C Working Draft since 3rd December 2009. [W3C/XSD1.1a]
[2] OWL 2.0 is W3C Recommendation since 27th October 2009. [W3C/OWL2]
[3] Identity constraints use only a subset of XPath expressions. This subset is described in following sections and includes only path expressions.

- **Chapter 2** presents the set of background technologies and standards on which this thesis relied through the requirement analysis, design and implementation. Chapter 2 also includes a description of the previously developed frameworks, XS2OWL 1.0 and SPARQL2XQuery 1.0.
- **Chapter 3** describes the XS2OWL 2.0 framework, the components that comprise it and the processes taking place through the execution of it on inputs. Furthermore, XS2OWL 2.0 is presented as extension of XS2OWL 1.0. Thus, this report avoids mentioning unchanged features, but refers to them.
- **Chapter 4** introduces the SPARQL2XQUERY 2.0 framework, the architecture of the framework and any changes we performed in order to exploit the OWL 2.0 semantics and new products' structure of XS2OWL 2.0 framework. Thus, this report avoids mentioning the unchanged features since SPARQL2XQUERY 1.0, but refers to them.
- **Chapter 5** shows the implementation issues, the problems were overcome during the development of the XS2OWL 2.0 and SPARQL2XQuery 2.0 frameworks, the technologies that were used to develop each one of them and how the integrated framework can be used in real world applications of the Semantic Web and preexisting domains of knowledge.
- Finally, **chapter 6** presents the conclusion of the thesis and some proposed future work that can be applied to the presented work in order to extend even more.

2. Background Technologies

This thesis stands between two environments, the XML and the OWL 2.0, which make different assumptions, the **Closed World Assumption (CWA)** and the **Open World Assumption (OWA)** respectively, about the data and the semantics of the relationships between the web of data. There are essential differences between the two worlds, the Open and the Closed. The *open world assumption*, made by the Semantic Web technologies, tries hard to avoid the process of validation and anything that is done for validation reasons. In an open world assumption, validation is very limited. On the other hand the *closed world assumption*, made by the XML technologies, is much stricter and validates everything according to data types and rules.

Furthermore, there are some use cases that clarify the way of thinking in each of the two assumptions. The closed world assumption implies that everything we do not know is false, while the open world assumption states that everything we do not know is undefined. In the open world assumption the absence of a particular statement within the web means that the statement has not been made explicitly yet, independently of whether it would be true or not, and independently of whether it is (or would be) true or not. In essence, only from the absence of a statement, a deductive reasoner cannot (and must not) infer that the statement is false. An example of this difference is demonstrated in the example shown below:

Example: Let the statement *"Yannis is an undergraduate student of TUC"* be described with the appropriate constructs by both the Open World Assumption and the Closed World Assumption. Also, consider the question *"Is Yannis an undergraduate student of MUSIC?".*

Statement: *"Yannis is an undergraduate student of TUC"*
Question: *"Is Yannis an undergraduate student of MUSIC lab?"*

Answer of a reasoner in OWA: *"unknown / I can't tell"*
Answer in CWA: *"No"*

In OWA, the failure to derive a fact does not imply the opposite. This fact is due to the assumption of the Open World about the completeness of the knowledge in a specific field. In particular, a reasoner in the Open World assumes that the already described knowledge is not and will never be complete. Moreover, the Open World Assumption tries not to trigger any errors; instead of adding new information, it never falsifies a previous conclusion. This is shown in the next example.

Example: Consider the following statement, which is expressed using Semantic Web technologies (OWL, RDF) *"Yannis" "hasFather" "Nikos"*, where the cardinality of the *"hasFather"* relationshop is one. Consider that this statement is described with the appropriate constructs by both the Open World Assumption and the Closed World Assumption. Now, suppose that a new erroneous entry is created and merged with the existing model that is comprised of the statement *"Yannis" "hasFather" "Ioanna"* (while Ioanna is Yannis' mother and the model has already declared Yannis' father).

Statement: *"Yannis" "hasFather" "Nikos"*
Restriction: *"hasFather"* has exact cardinality of 1
False entry: *"Yannis" "hasFather" "Ioanna"*

Response in OWA: The creation of a *new statement* is triggered. There can only be one father, thus "Nikos" is the "same as" "Ioanna". This reasoning would be inconsistent if the "Nikos" and "Ioanna" individuals were explicitly specified as different.

Response in CWA: An *error* is triggered as there can be only one father and the model contains two statements that are in conflict.

Consequently, both of these assumptions have pros and cons, but they are really hard to mix. This discussion is presented at this point in order to clarify that this thesis aims to develop an interoperable framework between these two different environments.

The rest of this chapter describes the XML 1.0 in section 2.1, the XML Schema 1.0 in section 2.2, the XML Schema 1.1 in section 2.3, the XPath in section 2.4, the XQuery in section 2.5, the Notifications and conventions in section 2.6, the RDF/S in section 2.7, the OWL 1.0 in section 2.8, the OWL 2.0 in section 2.9, the SPARQL in section 2.10, the XS2OWL 1.0 framework in section 2.11, the SPARQL2XQuery 1.0 framework in section 2.12 and a conclusion in section 2.13.

2.1. XML 1.0

It is common for XML (Extensible Markup Language) [W3C/XML] to be used in interchanging data over the Internet. XML is a markup language for documents containing structured information and was designed for data encoding and delivery. A markup language is a mechanism to specify structures in a document. The XML specification defines a standard way to add markup to documents.

XML is a flexible markup language that programmers can modify to cover their needs. That is, programmer decides the XML tags that describe data rather than having to adhere to a standard set of tags as he does with HTML. It is worthwhile to mention that XML describes the data itself in contrast to HTML that describes how data should look on the screen. This flexibility allows the companies to create their own standard tags to describe data that is particular to their business.

XML is the most preferred way to transfer data between web applications, web services and systems based on web interaction for a couple of reasons. XML represents data, as a consequence, the data can be easily shared among different kinds of applications that run on different operating systems as it is independent from operating systems, transfer protocols and organizations and it is compatible with the majority of telecommunication protocols. Moreover, XML can be compressed in high levels in order to achieve faster transmission via networks.

2.1.1. XML Document, Elements and Attributes

An XML document contains nested elements; this way the relations between the tags of the XML document are implied. This hierarchy creates a tree of element nodes that depicts the data.

Every single element is enclosed in the angled brackets structural delimiters (< >), and always have an open (<) and closed markup tag (</). Child elements are placed within the open and closed markup tags of a parent element, and information is placed within the open and closed markup tags of a child element. The content type of an element may be plain text, a set of elements or combination of text and elements. An XML document must have one and only one root element, which includes the remaining elements of the data representation. Improperly nesting elements and orphan closing tags form an invalid xml document. A comment is information in the XML document that's typically not part of the actual data and is enclosed in the (<!--) and (-->) constructs.

Element tags may include *attributes*. An attribute is information that modifies an XML markup tag. Attributes are placed within the opening markup tag. One may create as many attributes as required; however, each attribute must have a unique name and a value contained within quotations (attributeName="value"). Each name/value pair must be separated by a space. XML allows the flexibility to create custom attributes.

Element names and attribute names cannot contain whitespace characters; on the contrary, their values can contain whitespace.

```
<cellar>
   <wine>
      <title>Vinsanto</title>
      <producer country="Greece">
         <Brandname>Santo Wines</Brandname>
      </producer>
      <vintage>2003</vintage>
      <!-- Vintage is the year the grapes were harvested. -->
      <purchase-date>July 2009</purchase-date>
   </wine>
</cellar>
```

Figure 2.1 XML document example

An example of an XML document is shown in Figure 2.1. This is a part of an XML document that carries information about a cellar. The hierarchy of "Cellar" and "wine" elements implies their "parent-child" relation. The "wine" element describes a bottle of wine in an individual's cellar and contains four elements: <title>, <producer>, <vintage>, <purchase-date> and one comment: <!-- Vintage is the year the grapes were harvested. -->. All elements have some text which is the information that is carried by the "wine" element. Furthermore, the <producer> element, except from its value, "SantoWines", contains an attribute named "country" that describes the origin of the wine producer.

2.1.2. Well-formed and valid documents

A **well-formed** XML document has correct XML syntax, which means that it follows the basic structural rules of XML:

- It contains one or more elements.
- There is exactly one element, the root or document element, no part of which appears in the content of any other element.
- The elements, delimited by start- and end-tags, nest properly within each other.
- The attribute values must be quoted.

Well-formed documents are well-formed because they do not have to be created in a structured environment, against a pre-defined set of structural rules, but merely have to comply with XML well-formedness constraints as presented above.

A **valid** XML document is a well-formed XML document, which also conforms to the rules of a Document Type Definition (DTD) or an XML Schema (XSD).

2.2. XML Schema 1.0

XML (Extensible Markup Language), described in section 2.1, allows the developers to create their own formats for storing and sharing information. XML Schema is the formal declaration and documentation of those formats, providing a foundation on which software developers can build software. XML Schema is the language for defining classes in XML documents. A set of attributes and content structure elements are defined for each class and its instances. An XML Schema language is a formalization of the constraints, expressed as rules or a model of structure, that apply to a class of XML documents. In many ways, schemas serve as design tools, establishing a framework on which implementations can be built.

The flexibility and the advanced structure-description capabilities of the XML Schema language [W3C/XSD1] have made it a de facto standard in metadata description. This is the case in several application domains like, for example, the multimedia domain, where both the MPEG-7 [ChSiPuMPEG7] and MPEG-21 [PerMPEG21] standards (used, respectively, for multimedia content and service description) have been expressed using XML Schema syntax.

XML schema [W3C/XSD1] is expressed in XML 1.0 syntax and is intended to describe the structure and express the content constraints of documents written in XML. Thus, XML Schema defines the structure of XML instance documents. XML Schema comprises a set of components in three main groups: primary, secondary and helper components.

2.2.1. Schema components

The **primary components** include Simple Type definitions, Complex Type definitions, Element definitions and Attribute definitions. Simple Types and Complex Types define the xml schema author-defined types that can be used across an xml schema by referring to them through the “type” attribute of elements. Complex Types may be defined as concrete or abstract, respectively allowing the derivation or not of instances, through the Boolean value of the optional attribute “abstract”. Simple Types are defined as restrictions, unions or lists of built-in types (like string, integer, double, Boolean etc.) or author-defined types of an xml schema. All kinds of types may or may not have a name, according to their location in the schema. Unnamed types are nested in other xml structures (elements, simple types etc.) and their scope is restricted within those structures. On the other hand, the element and attribute definitions must have a name.

Moreover, XML Schema supports mechanisms of inheritance both for types and elements, through the extension and restriction for the former and through the “substitutionGroup” for the latter. The types may include in their definition the “final” attribute with acceptable values “extension” and “restriction”, which prevents further derivations of a type by extension and restriction respectively. The SubstitutionGroup attribute supports the substitution of one and only one (this fact has changed in XML Schema 1.1 [W3C/XSD1.1a]) named element from another. Any top-level element declaration can serve as the defining member, or head, for an element substitution group. Other top-level element declarations, regardless of the target namespace, can be designated as members of the substitution group headed by this element. Note that the members must have

type definitions which are either the same as the head's type definition or restrictions or extensions of it.

The **secondary components** include attribute group definitions, identity constraint definitions, model group definitions and notation declarations. The attributes capture information about complex types and form attribute groups when there is need to use simultaneously all of them. Attribute groups must be declared in top-level, in contrary to individually declared attributes that may appear at top-level or nested in complex types. The same fact is applied to model groups, too. Model groups are collections of elements with a specific behavior among them. A model group specifies a sequential (sequence), disjunctive (choice) or conjunctive (all) interpretation of the group members. The elements taking part in sequences, choices and all structures, may carry the "minOccurs" or/and the "maxOccurs" attribute in their declaration. These attributes specify, respectively, the minimum and the maximum number of occurrences of each element in the model group. The default value, in case of absence, is 1, implying exactly one occurrence.

The **helper components** include annotations, model groups, particles (i.e. min occurs and max occurs), wildcards and attribute uses. These components essentially are small parts of other components, since they are not independent of their context.

2.2.2. Identity constraints

The identity constraint mechanism was introduced in [W3C/XSD1str] and is a very powerful tool for schema authors. XML Schema provides two ways to identify and reference elements and attributes inside an XML document. These are the ID/IDREF/IDREFS and key/keyref. The former combination is inherited from XML's DTDs while the latter is introduced by the XML Schema in order to offer more flexibility by using XPath. The usage semantics of the identity constraints are close to those of foreign keys in relational databases.

The first approach to describe identifiers and references with W3C XML Schema, ID/IDREF, can be used to define either attributes or elements by binding their type to the xs:ID, xs:IDREF or xs:IDREFS built-in datatypes. IDREFS is a whitespace-seperated list of IDREF values. IDs are global to a document, meaning that we are not allowed to use the same ID value for different constructs within the same document and indicates that the attribute value uniquely identifies the containing element. In Figure 2.2 and Figure 2.3 are presented an XML Schema that uses ID/IDREF/IDREFS and a sample XML document based on it, respectively.

```
<xs:element name="wine">
 <xs:complexType>
    <xs:sequence>
       <xs:element name="title" type="xs:string" minOccurs="1" maxOccurs="1"/>
       <xs:element name="producer" minOccurs="1" maxOccurs="1">
           <xs:complexType>
                <xs:attribute name="country" use="required" type="xs:string"/>
                <xs:attribute name="ref" type="xs:IDREF" use="required"/>
          </xs:complexType>
       </xs:element>
       <xs:element name="vintage" type="xs:IDREFS" minOccurs="1" maxOccurs="1"/>
       <xs:element name="purchase-date" type="xs:string" minOccurs="1" maxOccurs="1"/>
   </xs:sequence>
   <xs:attribute name="wineID" type="xs:ID" use="required"/>
 </xs:complexType>
</xs:element>
```

Figure 2.2 XML Schema element with ID/IDRef/IDRefs

This is a possible xml schema that describes a wine bottle in a cellar. Notice that the attributes values of the ID and IDREF types cannot start with a number. Also, notice that element “producer” cannot have any children, since the IDREF content type is empty. Moreover, the element “vintage” is a whitespace-separated list of years. Finally, the years start with the “Y” letter, because an IDREF value cannot start with a number. Under these limitations, a sample XML document based on the schema presented in Figure 2.2 is shown in Figure 2.3.

```
<wine wineID="IDWINE021">
          <title>Vinsanto</title>
          <producer country="Greece" ref="IDPROD0036"/>
          <vintage>Y2003 Y2005</vintage>
          <purchase-date>07-2009</purchase-date>
</wine>
```

Figure 2.3 XML document with ID/IDREF/IDREFS

In the second approach, XML Schema provides a more flexible feature for defining identity constraints without limitation on its lexical space (since the lexical space of the IDs, xs:NCName, cannot start with a number and whitespace is prohibited) and allowing local keys and references, as well as multinode keys. The constraints imposed by the unique, key and keyref elements have similar declaration and their augmentation to the ID/IDREF mechanism is described below.

The identity constraints of the key/keyref type contain: (a) A selector element, which specifies the XML Schema elements on which the identity constraint is applied; and (b) A field element, where the XML Schema constructs (elements or attributes) that form the constraint value are specified. Both the selector and field elements specify the construct(s) they refer to in their xpath attribute. The value of that attribute is a restricted XPath (XPath 1.0 [W3C/XSD1]) expression reffering to instances of the element being declared.

```
<xs:element name="wine">
  <xs:complexType>
        <xs:sequence>
          <xs:element name="title" type="xs:string" minOccurs="1" maxOccurs="1"/>
          <xs:element name="producer" minOccurs="1" maxOccurs="1">
                <xs:complexType>
                    <xs:all>
                      <xs:element name="Brandname" type="xs:string"/>
                    </xs:all>
                    <xs:attribute name="country" use="required" type="xs:string"/>
                </xs:complexType>
          </xs:element>
          <xs:element name="vintage" type="xs:integer" minOccurs="1" maxOccurs="1"/>
          <xs:element name="purchase-date" type="xs:string" minOccurs="1" maxOccurs="1"/>
        </xs:sequence>
  </xs:complexType>
  <xs:unique name="UniqueBottle">
        <xs:selector xpath="wine"/>
        <xs:field xpath="@title"/>
        <xs:field xpath="producer/Brandname"/>
        <xs:field xpath="vintage"/>
  </xs:unique>
</xs:element>
```

Figure 2.4 XML Schema element with Unique identity constraint

The enhancements of the ID/IDREF mechanism are the following: a)Functioning as a part of an identity-constraint is in addition to, not instead of, having a type; b)Not only attribute values, but also element content and combinations of values and content can be declared to be unique; c)Identity-constraints are specified to hold within the scope of particular elements; d)(Combinations of) attribute values and/or element content can be declared to be keys, that is, they are not only unique, but always present and non-nillable; and e) The comparison between keyref {fields} and key or unique {fields} is by value equality, not by string equality.

2.2.3. Namespaces

The Namespaces provide a URI-based mechanism that allows differentiating XML vocabularies. XML Schema associates a namespace to all the objects (elements and attributes, but also simple

and complex types as well as groups of elements and attributes) defined in a schema, allowing the use of namespaces to build modular libraries of schemas. A Namespace is a URI represented by a prefix. The default XML Schema namespace, which includes built in types is "http://www.w3.org/2001/XMLSchema" and is often represented by the "xs" prefix. Namespace prefixes should only be considered to be local shortcuts to replace the URI references that are the real identifiers for a namespace.

2.3. XML Schema 1.1

XML Schema 1.0 has evolved to XML Schema 1.1 by adding even more features and richer semantics in terms of syntax and structures. With XML Schema 1.0 we can express grammar and syntax constraints by defining the order of data or the type of elements. With XML Schema 1.1 we can still express all these constraints and in addition we can also express data relations.

2.3.1. Assertions

The **assertion** new feature of XML Schema allows expressing policies and rules. The assertion element consists of XPath 2.0 expression, which is evaluated over XML data and instances of classes. In case an XPath is not satisfied by the XML data, the XML Schema Validator marks the XML Document as invalid. An example of this scenario is shown below (Figure 2.5):

```
<element name="even-integer">
<simpleType>
   <restriction base="integer">
        <minInclusive value="0" />
        <maxInclusive value="100" />
        <assertion test="$value mod 2 = 0" />
   </restriction>
</simpleType>
</element name="even-integer">
```

XML Schema Simple Type with assertion

```
<even-integer> 4 </even-integer>
<even-integer> 3 </even-integer>
```

XML Data

The first row is valid according to the XML Schema. In contrast to the first row, the second row is not valid, because 3 mod 2 != 0.

Figure 2.5 Assertion in Simple Type definition

As shown in Figure 2.5, an assertion is part of a Simple Type definition restriction. An assertion can also appear out of Simple Type definitions with the tag name **assert.** It has exactly the same functionality and usage. The XML Schema 1.1 <assert> element is used to make assertions about the values of elements and attributes. In the example below (Figure 2.6), assert is part of a Complex Type:

```
<xs:element name="intRange">
  <xs:complexType>
    <xs:attribute name="min" type="xs:int"/>
    <xs:attribute name="max" type="xs:int"/>
    <xs:assert test="@max le 6"/>
  </xs:complexType>
</xs:element>
```

XML Schema Complex Type with assert restriction

```
<intRange min=″3″ max =″12″/>
<intRange min=″0″ max =″2″/>
```

XML Data

The first row is invalid according to the XML Schema. In contrast to the first row, the second row is valid, because max < 6.

Figure 2.6 Assert in Complex Type definition

2.3.2. Alternative

XML Schema 1.1 allows expressing conditional content using **alternatives**. The instances of the Alternative type provide associations between boolean conditions (as XPath 2.0 expressions) and Type Definitions. They are used in conditional type assignment. This feature is very useful and makes XML Schema more expressive. The XPath expression can only reference attributes of the current element. It can reference neither ancestor elements nor descendent elements.

The example shown below (Figure 2.7) presents a sample of an XML Schema that uses the alternative construct and the correspondent XML data. In particular, the “Person” element has the “PersonType” complex type. In addition, the alternative element assigns a different complex type to the “Person” element according to the value of the “kind” attribute, where in case the “kind” has the “employee” value then the type of the “Person” element is set to the “EmployeeType” complex type. In the second part of the figure is presented a sample of XML data that follows the already described schema. Thus, the “Person” element, which has the “employee” value at the “kind” attribute shall contain content that is specified by the “EmployeeType” complex type.

```
<xs:complexType name="PersonType">
    <xs:group ref="personGroup"/>
    <xs:attribute name="kind" type="xs:string" />
</xs:complexType>
<xs:element name="Person" type="PersonType">
    <xs:alternative  test="@kind eq 'employee' "  type="EmployeeType" />
</xs:element>
```

XML Schema Complex Type PersonType and element of PersonType

```
<Person kind=”employee”>
   “content of EmployeeType instead of PersonType”
</Person>
<Person kind=”employer”>
   “content of PersonType”
</Person>
```

XML Data

Figure 2.7 Example of the usage of Alternative

2.3.3. Override / Redefine

The **redefine** construct defined in XML Schema 1.0 is useful in schema evolution and versioning, when it is desirable to have some guaranteed restriction or extension relation between the old component and the redefined component. But there are occasions when the schema author simply wants to replace old components with new ones without any constraint. In addition, the redefine construct is deprecated. The **override** construct is defined in XML Schema 1.1 in order to replace redefine. The content of override includes one or more occurrences of simpleType, complexType, group, attributeGroup, element, attribute or notation. Besides the content structure, the override definition requires an attribute, called schemaLocation, which declares the IRI[(1)] of a referenced schema as shown in the following example (Figure 2.8). Override is defined in an XML Schema document other than the one targeted by the override element.

In particular, the sample of XML Schema shown in Figure 2.8 specifies that the “validAgeType” simple type of the XML Schema described in the “persons.xsd” file will be replaced by the definition that is contained within the override element for the current XML Schema. Thus, this schema reuses the corporate schema “persons.xsd”, but customizes the "validAgeType" with the definition that is particular to the current XML Schema.

[1] *IRI is an Internationalized Resource Identifier Reference. See Figure 2.40 for URI, IRI definitions.*

```
<xs:override schemaLocation="persons.xsd">
   <xs:simpleType name="validAgeType" >
      <xs:annotation>
            <xs:documentation>
                  Overrided Simple Type.
            </xs:documentation>
      </xs:annotation>
      <xs:restriction base="xs:integer">
            <xs:minInclusive value="23"/>
            <xs:maxInclusive value="55"/>
      </xs:restriction>
   </xs:simpleType>
</xs:override>
```

Figure 2.8 Example of the usage of override

2.3.4. Error datatype

XML Schema 1.1 introduces a new built-in datatype, **xs:error**. The datatype xs:error has no valid instances, it has an empty value space and an empty lexical space. It can be used in any place where other types are normally used. The type xs:error is expected to be used mostly in conditional type assignment. Whenever it serves as the governing type definition for an attribute or element information item, that item will be invalid. Examples (Figure 2.9), (Figure 2.10) show exactly the way it works.

In Figure 2.9 is presented an extended version of example shown in Figure 2.7, which described the semantics of the alternative XML construct. In addition to the previous example, this one defines one more alternative element within the “Person” element, which sets the error datatype as the governing type of the “Person” element, if the value of the “kind” attribute is not equal to the “employee” value or to the “employer” value.

```
<xs:complexType name="PersonType">
    <xs:group ref="personGroup"/>
    <xs:attribute name="kind" type="xs:string" />
</xs:complexType>
<xs:element name="Person" type="PersonType">
    <xs:alternative  test="@kind eq 'employee' "  type="EmployeeType" />
    <xs:alternative  test="(@kind ne 'employee') or (@kind ne 'employer')"
                                                        type="xs:error" />
</xs:element>
```

Figure 2.9 Error in an Alternative element

In Figure 2.10 is presented the “Country” element of error datatype. The semantics of this declaration specify that if there exists an individual of this element among the XML data, then the

XML document will be invlalid. This could be useful in a use case, where an XML Schema already exists and is being extended by another XML Schema, in which the “Country” element is overrided and assigned the error datatype in order to prevent from using it in the XML documents.

```
<xs:element name="Country" type="xs:error" />
```

Figure 2.10 Error in an XML Schema element

2.3.5. SubstitutionGroup

Through the **substitution group** mechanism, XML Schema 1.0 provides a model supporting substitution of one named element for another. Any top-level element declaration can serve as the defining member, or head, for an element substitution group. Other top-level element declarations, regardless of the target namespace, can be designated as members of the substitution group headed by this element.

XML Schema 1.0 only allowed an element to be substitutable by another element. XML Schema 1.1 permits an element to be substitutable by multiple elements. To be precise, it allows for replacing an element with another. XML Schema 1.1 enhances this functionality by allowing an element to be substitutable for one or more elements. Note that element substitution groups are not represented as separate components. They are specified in the property values of the element declarations.

The example shown in Figure 2.11 demonstrates the functionality of the substitution group construct. In particular, in the context of the example are declared the “metrorail” element that represents any electric passenger railway in an urban area, the “subway” element that represents an underground railway and the “metro” element. The “metro” element is substitutable for the “metrorail” element as well as for the “subway” element according to the semantics of the substitution group. That means that the “metro” element can be replaced by the “metrorail” or the “subway” element within an XML document.

```
<xs:element name="metrorail" type="xs:string" />
<xs:element name="subway" type="xs:string" />

<xs:element name="metro" substitutionGroup="metrorail  subway" type="xs:string" />
```

Figure 2.11 XML Schema Substitution Group

The datatype of the “metro” element must derive from the datatype of the other two elements, which are the “metrorail” and the “subway” elements. If `metrorail`'s datatype is xs:string and `metro`'s datatype was xs:integer, that would be an invalid XML Schema.

2.3.6. Identity constraints

The identity constraints unique, key, keyref have the same declaration and behavior, with the one they had in XML Schema 1.0 recommendation, within an XML schema. The only difference in syntax is an additional optional attribute, which specifies if the declaration refers to an existing identity constraint of the same constraint type and where it is placed in the schema. In case of usage, in the identity constraint declaration only id and annotation are allowed to appear together with ref. Moreover, the xpath attribute of the field and selector elements follows the XPATH 2.0 recommendation ([W3C/XPATH2]) in contrast to the previous XML schema version, 1.0, that followed the XPATH 1.0 recommendation ([W3C/XPATH]).

2.3.7. XML Schema Motivation Example

An XML Schema is presented in Figure 2.12, which describes a sequence of persons, their personal information and the relations existing between them.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified">
        <xs:element name="Persons" type="PersonsType" >
                <xs:unique name="knowsUnique">
                        <xs:selector xpath="Person"/>
                        <xs:field xpath="Name"/>
                        <xs:field xpath="Address/@city"/>
                </xs:unique>

                <xs:key name="personID">
                        <xs:selector xpath="Person"/>
                        <xs:field xpath="ID"/>
                </xs:key>

                <xs:keyref name="IDRef" refer="ID">
                        <xs:selector xpath="Person"/>
                        <xs:field xpath="Knows"/>
                </xs:keyref>
        </xs:element>

        <xs:complexType name="PersonsType">
                <xs:sequence>
                        <xs:element ref="Person" maxOccurs="unbounded" />
```

```
            </xs:sequence>
        </xs:complexType>

        <xs:element name="Person" type="PersonType"/>
        <xs:element name="Employee" type="EmployeeType"/>
        <xs:element name="staff" substitutionGroup="Person Employee"/>
        <xs:complexType name="PersonType" >
            <xs:group ref="personGroup"/>
        </xs:complexType>
        <xs:complexType name="EmployeeType">
            <xs:complexContent>
                <xs:extension base="PersonType">
                    <xs:sequence>
                        <xs:element name="Employer" type="xs:string"
                                                    maxOccurs="unbounded"/>
                    </xs:sequence>
                </xs:extension>
            </xs:complexContent>
        </xs:complexType>
        <xs:group name="personGroup">
            <xs:sequence>
                <xs:element name="ID" type="IDType"/>
                <xs:element name="Name" type="NameType" />
                <xs:element name="Address" type="AddressType"
maxOccurs="unbounded"/>
                <xs:element name="Age" type="validAgeType"/>
                <xs:element name="Knows" type="IDType"/>
            </xs:sequence>
        </xs:group>

        <xs:complexType name="AddressType">
            <xs:sequence>
                <xs:element name="Street" type="xs:string"/>
                <xs:element name="Number" type="xs:nonNegativeInteger"/>
            </xs:sequence>
            <xs:attributeGroup ref="addressGroup"/>
        </xs:complexType>
        <xs:attributeGroup name="addressGroup">
            <xs:attribute ref="country" use="required" />
            <xs:attribute name="city" type="xs:string"/>
            <xs:attribute name="postCode" type="xs:integer"/>
        </xs:attributeGroup>
        <xs:attribute name="country">
            <xs:simpleType>
                <xs:restriction base="xs:string">
```

```
                        <xs:whiteSpace value="collapse"/>
                        <xs:pattern value="[a-zA-Z]{2}"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:attribute>
            <xs:complexType name="NameType">
                <xs:all>
                    <xs:element name="FirstName" type="xs:string"/>
                    <xs:element name="Surname" type="xs:string"/>
                </xs:all>
                <xs:attribute name="prefix" type="xs:string"/>
            </xs:complexType>
            <xs:complexType name="IDType">
                <xs:choice>
                    <xs:element name="PassportID" type="xs:nonNegativeInteger"/>
                    <xs:element name="IdentityCardID" type="xs:string"/>
                </xs:choice>
            </xs:complexType>
            <xs:simpleType name="validAgeType" >
                <xs:annotation>
                    <xs:documentation>
                        SimpleType needs an annotation too.
                    </xs:documentation>
                </xs:annotation>
                <xs:restriction base="xs:float">
                    <xs:minInclusive value="0.0"/>
                    <xs:maxInclusive value="150.0"/>
                </xs:restriction>
            </xs:simpleType>
</xs:schema>
```

Figure 2.12 XML Schema describing Persons

This XML Schema is used through our examples in the following sections. It contains a root schema element which has two top level descendants “Persons” and ”Person” belonging, respectively to the Complex Types “PersonsType” and “PersonType”. “PersonType” is made up of a group reference, “personGroup”, which contains a sequence group model named “personGroup”. This sequence has five elements named “ID”, “Knows”, “Name”, “Address”, “Age” of Complex Types “IDType”, “IDType”, “NameType”, “AddressType” and Simple Type “validAgeType” respectively. Besides “personGroup”, our schema contains an attribute group named “addressGroup”, which holds information about the country, the city and the post code of an address. This attribute group is referenced by the Complex Type “AddressType”, that has been previously described.

In the following figure (Figure 2.13) is presented an XML document (PersonsMUSIC.xml), which contains the data that is used in our query examples through this section and conforms to the XML Schema described in Figure 2.12.

```
<Persons>
   <Person>
      <ID><IdentityCardID>A123456</IdentityCardID></ID>
      <Knows><PassportID>1234612</PassportID></Knows>
      <Name>
         <FirstName>Ioannis</FirstName>
         <Surname>Stavrakantonakis</Surname>
      </Name>
      <Address country="GR">
         <Street>Hrwwn Polutexneiou</Street>
         <Number>21</Number>
      </Address>
      <Age>24</Age>
   </Person>
   <Person>
      <ID><PassportID>1234612</PassportID></ID>
      <Knows><IdentityCardID>A123456</IdentityCardID></Knows>
      <Knows><IdentityCardID>E347873</IdentityCardID></Knows>
      <Name>
         <FirstName>Chrisa</FirstName>
         <Surname>Tsinaraki</Surname>
      </Name>
      <Address country="GR">
         <Street>M.Mpotsari</Street>
         <Number>12</Number>
      </Address>
      <Address country="IT">
         <Street>Trento St.</Street>
         <Number>20</Number>
      </Address>
   </Person>
   <Employee>
         <ID><IdentityCardID>E347873</IdentityCardID></ID>
         <Name>
            <FirstName>Nektarios</FirstName>
            <Surname>Gioldasis</Surname>
         </Name>
         <Address country="GR">
            <Street>Kounoupidianwn</Street>
            <Number>28</Number>
         </Address>
         <salary>5000</salary>
   </Employee>
</Persons>
```

Figure 2.13 The XML document PersonsMUSIC.xml with data about members of MUSIC/TUC laboratory

The XML Schema of Figure 2.12 does not cover every aspect and pattern that XS2OWL can serve. Because of this, during the further analysis and description of the transformation model may be used some additional XML Schema constructs, as an extension of this schema, not mentioned in Figure 2.12. In my case, the majority of the our examples is based on the schema of Figure 2.12.

2.4. XPath

XPath (XML Path Language) [W3C/XPATH] [W3C/XPATH2] [W3C/XQXPfn] is a language used for totally data in XML documents by parsing those XML documents for specific values. XPath performs parsing of XML documents by applying an expression to the text of an XML document. The effective result is that an XPath expression allows navigation through XML document elements and attributes, retrieving items and values that match the expression passed into the XML document by XPath. This effectiveness made XPath a very useful tool in the W3C XML technology toolbox. It is used in XSLT [W3C/XSLT], XPointer [W3C/Xpointer] and XQuery [W3C/XQuery] when there is a need of path navigation through a tree of nodes.

In its simplest form, an XPath expression forms an absolute or a relative path. An absolute path has a starting forward slash (/), and a relative path does not. An absolute path searches an XML document from the precise path specified. A relative path searches relatively from wherever specified. Figure 2.15 shows an absolute path of the XML document presented in Figure 2.13 and

```
/Persons/Person/Name
```

Figure 2.15 XPath absolute path example

Figure 2.14 describes a relative path from the element that the first example points to. Note that <Persons> node is the root node of the XML document.

```
Name/FirstName
```

Figure 2.14 XPath relative path example

XPath uses a path expression to pull nodes from an XML document. A path expression is an expression, expressed as a path, through a hierarchical structure. The path is a pattern that is matched to the structure of an XML document. It finds all the paths that match to the expression. The most basic XPath expressions are shown in Table 2.1.

Expression	Matching results
Node	All child nodes
/	Search from the root node down

//	Search from specified node down
.	Current node (context)
..	Parent node
@	Attributes

Table 2.1 Basic XPath expressions

Beyond the functionality that has been discussed till now, XPath allows filtering the result tree of a path expression, in the same way the where clause is used in SQL, by using the predicate-mechanism. Thus, a predicate can be used to qualify which nodes are retrieved based on the values contained within those nodes. A predicate is applied to a specific node within an XPath expression, and is enclosed in square brackets [ ] as shown in Figure 2.16 and Figure 2.17. In the former is presented an arithmetic predicate consisted by a number, which specifies the retrieved

```
/Persons/Person[2]/Name
```

Figure 2.16 XPath expression with arithmetic predicate

node among the children of the parent element according to its position. In the latter is presented a Boolean predicate that retrieves only the nodes for which the expression in the square brackets is true.

The XPath expression of Figure 2.16 retrieves the Name node of the second person element under the Persons node, while the XPath expression of Figure 2.17 retrieves the Name node of the

```
/Persons/Person/Name[./FirstName="Ioannis"]
```

Figure 2.17 XPath expression with boolean predicate

person element the FirstName element of which equals the string "Ioannis".

In Figure 2.17, the operator of equality (=) is used in the expression of the predicate. There is a broad set of operators [W3C/XQXPfn] available to use in such an expression. Some of them are presented in Table 2.2.

Operator	Operation
|	Finds two nodes
+	Addition
-	Subtraction
*	Multiplication
=	Equal to
<=	Less than or equal to
>=	Greater than or equal to
!=	Not equal to

>	Greater than
<	Less than
Or	Logical OR
And	Logical AND
Mod	Modulus

Table 2.2 XPath operators

Finally, XPath expressions use wildcards. XPath wildcards can be used to select unknown XML elements. Table 2.3 describes the meaning of the available wildcards.

Wildcard	Matching results
*	Matches any element node
@*	Matches any attribute node
node()	Matches any node of any kind

Table 2.3 XPath wildcards

2.5. XQuery

XQuery is a query language mainly intended for use with XML data and XML databases, but it is also used as a language for data integration, even with data sources that may not be XML but can be considered as XML structured. XQuery is a standardized language for combining documents, databases, Web pages and almost type of data.

Talking about the query language of databases expressed in XML, one would recall SQL, the query language of relational databases. XQuery and SQL are both query languages for databases, so it is natured to have some similarities in their semantics. The biggest difference that can be figure out is that SQL results in sets of rows (tuples) without any hierarchy imposed, in contrast to XQuery which focuses on ordered sequences of values and hierarchic of nodes.

2.5.1. Language Features

XQuery is a functional language and every query derived from the set of its rules is an expression. There are several types of expressions including primary expressions, path expressions, sequence expressions, arithmetic expressions, comparison expressions, logical expressions, constructors, FLWOR expressions, conditional expressions, quantified expressions, expressions on sequence types and validation expressions.

Primary expressions are the basic primitives of the language. They include literals, variables, function calls, and parenthesized expressions.

Sequence expressions: XQuery supports operators to construct and combine sequences. A sequence is an ordered collection of zero or more items. An item is either an atomic value or a node.

Arithmetic expressions: XQuery provides arithmetic operators for addition, subtraction, multiplication, division, and modulus, in their usual binary and unary forms.

The comparison expressions allow two values to be compared. XQuery provides three kinds of comparison expressions, called value comparisons, general comparisons, and node comparisons.

Logical expressions: A logical expression is either an and-expression or an or-expression. The value of a logical expression is always one of the boolean values: true or false.

Conditional expressions: A conditional expression supports conditional evaluation of one of two expressions.

Quantified expressions: XQuery defines two quantification expressions, “some” and “every”.

Expressions on SequenceTypes: Some of the expressions relying on the SequenceTypes syntax are called expressions on SequenceTypes.

Validate expressions: A validate expression validates its argument with respect to the in-scope schema definitions. The argument of a validate expression must be either an element or a document node. Validation replaces all the nodes with new nodes that have their own identity and default values created during the validation process.

There are three more types of expressions to complete the list of the XQuery expression types that can be used to formulate a query. These types are path, construct and FLWOR expressions being described in subsections 2.4.2, 2.4.3 and 2.4.4 respectively.

2.5.2. XQuery uses XPath

XQuery is used to query XML documents, as it was previously described, which all have a tree structure of nodes. XPath is the best way to locate and retrieve information from nodes within a tree. Thus, XQuery uses **path expressions**, which are based on XPath. There are two kinds of path expressions, absolute path expressions and relative path expressions. An absolute path expression is a rooted relative path expression. A relative path expression is composed of a sequence of steps.

In Figure 2.18 is presented an XQuery path expression. The query processor parses the PersonsMUSIC.xml document, which is imposed by the doc() function, and returns the document’s

```
doc("PersonsMUSIC.xml")//Person[./Name/FirstName="Ioannis"]
```

Figure 2.18 XQuery path expression

root node. Double slash, //, is used to select all the person nodes in the document, regardless of where the nodes are located. Finally, the set of person elements is restricted, by using the predicate in square brackets [ ], only for those person elements that have a particular FirstName value.

It is worthwhile to mention that the choice to leave outside the braces only the element Person implies that we need person elements to our result tree. In case the query was:

```
doc("PersonsMUSIC.xml")//Person/Name/FirstName[.="Ioannis"]
```

The result would be a tree comprised of FirstName nodes.

```
<Person>
      <ID><IdentityCardID>A123456</IdentityCardID></ID>
      <Knows><PassportID>1234612</PassportID></Knows>
      <Name>
         <FirstName>Ioannis</FirstName>
         <Surname>Stavrakantonakis</Surname>
      </Name>
      <Address country="GR">
         <Street>Hrwwn Polutexneiou</Street>
         <Number>21</Number>
      </Address>
      <Age>24</Age>
   </Person>
```

Figure 2.19 XQuery path expression result tree

2.5.3. XQuery constructs nodes

XQuery includes expressions that can create new nodes by using constructors. XQuery supports two forms of constructors, direct and computed. Direct constructors support literal XML syntax for elements, attributes, text nodes, processing-instructions and comments. Computed constructors can be used to construct elements, attributes, text nodes, processing-instructions, comments, and document nodes. All the direct constructors are normalized into computed constructors, i.e., there are no direct-constructor expressions in the Core of XQuery.

In Figure 2.20 is shown the creation of two elements using two direct constructors.

```
<FirstName>{doc("PersonsMUSIC.xml")//Person[1]/Name/FirstName}</FirstName>
<Surname>Stavrakantonakis</Surname>
```

Figure 2.20 XQuery with direct constructors

The first direct constructor contains one enclosed expression and the second contains one contiguous sequence of characters. The expression enclosed in curly brackets, as shown in Figure 2.20, is evaluated and the result replaces the curly braces and the content of them. Thus the processing of the constructors, shown in Figure 2.20, produces the elements of Figure 2.21.

```
<FirstName>Ioannis</FirstName>
<Surname>Stavrakantonakis</Surname>
```

Figure 2.21 XML result tree created from a direct constructors

The computed constructors provide an alternative syntax for expressing a constructor as shown in Figure 2.22. In case the name and the content must be computed, the usage of a computed constructor is the only choice.

```
element Name { doc("PersonsMUSIC.xml")//Person[1]/Name/FirstName,
doc("PersonsMUSIC.xml")//Person[1]/Name/Surname }
```

Figure 2.22 XQuery with computed constructor

The query of Figure 2.22, results in the tree shown below (Figure 2.23):

```
<Name>
        <FirstName>Ioannis</FirstName>
        <Surname>Stavrakantonakis</Surname>
</Name>
```

Figure 2.23 XML result tree created for computed constructor

2.5.4. XQuery functions

XQuery defines a broad number of built-in functions [W3C/XQXPfn]. These functions are also used by XPath. They range in functionality from the simple *concat()* used in Figure 2.29 to complex regular expression handlers like *matches()*. Every built-in function resides in the namespace http://www.w3.org/2003/11/xpath-functions, which is bound to the predefined namespace prefix *fn*. Except from the built-in functions, XQuery allows the declaration of user-defined functions.

User-defined functions are declared using the declare function expression. This expression takes the function signature, followed by an expression enclosed in curly braces ({}) that defines the body of the function, and ends with a semicolon. If the declared function name doesn't use a prefix, then it doesn't have a namespace (not even the default function namespace).

Some built-in functions are overloaded, meaning that there are several functions with the same name but with different arguments and/or return types. On the contrary, the user-defined functions are never overloaded, and always take a fixed number of arguments.

An example of a user-defined function is presented in Figure 2.24. This function computes the absolute value of an integer passed to it. It takes one integer argument ($i), and returns an integer that is the absolute value of the argument.

```
declare function abs($i as xs:integer) as xs:integer {
  if ($i < 0) then -$i else $i
};
```

Figure 2.24 XQuery user-defined function

2.5.5. XQuery flower expressions

The FLWOR (pronounced "flower") expression is the most powerful type among the expression set of XQuery. It is capable of introducing variables, filtering sequences, sorting and grouping, iterating over sequences to produce some result, and joining different data sources, all at once. FLWOR stands for the syntactic units of a FLWOR expression, FOR – LET – WHERE – ORDER BY –

RETURN. These clauses are presented in the subsections 2.5.5.1 and 2.5.5.2. Every FLWOR expression begins with one or more *for* and/or *let* clauses (in any order), followed by an optional *where* clause, an optional *order by* clause, and finally one *return* clause, which is mandatory. XQuery is case sensitive both for reserved keywords and user defined constructs.

The functionality of the clauses of the FLWOR expressions is the following:

- The **"for"** clause binds one or more variables to a collection of elements and sets up an iteration through them from one element to another.
- The "**let"** clause allows creating a variable and assigning a value to it. "Let" does not result in iteration. It can improve performance, because the expression is evaluated only once instead and not each time it is needed.
- The "**where"** clause allows filtering of the items that should be used in the collection.
- The "**order by"** clause sorts the resulting collection of items sent back by the return clause.
- The "**return**" clause specifies specific data items that should be returned from a FLWOR expression.

2.5.5.1. For vs. Let clause

The "for/let" clauses determine the collection of elements that the *where*, *order by* and *return* clauses are applied on. Every FLWOR expression has at least one for or let clause. The previous section described in brief the difference between for and let. Through the examples shown below is shown how the results of a FLWOR expression are affected by our decision on which of two clauses we use.

In Figure 2.25 there is a query over the xml data of Figure 2.13. This query is based on a clause and creates an element named "GreekPerson" for each person that has an address in Greece.

```
for $x in doc("PersonsMUSIC.xml")/Persons/Person
where $x/Address/@country='GR'
return (<GreekPerson>{$x/Name/Surname}</GreekPerson>)
```

Figure 2.25 FLWOR expression with For clause

In this example the query processor binds the $x variable to the collection of Person elements that appear in the document PersonsMUSIC.xml. This collection is restricted by the where clause, which eliminates from the Person element collection, those that do not have an Address element with the "country" attribute equal to "GR". Finally, the return clause creates an element named

"GreekPerson" for each element of the restricted collection after the where clause applied to the initial collection. The result of the query described above is shown in Figure 2.26.

```
<GreekPerson><Surname>Stavrakantonakis</Surname></GreekPerson>
<GreekPerson><Surname>Tsinaraki</Surname></GreekPerson>
```

Figure 2.26 Result of XQuery with For clause

The same query can be expressed by using the let clause as it is demonstrated in Figure 2.27 and Figure 2.28.

```
for $x in doc("PersonsMUSIC.xml")/Persons/Person
where $x/Address/@country='GR'
return (<GreekPerson>{$x/Name/Surname}</GreekPerson>)
```

Figure 2.27 FLWOR expression with Let clause

The result of the query expressed in Figure 2.27 is shown in figure 2.28:

```
<GreekPersons>
        <Surname>Stavrakantonakis</Surname>
        <Surname>Tsinaraki</Surname>
</GreekPersons>
```

Figure 2.28 Result of XQuery with Let clause

Comparing Figure 2.26 with Figure 2.28, the result trees of the *for* and *let* clauses respectively, one can easily clarify how both of them work on a collection of elements. The *for* clause forces an iteration for every single element of the collection at the return statement and creates a "GreekPerson" element for each iteration. On the other hand, the let clause binds a variable to the entire collection and avoids iterations. Thus, the result tree is different as it contains only one "GreekPersons" element, which contains the entire collection bound to the variable.

2.5.5.2. Order by clause

The order by clause is applied just before the return clause to arrange in ascending or descending order the elements of the collection that form the result tree. The specified element/attribute of the order by clause may be different from the one of the return clause. In Figure 2.29 is presented a query that discovers some persons of the xml data repository, according to the where clause,

orders them based on the FirstName element and returns a collection of nodes that contain the concatenation of FirstName and Surname.

```
for $x in doc("PersonsMUSIC.xml")/Persons/Person
where $x/Address/@country='GR'
order by $x/Name/FirstName ascending
return (<GreekPerson>{concat($x/Name/FirstName,' ',$x/Name/Surname)}</GreekPerson>)
```

Figure 2.29 FLWOR expression with order by clause

The result tree of the query of Figure 2.29 shown above is shown in Figure 2.30.

```
<GreekPerson>Chrisa Tsinaraki</GreekPerson>
<GreekPerson>Ioannis Stavrakantonakis</GreekPerson>
```

Figure 2.30 Result of XQuery with Order By clause

2.6. RDF/S

RDF (Resource Description Framework) [W3C/RDF] is actually one of the older specifications, with the first working draft produced in 1997 [W3C/RDF97]. In the earliest version, authors established a mechanism for working with metadata that promotes the interchange of data between automated processes. This mechanism became the base on which RDF was developed. Regardless of the transformations RDF has undergone and its continuing maturing process, this statement forms its immutable purpose and focal point.

The Resource Description Framework (RDF) is a language designed to support the Semantic Web, in much the same way that HTML is the language that helped initiate the original Web. RDF is a framework for supporting resource description, or metadata (data about data), for the Web. RDF provides common structures that can be used for interoperable XML data exchange.

One of the differences between XML and RDF is about the tree-structured nature of XML, as compared to the much flatter triple-based pattern of RDF . XML is hierarchical, which means that all related elements must be nested within the elements they are related to. RDF does not require this nested structure. On the other hand, XML does not provide any information about the data described. That is, the nesting structure of the nodes does not imply in any way the relations among the data described, but only which element is the parent of the other element. In contrast to this fact, an RDF triple pattern gives the information how a datum is related to the rest of the data of the domain.

2.6.1. Basic features of RDF

RDF conceptualizes anything (and everything) in the universe as a **resource**. A resource is simply anything that can be identified by a Universal Resource Identifier (URI). URI provides a unique identifier for the information. Anything whether we can retrieve it electronically or not, can be uniquely identified in a similar way.

An **RDF triple** is formed by three components (predicate, subject, and object). These components create a **statement**, subject – predicate – object, in which the predicate specifies the relation between the subject and the object. The subject and the object may be any resources or literals (literals are atomic values, strings). Moreover, predicates are also known as **properties**. RDF properties may be thought of as attributes of resources and in this sense they correspond to traditional attribute-value pairs. RDF properties represent relations between resources. Let P be a predicate and x, y the subject and the object respectively (x, P, y).The property P can be regarded as a logical function: P(x,y) of two inputs, which describes the relation between x and y.

Furthermore, RDF triples can be described by an RDF graph, a directed labeled graph that contains nodes and arcs. The RDF triple is comprised of a resource node (the subject) which is linked to another resource node (the object) through an arc labeled with a third resource (the predicate). In Figure 2.31 is shown an RDF graph that contains one triple.

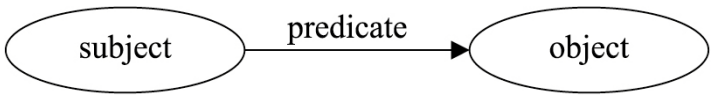

Figure 2.31 The RDF graph of a RDF triple

Unfortunately, recording the RDF data in a graph is not the most efficient means of storing or retrieving this data. Instead, encoding RDF data, a process known as serialization, is usually preferend, resulting in **RDF/XML** which is based on XML structures. An RDF/XML document starts with the root element rdf:RDF. The children of this element include the data descriptions (rdf:Description). A description element expresses a statement about one resource. The reference to that resource can be established by a) Using the rdf:id, in case the resource is new; b) Using the rdf:about, in case to refer to an existing resource; or c) Without using any name in order to be anonymous. Anonymous resources belong to a special category of nodes, the blank nodes, defined in Definition 2.1.

Definition 2.1. Blank nodes are nodes that don't have a URI. When identifying a resource is meaningful, or the resource is identified within the specific graph, a URI is given for that resource. However, when the identification of the resource does not have sense or does not exist within the specific graph at the time the graph was recorded, the resource is treated as a blank node. In either case, not having a URI for the item does not mean we cannot talk about it and refer to it.

Blank nodes are graph nodes that represent a subject (or object) for which we would like to make assertions, but have no way to address with a proper URI.

The following example demonstrates the usage of blank nodes within an RDF graph:

Example 2.1. The following RDF graph describes that "*John has a friend born the 26th of January"*. This expression can be written with two triples linked by a blank node representing the anonymous friend of John.

ex:John foaf:knows _:p1
_:p1 foaf:birthDate 01-26

The first triple specifies that "John knows p1". The second triple specifies that "p1 is born on January 26th". Moreover, "ex:John" is a named resource, which means this resource is absolutely identified by the URI obtained by replacing the "ex:" prefix by the XML namespace it stands for, such as http://music.tuc.gr/Person#John. The "_:p1" blank node represents John's anonymous friend, not identified by a URI. One can know by the semantics declared in the FOAF vocabulary [FOAF] that the class of "_:p1" is "foaf:Person".

2.6.2. RDFS - RDF Schema

RDF, provides no mechanisms for describing properties, nor does it provide any mechanism for describing the relations between these properties and other resources. In other words, does not make assumptions about any particular application domain and does not specify the semantics of any domain. That is the role of the RDF vocabulary description language, RDF Schema [W3C/RDFS], which is a semantic extension of RDF. RDF Schema defines classes and properties that may be used to describe classes, properties and other resources. These classes and properties specify the semantics of RDF statements in a particular knowledge field. The classes represent the entities of a field and the rules that apply to them, while the properties specify the relations among those entities by defining a domain and a range for each property. The domain and the range of a property may be regarded as restrictions on the property values. Individual objects belonging to a class are called instances. However, RDFS has some limitations like:

- There is nothing in the schema that restricts the cardinality of a specific property, that is, how many resources can be related to a property.
- There is no information that two properties are disjoint
- We cannot use the semantics of union, intersection and complement.
- The range of a property is applied to the whole set of classes. Thus, we cannot define a restriction applied only on a subset of the classes.
- We cannot define properties to be unique, transitive or inverse of another property.

The Semantic Web will build on XML the ability of to define customized tagging schemes and on the flexible data representetion approach. The next element required for the Semantic Web is a Web ontology language which can formally describe the semantics of classes and properties used in web documents and solve all the problems that were described above. In order for the

machines to perform useful reasoning tasks on these documents, the language must go beyond the basic semantics of RDF Schema. If we could draw analogies from other existing data schemes, if RDF and the relational data model were comparable, then RDF/XML is also comparable with the existing relational databases, and OWL would be comparable with the business domain applications.

2.7. OWL 1.0

The OWL Web Ontology Language [W3C/OWL] is a language for defining and instantiating Web ontologies. Ontology is a term borrowed from philosophy[1] that refers to the science of describing the kinds of entities in the world and how they are related. An OWL ontology may include descriptions of classes, properties and their instances. Someone would wonder which need lead to one more language for expressing data, since the XML and XML Schema are the de facto standards in data description. An ontology differs from an XML schema in that it is a knowledge representation, not a message format. The rest of the W3C specifications are not designed to support reasoning. Thus, an advantage of the OWL ontologies will be the availability of tools that can reason about them. Someone would want to develop an ontology for the following resons: a) to share common understanding of the structure of information among people or software agents, b) to allow the reuse of domain knowledge, c) to make domain assumptions explicit, d) to separate domain knowledge from the operational knowledge, and e) to analyze domain knowledge.

The W3C OWL Web Ontology Language (OWL) is a Semantic Web language designed to represent rich and complex knowledge about things, groups of things, and relations between things. It is a computational logic-based language such that knowledge expressed in OWL can be reasoned with by computer programs either to verify the consistency of that knowledge or to make implicit knowledge explicit. The OWL documents, known as ontologies, can be published in the World Wide Web and may refer to or be referred from other OWL ontologies.

The OWL language provides three increasingly expressive sublanguages designed for use by specific communities of implementers and users.

- **OWL Lite** supports those users primarily needing a classification hierarchy and simple constraint features. For example, while OWL Lite supports cardinality constraints, it only allows cardinality values of 0 or 1. Thus, it is should be simpler to provide tool support for OWL Lite than its more expressive relatives, and provide a quick migration path for thesauri and other taxonomies.

- **OWL DL** supports those users who want the maximum expressiveness without losing computational completeness (all entailments are guaranteed to be computed) and

[1] Parmenides was among the first to propose an ontological characterization of the fundamental nature of reality.

decidability (all computations will finish in finite time) of the reasoning systems. OWL DL includes all the OWL language constructs with restrictions such as type separation (a class cannot also be an individual or property, a property cannot also be an individual or class). OWL DL is so named due to its correspondence with description logics [Description Logics], a field of research that has studied a particular decidable fragment of first order logic. OWL DL was designed to support the existing Description Logic business segment and has desirable computational properties for the reasoning systems.

- **OWL Full** is meant for users who want maximum expressiveness and the syntactic freedom of RDF with no computational guarantees. For example, in OWL Full a class can be treated simultaneously as a collection of individuals and as an individual in its own right. Another significant difference from OWL DL is that an OWL full datatype property may be inverse functional. OWL Full allows an ontology to augment the meaning of the pre-defined (RDF or OWL) vocabulary. It is unlikely that any reasoning software will be able to support every feature of OWL Full.

Each of these sublanguages is an extension of its simpler predecessor, both in what can be legally expressed and in what can be validly concluded. The following set of relations hold.

- Every legal OWL Lite ontology is a legal OWL DL ontology.
- Every legal OWL DL ontology is a legal OWL Full ontology.
- Every valid OWL Lite conclusion is a valid OWL DL conclusion.
- Every valid OWL DL conclusion is a valid OWL Full conclusion.

2.7.1. OWL 1.0 fundamental constructs

In this section, we will present the fundamental elements of OWL, which include the classes, properties and individuals. Every OWL 1 constrict is uniquely identified by an rdf:ID. Furthermore, every OWL construct may include human readable text, which is not taken in to consideration by reasoning mechanisms and is enclosed in labels or/and comments defined using the rdf:label and rdf:comment elements respectively.

The OWL classes represent sets of individuals that have common properties and belong to the same category. The most basic concepts in a domain should correspond to classes that are the roots of various taxonomic trees. Every individual in the OWL world is a member of the class owl:Thing. Thus, each user-defined class is implicitly a subclass of owl:Thing. Domain specific root classes are defined by simply declaring a named class. OWL also defines the empty class, owl:Nothing.

```
<owl:Class rdf:ID="RedWine"/>
<owl:Class rdf:ID="Winery"/>
```

Figure 2.32 OWL root classes

In Figure 2.32, we show two declarations of root classes inside an ontology. OWL classes are defined inside an element <owl:Class> as shown in Figure 2.32. The declarations shown above describes only the unique ID of the classes, without going deeper. A class can be defined as the union, intersection and complement of other classes of the ontology by using the constructs "owl:unionOf", "owl:intersectionOf" and "owl:complementOf" respectively, or as an enumeration of its members by using the construct "owl:oneOf".

Moreover, the fundamental taxonomic constructor for classes is rdfs:subClassOf. It relates a more specific class to a more general one. If X is a subclass of Y, then every instance of X is also an instance of Y. The rdfs:subClassOf relation is transitive. If X is a subclass of Y and Y a subclass of

```
<owl:Class rdf:about="RedWine">
  <rdfs:subClassOf rdf:resource="#Wine" />
  ...
</owl:Class>
```

Figure 2.33 OWL subclass definition

Z then X is a subclass of Z. In Figure 2.33 is used the "rdf:about" attribute within the "owl:Class" element in order to refer to the predefined "RedWine" class of Figure 2.32. This type of reference is used to extend the definition of a resource.

Through the example of Figure 2.33 is shown how the class "RedWine" is derived from the general class "Wine". The construct "rdf:about" is used because the class "RedWine" is already declared and at this moment we want to extend this class by relating it to a general class, through the subclass mechanism, in order to inherit the properties and the characteristics of "Wine".

Furthermore, two OWL classes may be regarded as equivalent or disjoint by using the mapping constructs "owl:equivalentClass" and "owl:disjointWith", respectively. The construct "owl:equivalentClass" will be used in later section 3.3.2 to define new datatypes and exploit the new features of OWL 2.0.

OWL individuals are the members OWL class. Instances of classes are defined as shown in Figure 2.34, using the "rdf:type" construct or the name of the class as the name of the element in which the individual is declared. The individuals may have the properties and satisfy the constraints that are valid for the OWL class in which they belong.

```
<RedWine rdf:ID= "Syrah" >

                          -OR-

<owl:Thing rdf:ID="Syrah" />
<owl:Thing rdf:about="#Syrah">
  <rdf:type rdf:resource="#RedWine"/>
</owl:Thing>
```

Figure 2.34 OWL class individual

OWL properties allow asserting general facts about the classes and specific facts about the class individuals. There are two categories of properties: object properties and datatype properties.

- **Object properties** are relations between the instances of two classes. An object property is described using the "owl:objectProperty" construct, which relates individuals of the domain class to individuals of the range class.

- **Datatype properties** are relations between class instances and RDF literals or XML Schema datatypes. A datatype property is described using the "owl:DatatypeProperty" construct, which relates individuals of the domain class to values of the range datatype.

```
<owl:Class rdf:ID="VintageYear" />
<owl:DatatypeProperty rdf:ID="yearValue">
   <rdfs:domain rdf:resource="#VintageYear" />
   <rdfs:range  rdf:resource="&xsd;positiveInteger"/>
</owl:DatatypeProperty>
```

Figure 2.35 OWL Datatype property definition

Figure 2.35 describes the definition of a datatype property which relates the vintage years of a wine production to positive integers. An instance of the VintageYear class is shown in Figure 2.36.

```
<VintageYear rdf:ID="Year1998">
  <yearValue rdf:datatype="&xsd;positiveInteger">1998</yearValue>
</VintageYear>
```

Figure 2.36 OWL Class instance with datatype property

The rest of this section describes the mechanisms used to further specify properties. It is possible to specify property characteristics, which provide a powerful mechanism for enhanced reasoning about a property. A property can be described as transitive, symmetric, functional, inverse or inverse functional.

If a property, P, is transitive then for any x, y, and z:

$$P(x,y) \text{ and } P(y,z) \text{ implies } P(x,z) \quad (2.1)$$

If a property, P, is symmetric then for any x and y:

$$P(x,y) \text{ iff } P(y,x) \quad (2.2)$$

If a property, P, is tagged as functional then for all x, y, and z:

$$P(x,y) \text{ and } P(x,z) \text{ implies } y = z \quad (2.3)$$

If a property, P1, is the owl:inverseOf P2, then for all x and y:

$$P1(x,y) \text{ iff } P2(y,x) \quad (2.4)$$

If a property, P, is tagged as InverseFunctional then for all x, y and z:

$$P(y,x) \text{ and } P(z,x) \text{ implies } y = z \quad (2.5)$$

In OWL Full, one may specify a DatatypeProperty as inverseFunctional. This allows us to identifying a string as a unique key. This need is covered by the new recommendation of OWL [W3C/OWL2] by the new features introduced in the syntax of OWL (see Figure 2.39).

In addition to designating property characteristics, it is possible to further constrain the range of a property in specific contexts in a variety of ways using the “owl:Restriction” construct. We can restrict the types of the elements (“owl:allValuesFrom” and “owl:someValuesFrom”), specify the cardinality (“owl:cardinality”, “owl:minCardinality” and “owl:maxCardinality”) and specify classes based on the existence of particular property values (“owl:hasValue”).

OWL allows declaring custom datatypes with the "rdf:Datatype" construct but not to define datatypes. Hence, in order to use custom datatypes within an ontology, we have to define those datatypes as simple types inside an XML Schema. Thus, the "rdf:Datatype" declaration refers to the simple type defined inside a specific XML Schema document. This drawback has been overcome with the OWL 2.0 recommendation [W3C/OWL2], which introduces the DatatypeDefintion axiom (see section 2.9 for datatypes OWL 2.0).

2.8. OWL 2.0

The great interest of the academia and the industry in semantic web has led to new W3C recommendations of existing standards in order to cover the needs of different domains. OWL has evolved to OWL 2.0 by adding even more expressiveness and potential of reasoning over the ontology axioms. The use cases that motivated the OWL 2 new features include brain image annotation for neurosurgery, the Foundational Model of Anatomy (human 'canonical' anatomy), the chemical compounds classification, querying multiple sources in an automotive company, OBO ontologies for biomedical data integration, spatial and topological relationships at the Ordnance Survey, the Systematized Nomenclature of Medicine, Simple part-whole relations in OWL Ontologies, the kidney Allocation Policy in France, Eligibility Criteria for Patient Recruitment, Protege report on the experiences of OWL users, Web service modeling, managing vocabulary in collaborative environments, the UML Association Class, database federation, virtual Solar Terrestrial Observatory, Semantic Provenance Capture and biochemical self-interaction. These use

cases belong to different domains like the chemical domain, earth and space, telecommunication, automotive and Health Care and Life Sciences, with the latter as the subject of the Semantic Web Health Care and Life Sciences (HCLS) Interest Group. Consequently, OWL 2.0 aims to cover complex reasoning among different domains and science fields.

In the OWL 2.0 Recommendation a new syntax is used to define the grammar of the language, that is **Functional-style syntax**. The functional-style syntax has been introduced to allow for easy writing of OWL 2 axioms (for example, see Figure 2.37 and Figure 2.38 below). Another benefit of the OWL 2 Functional Syntax is that it is closer to the syntax used in first order logic, which makes various specification issues as well as relating OWL 2 constructs to the general literature easier.

The main component of the OWL 2 ontology is a set of **axioms**. Axioms are statements that say what is true in the domain. OWL 2 provides an extensive set of axioms, all of which extend the Axiom class in the structural specification. Axioms can be declarations, axioms about classes, axioms about object or data properties, datatype definitions, keys, assertions (sometimes also called facts), and axioms about annotations.

The datatype Definition axiom has the syntax presented in Figure 2.37 and it specifies a new datatype DT as being semantically equivalent to the data range DR; the latter must be a unary data range. This axiom allows one to use the defined datatype DT as a synonym for DR — that is, in any expression in the ontology containing such an axiom, DT can be replaced with DR without affecting the meaning of the ontology.

DatatypeDefinition :=
'DatatypeDefinition' '(' **axiomAnnotations Datatype DataRange** ')'

Figure 2.37 DatatypeDefinition axiom

While OWL 1 allows a new class to be constructed by combining classes, it does not provide the means to construct a new datatype by combining other datatypes. In OWL 2 it is possible to define new datatypes in this way. In OWL 2, combinations of data ranges can be constructed using intersection (DataIntersectionOf), union (DataUnionOf), and complement (DataComplementOf) of data types. The data ranges are formally defined in Figure 2.38.

DataRange := Datatype | DataIntersectionOf | DataUnionOf |
DataComplementOf | DataOneOf | DatatypeRestriction

Figure 2.38 DataRange axiom

The following example demonstrates the datatype Definition axiom:

Example 2.2. Assume that a social security number (SSN) is a string that matches the given regular expression.

```
DatatypeDefinition(
   a:SSN
   DatatypeRestriction( xsd:string xsd:pattern "[0-9]{3}-[0-9]{2}-[0-9]{4}" )
)
```

OWL 2.0 supports keys natively by introducing **Key** axiom. The key axiom HasKey formally defined Figure 2.39 states that each (named) instance of the class expression CE is uniquely identified by the object property expressions OPEi and/or the data property experssions DPEj with $0 \leq i \leq n$ and $0 \leq j \leq m$, where n is the total number of the class object properties and m is the number of the class data type properties — that is, no two distinct (named) instances of CE can coincide on the values of all object property expressions OPEi and all data property expressions DPEj. In every key defined axiom in an OWL ontology, m or n (or both) must be larger than zero. A key axiom of the form HasKey(owl:Thing (OPE) ()) is similar to the axiom InverseFunctionalObjectProperty(OPE), with the main difference being that the former axiom is applicable only to individuals that are explicitly named in an ontology, while the latter axiom is also applicable to individuals whose existence is implied by existential quantification.

HasKey :=
'HasKey' '(' **axiomAnnotations**
ClassExpression '(' { **ObjectPropertyExpression** } ')'
'(' { **DataPropertyExpression** } ')'
')'

Figure 2.39 HasKey axiom

The following example demonstrates the datatype Definition axiom:

Example 2.3. Assume that each member of the family is uniquely identified by its name. The HasKey axiom shown below defines these semantics.

```
HasKey( a:FamilyMember () ( a:hasName ) )
```

Uniform Resource Locators (URIs) were used in OWL 1 to identify classes, ontologies, and other ontology elements. URIs are strings formed using a subset of the ASCII characters. This was quite limiting, particularly with respect to non-English language names since ASCII only includes letters from the English alphabet. To support broad international needs, OWL 2 uses Internationalized Resource Identifiers (IRIs) for identifying ontologies and their elements. IRIs are strings formed using UNICODE. In Figure 2.40 is presented the relation of the IRI, URI and URL sets using Venn diagrams.

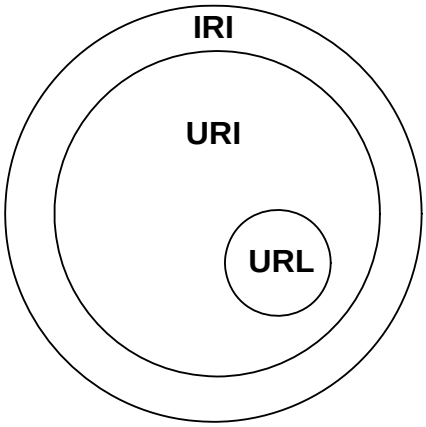

Figure 2.40 Diagram depicting the relation between IRI, URI and URL

2.9. SPARQL

SPARQL (Simple Protocol and RDF Query Language) [W3C/SPARQL] is the standard language for querying RDF data. SPARQL can be used to express queries across diverse data sources, whether the data is stored natively as RDF or viewed as RDF via middleware. SPARQL contains capabilities for querying required and optional graph patterns along with their conjunctions and disjunctions. SPARQL also supports extensible value testing and constraining queries using an RDF input graph. The SPARQL queiry results can be results sets or RDF graphs.

2.9.1. SPARQL Semantics

In this section is presented the SPARQL query language semantics.

Definition 2.2 RDF Triple

A tuple (s, p, o) ϵ (I ∪ B ∪ L) x I x (I ∪ B ∪ L) is called an RDF triple. In this tuple, s is the subject, p the predicate and o the object. Thus, an RDF triple may have:

- Subject: An IRI, Blank node or Literal value
- Predicate: An IRI value
- Object: An IRI, Blank node or Literal value

Definition 2.3 RDF Graph

An RDF graph is a set of RDF triples as defined in Definition 2.2.

Definition 2.4 RDF Dataset

An RDF dataset is a set :

$$D = \{G_0, \langle u_1,G_1\rangle, \ldots, \langle u_n,G_n\rangle\}$$

Where $G_0, \ldots, G_n$ are RDF graphs, $u_1, \ldots, u_n$ are IRIs, and $n > 0$. In the dataset, G_0 is the default graph, and the pairs $\langle u_i,G_i\rangle$ are named graphs, with u_i being the name of G_i.

Definition 2.5 Query Variable

A query variable is a member of the set infinite V, which is disjoint from the sets I, B, L, and T.

Definition 2.6 Triple Pattern

A triple pattern is member of the set: (T union V) x (I union V) x (T union V). The Triple pattern can be regarded as a cartesian product of RDF Triple and the set V.

This Triple Pattern definition includes literal subjects. SPARQL, throught does not allow literals to appear as subjects.

Definition 2.7 Basic Graph Patterns

A Basic Graph Pattern is a set of Triple Patterns as defined in Definition 2.6. Moreover, empty graph pattern is called the graph pattern which is the empty set. Also, a basic graph pattern may contain FILTER expressions as it was discussed in section 2.10.1 at paragraph “Basic Graph Pattern” (includes example, Figure 2.42).

Definition 2.8 SPARQL Graph Pattern

A SPARQL graph pattern expression is defined recursively as follows:

1. A basic graph pattern is a graph pattern.

2. If P1 and P2 are graph patterns, then the expressions (P1 AND P2), (P1 OPT P2), and (P1 UNION P2) are graph patterns (conjunction graph pattern, optional graph pattern, and union graph pattern, respectively).

3. If P is a graph pattern and R is a SPARQL built-in condition, then the expression (P FILTER R) is a graph pattern (a filter graph pattern).

Note: The variables of a filter R, may be part of a graph pattern P. The SPARQL recommendation is not clear about these use cases. Thus, this thesis makes the assumption that the filter variables are also included in the graph pattern P.

Definition 2.9 Subgraph Matching

Let G be an RDF graph, and P a basic graph pattern. The evaluation of P over G, denoted by [[P]]G is defined as the following set of mappings:

$$[[P]]_G = \{\mu : V \rightarrow T \mid dom(\mu) = var(P) \text{ and } \mu(P) \subseteq G\}$$

If $\mu \in [[P]]_G$, we say that μ is a solution for P in G.

Notice that: For every RDF graph G, $[[\emptyset]]_G = \{\mu_\emptyset\}$, i.e. the evaluation of an empty basic graph pattern against any graph always results in the set containing only the empty mapping. For every basic graph pattern $P \neq \emptyset$, $[[P]]_\emptyset = \emptyset$.

Definition 2.10 Solution Mapping

Solution mapping is the mapping between a set of variables (V) and a set of RDF Terms (T). In other words, a solution mapping, μ, is a partial function μ : V -> T. The domain of μ, dom(μ), is the subset of V where μ is defined.

Definition 2.11 Compatible Mappings

Two mappings $\mu_1 : V \rightarrow T$ and $\mu_2 : V \rightarrow T$ are compatible if for every ?X ? $dom(\mu_1) \cap dom(\mu_2)$ it is the case that $\mu_1(?X) = \mu_2(?X)$, i.e. when $\mu_1 \cap \mu_2$ is also a mapping.

Note: Two mappings with disjoint domains are always compatible, and the empty mapping $\mu_\emptyset$ is compatible with any other mapping.

Definition 2.12 Set of Mappings and Operations

Let Ω_1 and Ω_2 be sets of mappings. We define the join of, the union of, and the difference between Ω_1 and Ω_2 as:

$\Omega_1 \bowtie \Omega_2 = \{\mu_1 \in \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2$ and μ_1, μ_2 are compatible mappings$\}$,
$\Omega_1 \cup \Omega_2 = \{\mu \mid \mu \in \Omega_1$or $\mu \in \Omega_2\}$,
$\Omega_1 \setminus \Omega_2 = \{\mu \in \Omega_1 \mid$ for all $\mu' \in \Omega_2$, μ and μ' are not compatible$\}$.

Based on the previous operators, we define the left outer-join as

$\Omega 1 \ \overline{\bowtie}\ \Omega 2 = (\Omega 1 \bowtie \Omega 2) \cup (\Omega 1 \setminus \Omega 2)$

Definition 2.13 Evaluation of Graph Pattern

Let D be an RDF dataset and G an RDF graph in D. The evaluation of a graph pattern over G in the dataset D, denoted by $[\![.]\!]_G^D$, is defined recursively as follows:

1. $[\![(P_1 \text{ AND } P_2)]\!]_G^D = [\![P_1]\!]_G^D \bowtie [\![P_2]\!]_G^D$
2. $[\![(P_1 \text{ UNION } P_2)]\!]_G^D = [\![P_1]\!]_G^D \cup [\![P_2]\!]_G^D$
3. $[\![(P_1 \text{ OPT } P_2)]\!]_G^D = [\![P_1]\!]_G^D \ \overline{\bowtie}\ [\![P_2]\!]_G^D$
4. $[\![(P \text{ FILTER } R)]\!]_G^D = \{\mu \in [\![P]\!]_G^D \mid \mu \models R\}$, where $\mu \models R$ denotes that μ satisfies the R expression of the FILTER.

Definition 2.14 Equivalent Graph Pattern

Two graph patterns P_1 and P_2 are equivalent, denoted by P1 ≡ P2, if $[\![P_1]\!]_D = [\![P_2]\!]_D$ for every RDF dataset D.

Definition 2.15 Let $\{t_1, t_2, \ldots, t_n\}$ be a basic graph pattern, where $n \geq 1$ and every t_i is a triple pattern ($1 \leq i \leq n$). Then for every dataset D:

$$[\![\{t_1, t_2, ..., t_n\}]\!]_D = [\![\{t_1\} \text{ AND } \{t_2\} \text{ AND } ... \text{ AND } \{t_n\}]\!]_D$$

Definition 2.16 Safe Graph Pattern

We say that a graph pattern Q is safe if for every subpattern (P FILTER R) of Q holds that var(R) $\subseteq$ var(P). This safety condition exist in many database query languages.

Definition 2.17 Well Designed Graph Patterns

A UNION-free graph pattern P is well designed if P is safe and, for every subpattern P = (P1 OPT P2) of P and for every variable ?X occurring in P, the following condition holds:

if ?X occurs both inside P2 and outside P, then it also occurs in P1

Definition 2.18 RDF Terms

Assume there are pairwise disjoint infinite sets I, B, and L for IRIs, Blank nodes, and literals respectively. We denote the union I ∪ B ∪ L by T. The set of RDF Terms is the set T (RDF-T).

Definition 2.19 Union-Free Graph Patterns

A Union-Free Graph Pattern *UF-GP* is a Graph Pattern *GP* that does not contain Union operators.

Definition 2.20 Shared Variable

A variable contained in a Union Free Graph Pattern is called a Shared Variable when it is referenced in more than one triple patterns of the same Union Free Graph Pattern regardless of its position in those triple patterns.

2.9.2. Query Forms

SPARQL has four query forms. These query forms use the solutions from pattern matching to form result sets or RDF graphs. The query forms are: a) SELECT, b) CONSTRUCT, c) ASK and d) DESCRIBE.

a. SELECT

The SELECT result clause returns a table of variables and values that satisfy the query. SELECT * selects all the variables mentioned in the query, otherwise the SELECT keyword is populated with the variable names for which the clause returns results.

For example, the RDF data shown below describe the friends of Alice and the SPARQL query, which follows the data, retrieves the names of the friends for a person that has friends. In particular, the query returns the result set that is presented at the end of the example.

RDF Data

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:a foaf:name "Alice" .
_:a foaf:knows _:b .
_:a foaf:knows _:c .
_:b foaf:name "Bob" .
```

```
_:c foaf:name "Clare" .
```

Query

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?nameX ?nameY
WHERE
{ ?x foaf:knows ?y .
  ?x foaf:name ?nameX .
  ?y foaf:name ?nameY .
}
```

Result

nameX	nameY
“Alice”	“Bob”
“Alice”	“Clare”

b. CONSTRUCT

CONSTRUCT is a SPARQL result clause alternative to SELECT. Instead of returning a table of result values, CONSTRUCT returns an RDF graph.

The resulting RDF graph is created by taking the results of the equivalent SELECT query and filling in the values of the variables that occur in the CONSTRUCT template.Triples are not created in the result graph for template patterns that involve an unbound variable.

If any such instantiation produces a triple containing an unbound variable or an illegal RDF construct, such as a literal in the subject or predicate position, then that triple is not included in the resulting RDF graph. The graph template may contain triples with no variables (known as ground or explicit triples), and these also appear in the output RDF graph returned by the CONSTRUCT query form.

For example, the RDF data shown below describe the contact information of Alice and the SPARQL query, which follows the data, retrieves the names of the persons that are defined among the RDF data. In particular, the query constructs a RDF graph that is based on the vcard vocabulary [W3C/VCard] as shown in the result set that is presented at the end of the example.

RDF Data

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:a foaf:name "Alice" .
_:a foaf:mbox <mailto:alice@example.org> .
```

Query

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX vcard: <http://www.w3.org/2001/vcard-rdf/3.0#>
CONSTRUCT { <http://example.org/person#Alice> vcard:FN ?name
}
WHERE { ?x foaf:name ?name }
```

Result

```
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
<http://example.org/person#Alice> vcard:FN "Alice" .
```

A template can create an RDF graph containing blank nodes. The blank node labels are scoped to the template of each solution. If the same label occurs twice in a template, then there will be one blank node created for each query solution, but there will be different blank nodes for the triples generated by different query solutions.

For example, the RDF data shown below describe the personal information of two persons using blank nodes and the SPARQL query, which follows the data, retrieves the names of the persons that are defined among the RDF data. In particular, the query constructs a RDF graph that is based on the vcard vocabulary [W3C/VCard] as shown in the result set that is presented at the end of the example. It is worthwhile to mention that the _:a and _:b blank nodes are replaced with the _:x1 and _:x2 blank nodes, respectively, due to the number of the result nodes and the name of the variable that refers to individuals within the SPARQL query. However, the names of the blank nodes do not affect the semantics of the result set.

RDF Data

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:a foaf:givenname "Alice" .
_:a foaf:family_name "Smith" .
_:b foaf:firstname "Bob" .
_:b foaf:surname "Smith" .
```

Query

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX vcard: <http://www.w3.org/2001/vcard-rdf/3.0#>
CONSTRUCT { ?x vcard:N _:v .
                    _:v vcard:givenName ?gname .
                    _:v vcard:familyName ?fname }
WHERE
{
   { ?x foaf:firstname ?gname } UNION { ?x foaf:givenname ?gname } .
   { ?x foaf:surname ?fname } UNION { ?x foaf:family_name ?fname } .
}
```

Result

```
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
_:x1 vcard:N _:v .
_:v vcard:givenName "Alice" .
_:v vcard:familyName "Smith" .
_:x2 vcard:N _:z .
_:z vcard:givenName "Bob" .
_:z vcard:familyName "Smith" .
```

c. ASK

Applications can use the ASK form to test whether or not a query pattern has a solution. No information is returned about the possible query solutions. The ASK result clause simply returns true or false depending on whether or not the query pattern has any matches in the dataset. As with SELECT queries, the boolean result is (by default) encoded in a SPARQL Results Format XML document.

For example, the RDF data shown below describea subset of the personal information about two individuals and the SPARQL query, which follows the data, specifies if there exists any individual with name “Alice” among the RDF data. In particular, the query returns the “yes”l iteral, as shown in the result set that is presented at the end of the example.

RDF Data

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:a foaf:name "Alice" .
_:a foaf:homepage <http://work.example.org/alice/> .
_:b foaf:name "Bob" .
_:b foaf:mbox <mailto:bob@work.example> .
```

Query

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
ASK { ?x foaf:name "Alice" }
```

Result

```
yes
```

d. DESCRIBE

The DESCRIBE form returns a single result RDF graph containing RDF data about resources. This data is not prescribed by a SPARQL query, where the query client would need to know the RDF of structure the data source, but, instead, is determined by the SPARQL query processor. The query pattern is used to create a result set. The DESCRIBE form takes each of the resources identified in a solution, together with any resources directly named by an IRI, and assembles a single RDF graph by taking a "description" which can come from any information available including the target RDF Dataset. The description is determined by the query service. The syntax DESCRIBE * is an abbreviation that describes all of the variables in a query.

The RDF graph returned is determined by the information publisher. It is the useful information the service has about a resource. It may include information about other resources: for example, the RDF data for a book may also include details about the author.

For example, the SPARQL query retrieves any available information about the employee of a specific ID value.

Query

```
PREFIX ent: <http://org.example.com/employees#>
DESCRIBE ?x WHERE { ?x ent:employeeId "1234" }
```

Possible Result set

```
@prefix foaf:   <http://xmlns.com/foaf/0.1/> .
@prefix vcard:  <http://www.w3.org/2001/vcard-rdf/3.0> .
@prefix exOrg:  <http://org.example.com/employees#> .
@prefix rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix owl:    <http://www.w3.org/2002/07/owl#>

_:a     exOrg:employeeId    "1234" ;

        foaf:mbox_sha1sum   "ABCD1234" ;
        vcard:N
         [ vcard:Family       "Smith" ;
           vcard:Given        "John"  ] .

foaf:mbox_sha1sum  rdf:type  owl:InverseFunctionalProperty .
```

2.9.3. Solution Sequence Modifiers

The query patterns generate unordered collections of solutions, each solution being a partial function from variables to RDF terms. These solutions are then treated as a sequence (a solution sequence), initially in no specific order; any sequence modifiers are then applied to create another sequence. Finally, this latter sequence is used to generate one of the results of a SPARQL query form. The Solution Sequence Modifier set includes: a) the order modifier, b) the distinct modifier, c) the reduced modifier, d) the offset modifier and e) the limit modifier. All the modifiers are applied to queries of the form SELECT, CONSTRUCT or DESCRIBE and not to queries of the ASK form. The DISTINCT and REDUCE modifiers are allowed to be used only within SELECT queries.

a. ORDER BY modifier

The ORDER BY clause establishes the order of a solution sequence dased on the value of one or more variables. An ORDER BY clause is a sequence of order comparators, composed of an expression and an optional order modifier (either ASC() or DESC()). Every ordering comparator

may be either ascending (indicated by the ASC() modifier or by no modifier) or descending (indicated by the DESC() modifier).

For example, the SPARQL query shown below retrieves the names of the persons of an RDF repository which is structures according to the foaf namespace [W3C/FOAF].

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name
WHERE { ?x foaf:name ?name ; :empId ?emp }
ORDER BY ?name DESC(?emp)
```

In particular, the order by clause orders the solution sequence by ascending order on the ?name variable. In case of equivalent values of the ?name variable among the sequence tuples, the second expression of the ORDER BY modified causes the descending ordering among those on the ?emp variable.

b. DISTINCT modifier

The DISTINCT solution modifier eliminates the duplicate solutions. In particular, each solution that binds the same variables to the same RDF terms as another solution is eliminated from the solution set.

For example, the SPARQL query shown below retrieves the names of the persons that are declared among the RDF data of the example. In particular, the query is presented in two different versions. The first one does not use the DISTINCT modifier and the result set contains the same individual more than one times, as it is shown below, in contrast to the second one that uses the DISTINCT modifier on the returned variable and the result set does not contain duplicates.

RDF Data

```
prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:x foaf:name "Alice" .
_:x foaf:mbox <mailto:alice@example.com> .

_:y foaf:name "Alice" .
_:y foaf:mbox <mailto:asmith@example.com> .

_:z foaf:name "Alice" .
_:z foaf:mbox <mailto:alice.smith@example.com> .
```

Query without DISTINCT modifier

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name WHERE { ?x foaf:name ?name }
```

Result of the Query without DISTINCT modifier

name
“Alice”
“Alice”
“Alice”

Query with DISTINCT modifier

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT DISTINCT ?name WHERE { ?x foaf:name ?name }
```

Result of the Query with DISTINCT modifier

name
“Alice”

c. REDUCED modifier

While the DISTINCT modifier ensures that the duplicate solutions are eliminated from the solution set, the REDUCED modified allows them to be eliminated. The cardinality of any set of variable bindings in a REDUCED solution set is at least one and not more than the cardinality of the solution set with no DISTINCT or REDUCED modifier.

For example, the last query presented above would be like:

Query with REDUCED modifier

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT REDUCED ?name WHERE { ?x foaf:name ?name }
```

In contrast to the DISTINCT query, presented in the previous paragraph, which allows only one appearance of the distinct values among the result set, this query may have one, two (as shown here) or three solutions depending on the SPARQL Query engine that executes it.

d. OFFSET modifier

OFFSET is used optionally in conjunction with the LIMIT and ORDER BY modifiers in order to take a slice of a sorted solution set (e.g. for paging). The OFFSET modifier causes the solutions generated to start after the specified number of solutions.

For example, the SPARQL query shown below retrieves the names of the persons of a RDF repository ordered by their name. Moreover, the first 10 solutions of the result set are excluded according to the OFFSET modifier.

Query

```
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>

SELECT  ?name
WHERE   { ?x foaf:name ?name }
ORDER BY ?name
OFFSET  10
```

e. Limit modifier

LIMIT is a solution modifier that limits the number of rows returned from a query. The LIMIT clause puts an upper bound on the number of the solutions returned. If the number of actual solutions is greater than the limit, then at most the limit number of solutions will be returned. Moreover, a limit may not be negative and if it is equal to zero that would cause no results to be returned.

For example, the SPARQL query shown below retrieves the names of 20 persons of a RDF repository. The limitation at the number of the solutions of the result set is specified by the LIMIT modifier.

Query

```
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>

SELECT ?name
WHERE { ?x foaf:name ?name }
LIMIT 20
```

This query would return no more than 20 results.

2.9.4. Graph Patterns

SPARQL is essentially a graph pattern matching query language. The body of the query is a complex RDF graph pattern expression that may include RDF triples with variables, conjunctions, disjunctions, optional parts, and constraints over the values of the variables. The pattern types that contribute a graph pattern expression are: a) the Triple patterns, b) the Basic Graph patterns, c) the Group Graph patterns, d) the Optional Graph patterns and e) the Alternative Graph patterns.

Triple patterns

The triple patterns are the simplest form of graph patterns. They are just like RDF triples (subject – predicate – object), except that any of the parts of a triple can be replaced with a variable. The SPARQL variables start with a question mark (?) and can match any node (resource or literal) in the RDF dataset. Some examples of triple patterns are shown in Figure 2.41.

```
a) ?person foaf:name ?name
b) <http://www.w3.org/People/Berners-Lee/card#i> foaf:name ?name
c) ?person ?p "Tim"
d) ?s ?p ?o
```

Figure 2.41 Triple pattern examples

In Figure 2.41, we present four use cases of triple patterns. In the first case (a), there is a triple pattern having two distinct variables in the place of subject and object, and the predicate is the property name found under the namespace of foaf [FOAF]. The second case (b) uses a URI for the subject, the propeprty name of the foaf namespace for the predicate and a variable for the object. The third case (c) has variables in both the subject and predicate positions, and the object of the triple is the literal “Tim”. Finally, in the last case (d) the subject is the variable s, the predicate the variable p and the object the variable o.

Basic Graph patterns

The basic graph patterns are sets of triple patterns joined with the (.) operator. The SPARQL graph pattern matching is defined in terms of combining the results from matching basic graph patterns. A sequence of triple patterns interrupted by a filter comprises a single basic graph pattern. The SPARQL FILTERs eliminate solutions that do not cause an expression to evaluate to true. A FILTER is made of expressions, which contain functions from a broad set of built-in functions, literals (e.g. 5, "Ioannis") and variables. Built-in functions may be logical functions (!, &&, ||), math functions

```
?x ns:FirstName "Tim" ;
      ns:LastName ?lastname;
      ns:Phone ?phone.
FILTER(?lastname="Berners-Lee")
```

Figure 2.42 Basic Graph Pattern example

(+, -, *, /), functions for comparison (=, !=, >, <, etc.), SPARQL tests (isURI, isBlank, isLiteral, bound), SPARQL accessors (str, lang, datatype) and other functions like sameTerm, langMatches and regex. A semicolon (;) can be used to separate two triple patterns that share the same subject. Figure 2.42 shows a basic graph pattern with a FILTER expression.

When using blank nodes of the form _:abc, labels for blank nodes are scoped to the basic graph pattern. A label can be used only in a single basic graph pattern in any query.

Group Graph patterns

The group graph patterns for in the most general category of SPARQL graph pattern as they may contain any type of the rest graph patterns. In a SPARQL query string, a group graph pattern is delimited with braces: {}.

The group pattern { } is called the empty group pattern and matches any graph with one solution that does not bind any variable.

In Figure 2.43 we present a group graph pattern that contains the basic graph pattern of Figure 2.42.

```
{ ?x ns:FirstName "Tim" ;
       ns:LastName ?lastname;
       ns:Phone ?phone.
  FILTER(?lastname="Berners-Lee") }
```

Figure 2.43 Group Graph Pattern example

Optional Graph patterns

Basic graph patterns allow applications to make queries where the entire query pattern must match in order to be a solution. It is useful to be able to have queries that allow information to be added to the solution if the query pattern is partially match. Optional matching provides this

```
OPTIONAL { ?x  ns:FirstName "Tim" ;
                    ns:LastName ?lastname.
                }
```

Figure 2.44 Optional Graph Pattern example

fanctionality: if the optional part does not match, it creates no bindings but does not eliminate the solution. Optional graph patterns are declared by using the OPTIONAL keyword and the curly braces { } to enclose the graph pattern on which optional matching is applied.

In Figure 2.44, we describe an optional graph pattern. If there is an individual with first name (ns:FirstName) “Tim” and some last name (ns:LastName), a solution will contain the object of that triple, otherwise it will be ignored.

A graph pattern may have zero or more optional graph patterns, and any part of a query pattern may have an optional part.

Alternative Graph patterns

SPARQL provides a means of combining graph patterns so that one of a number of alternative graph patterns may match. If more than one of the alternatives matches, all the possible pattern solutions are found. Graph pattern alternatives are syntactically specified using the UNION keyword.

```
{{ ?x  ns:FirstName "Tim". } UNION { ?x  ns:FirstName "John". }}
```

Figure 2.45 Alternative Graph Pattern

The result of the graph Pattern of Figure 2.45 will include persons whose first name is “Tim” or “John”.

2.10. XS2OWL 1.0 Framework

In this section is presended the XS2OWL 1.0 [TsChXS2OWL] transformation model. XS2OWL 1.0 was developed in the context of the PhD thesis of Chrisa Tsinaraki XS2OWL [TsPhD08] and the ancestor of XS2OWL 2.0, developed in this thesis. XS2OWL allows transforming the XML Schema constructs in OWL, so that applications using XML Schema based standards will be able to use the Semantic Web methodologies and tools. XS2OWL 1.0 also supports the conversion of the OWL-based constructs back to the XML Schema based constructs in order to maintain the compatibility with the XML schema versions of the standards. XS2OWL has been implemented as an XML Stylesheet Transformation Language (XSLT) [W3C/XSLT] stylesheet and transforms every XML Schema based standard in an OWL-DL Main Ontology. This way, the constructs of the standard become first class Semantic Web objects and may be integrated with domain knowledge expressed as OWL domain ontologies. In addition, all the OWL-based Semantic Web tools, including reasoners, can be used with the standard based descriptions. In Figure 2.46 is presented the architecture of the XS2OWL framework.

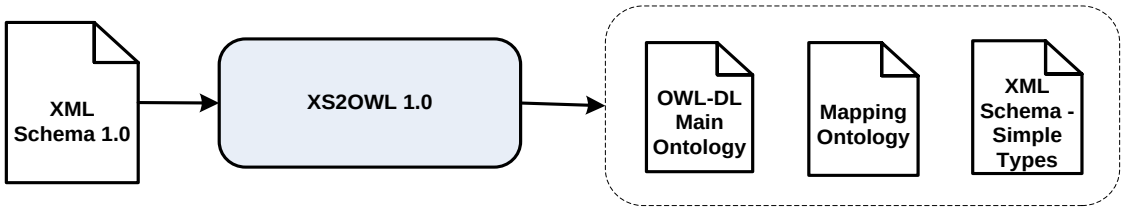

Figure 2.46 XS2OWL 1.0 Framework

As shown in Figure 2.46, the XS2OWL 1.0 transforms a source XML Schema in OWL syntax: The OWL representation includes:

- An OWL-DL **main Ontology** that represents the XML Schema constructs using semantically equivalent OWL constructs.

- A **mapping ontology** that associates the names of the XML Schema constructs with the IDs of the equivalent main ontology constructs and captures any information present in the XML Schema that cannot be captured in the main ontology due to the expressivity limitations of the OWL 1.0 syntax.
- An **XML Schema of Simple Types** that contains the simple types defined in the source XML Schema and are used inside the main ontology.

The XS2OWL1.0 model allows automatically transforming XML Schema constructs to OWL-DL constructs without loosing any information. This way, computational completeness and decidability of reasoning are guaranteed in the OWL ontologies produced and back transformations are supported.

2.10.1. Representation Model Overview

This section describes the model for the direct transformation of the XML Schema constructs in OWL-DL. As already mentioned, the result of the transformation of a source XML Schema is an OWL-DL main ontology, that captures the semantics of the XML Schema constructs, a mapping ontology.

Table 2.4 presents an overview of the XS2OWL 1.0 transformation model: The first column contains the XML Schema constructs and the rest describe the constructs generated from the XS2OWL 1.0 processor. In particular, the second column contains to the main ontology constructs, the third one contains to the mapping ontology constructs and the last one contains the XML Schema constructs in the main ontology. As it is shown below, complex XML Schema types are represented in the main ontology as classes, since both the OWL classes in OWL and the XML Schema complex types describe sets of entities with common features. The required information for mapping the complex XML Schema types to the OWL classes, and the schema information that cannot be captured in the main ontology due to the expressivity limitations of the OWL 1.0 syntax, is represented by individuals of "ComplexTypeInfoType" in the mapping ontology.

XML Schema 1.0	**XS2OWL 1.0**		
	Main Ontology	Mapping Ontology	XMLS Simple Types
xs:complexType	owl:Class	Individual ComplexTypeInfoType	
xs:simpleType	rdfs:Datatype		xs:simpleType
xs:element	owl:DatatypeProperty ή owl:ObjectProperty	Individual ElementInfoType	
xs:attribute	owl:DatatypeProperty	Individual DatatypePropertyInfoType	
xs:all	owl:Class – owl:intersectionOf		
xs:sequence	owl:Class – owl:intersectionOf	Individual SequenceInfoType	

xs:choice	owl:Class – owl:UnionOf	Individual ChoiceInfoType	
xs:annotation	rdfs:comment		

Table 2.4 XS2OWL 1.0 Transformation Model Overview

Likewise, the simple XML Schema types are declared in the main ontology as datatypes and are defined in the XML Schema of Simple Types. It was chosen in XS2OWL 1.0 this representation since OWL 1.0 only allows the declaration of datatypes defined in XML Schemas and prohibits native datatype definitions inside an ontology.

The complex XML Schema type attributes are represented as datatype properties in the main ontology, because both the XML Schema attributes and the OWL datatype property specify features with simple type values. The required information for mapping the XML Schema attributes and to OWL properties and the attribute semantics that cannot be captured in the main ontology, are represented by individuals of the "DatatypePropertyInfoType" class in the mapping ontology.

The complex XML Schema type elements are represented as object or datatype properties, in case of elements of complex and simple types respectively. Both XML Schema elements and OWL properties specify features. The required information for mapping the XML Schema elements and to OWL properties and the elements semantics that cannot be captured in the main ontology, are represented by individuals of "ElementInfoType" class in the mapping ontology.

The XML Schema sets (xs:all), sequences and choices are represented as unnamed classes in the main ontology, defined using the intersection (for the sets and sequences) and union (for the choices) operators. The sequence and choice semantics that cannot be captured in the main ontology are represented by individuals of "SequenceInfoType" and "ChoiceInfoType" classes respectively in the mapping ontology.

Finally, the Schema construct annotations are represented as labels of the corresponding OWL constructs in the main ontology.

The transformation of the individual XML Schema constructs is presented in sections 2.11.2 – 2.11.7 and the mapping ontology is described in section 2.11.8.

2.10.2. Simple XML Schema Datatypes

OWL does not directly support the definition of simple datatypes; it only allows importing simple datatypes. Existing XML Schema datatypes may be used in OWL ontologies if they have been declared in them. XS2OWL organizes all the simple XML Schema datatype definitions in the "datatypes" XML Schema and for each of them it generates an OWL datatype declaration.

Let `st` be the XML Schema Simple Type that is described by expression (2.1)

st (name, id, body) (2.1)

Body is the body of the definition of *st*, *id* is the (optional) identifier of *st* and *name* is the name of *st*. *st* is transformed into:

a. The st'(name', id, body) simple datatype, which is stored in the "datatypes" XML Schema; and
b. The dd(about, is_defined_by, label) datatype declaration in the main ontology.

2.10.3. Attributes

XML Schema attributes describe features with values of simple type. The OWL construct that can represent such features is the datatype property. Thus, XS2OWL transforms the XML Schema attributes into OWL datatype properties.

Let *a* be an XML Schema attribute that is described by expression 2.2

a(name, aid, type, annot, ct_name, fixed, default) (2.2)

name is the name of *a*, *aid* is the identifier of *a*, *type* is the type of *a*, *annot* is an (optional) annotation element of *a*, *ct_name* is the name of the complex XML Schema type c_type in the context of which *a* is defined (if *a* is a top-level attribute, ct_name has the null value), *fixed* is the (optional) fixed value of *a* and default is the (optional) default value of *a.* XS2OWL, transforms *a* into the OWL datatype property dp that is described by expression 2.3

dp(id, range, domain, label, comment) (2.3)

Where *id* is the unique rdf:ID of *dp* and has concatenate(*name*, '__', *type*) as value; *range* is the range of *dp* and has *type* as value; *domain* is the domain of *dp* and has *ct_name* as value; *label* is the label of *dp* and has *name* as value; and *comment* is the textual description of *dp* and has *annot* as value. If any of the features of *a* is absent, the corresponding feature of *dp* is also absent.

Notice that: (a) If a *fixed* value of *a* is specified, it is represented as a value restriction in the definition of the OWL class *c* that represents c_type; and (b) If a *default* value of *a* is specified, it cannot be represented in the main ontology.

2.10.4. Elements

XML Schema elements represent features of complex XML Schema types and are transformed into OWL properties. The simple type elements are represented as OWL datatype properties and the complex type elements are represented as OWL object properties.

Let *e* be an XML Schema element that is described by expression 2.4

e(name, type, eid, annot, ct_name, substitution_group) (2.4)

name is the name of *e*, *eid* is the identifier of *e*, *type* is the type of *e*, *annot* is an annotation element of *e*, *ct_name* is the name of the complex XML Schema type *c_type* in the context of which *e* is defined (if *e* is a top-level attribute, *ct_name* has the null value) and *substitution_group* is an (optional) element being extended by *e*.

The *e* element is represented in OWL as a (datatype or object) property *p, where*:

p(id, range, domain, label, comment, super_property) (2.5)

id is the unique rdf:ID of *p* and has concatenate(name, '__', type) as value; *range* is the range of *p* and has *type* as value; *domain* is the domain of *p* and has *ct_name* as value; *label* is the label of *p* and has *name* as value; *comment* is the textual description of *p* and has *annot* as value; and *super_property* is the specification of the property specialized by *p* and has *substitution_group* as value.

2.10.5. Complex Types

The XML Schema complex types represent classes of XML instances that have common features, just as the OWL classes represent sets of individuals with common properties. Thus, XS2OWL transforms the XML Schema complex types into OWL classes.

Let *ct* be an XML Schema complex type that is described by expression (2.6)

ct(name, cid, base, annot, attributes, sequences, choices) (2.6)

name is the name of *ct*; *aid* is the identifier of *ct*; *base* is the (simple or complex) type extended by *ct*; *annot* is an annotation element of *ct*; *attributes* is the list of the attributes of *ct*; *sequences* is the list of the *ct* sequences; and *choices* is the list of the *ct* choices.

If ct extends a complex type, XS2OWL transforms it to the OWL class c(id, super_class, label, comment, value_restrictions, cardinality_restrictions), where: (a) id is the unique rdf:ID of c and has name as value if ct is a top-level complex type. If ct is a complex type nested within the definition of an element e, name is a unique, automatically generated name of the form concatenate(ct_name, '_', element_name, '_UNType'), where ct_name is the name of the complex type containing e and element_name is the name of e. If e is a top-level element, ct_name has the 'NS' value;(b) super_class states which class is extended by ct and has base as value; (c) label is the label of ct and has name as value; (d) comment is the textual description of ct and has annot as value; (e) value_restrictions is the set of the value restrictions holding for the properties of c; and (f) cardinality_restrictions is the set of the cardinality restrictions assigned to the properties representing the ct attributes and the ct sequence/choice elements.

If ct extends a simple type, XS2OWL transforms it to the OWL class c(id, label, comment, value_restrictions, cardinality_restrictions), with the same semantics with the classes representing complex types that extend complex types on the corresponding items. The extension of the simple type is represented by the datatype property ep(eid, erange, edomain) of cardinality 1, where: (a) eid is the unique rdf:ID of ep and has concatenate(base, '_content') as value; (b) range is the

range of ep and has base as value; and (c) domain is the domain of ep and takes as value the id of c. The attributes and the elements that are defined or referenced in ct are transformed to the corresponding OWL-DL constructs.

2.10.6. Sequences and Choices

The XML Schema sequences and choices essentially are XML element containers, defined in the context of complex types. The main difference between sequences and choices is that the sequences are ordered, while the choices are unordered. XS2OWL transforms both the sequences and the choices into unnamed OWL-DL classes featuring complex cardinality restrictions on the sequence/choice items (elements, sequences and choices) and places them in the definition of the classes that represent the complex types where the sequences/choices are referenced or defined.

2.10.7. References

XML Schema attributes and elements that are referenced in complex type definitions are transformed into OWL-DL datatype (if they are or contain attributes or simple type elements) or object (if they contain complex type elements) properties.

Let ref(ae) be a reference, in a complex type *ct*, to the *ae* XML attribute or element. The reference is represented by the (datatype or object) property rp(id, domain), where *id* is the rdf:ID of *rp* and has as value the value of the rdf:ID of the property that represents *ae*, and domain is the domain of *rp* and has the rdf:ID of the OWL class *c* that represents *ct* as value.

2.10.8. Mapping Ontology

In this section is described the mapping ontology focusing on the ontology classes and properties. As already mentioned the mapping ontology captures the XML Schema semantics that cannot be directly transformed in OWL syntax.

The mapping ontology keeps information that is not usable by the Semantic Web tools, but can be of use in other applications like, for example, the transformation of RDF data structures according to the main ontology in XML syntax compliant with the original XML Schemas.

The mapping ontology contains five classes that are described in the following subsections: “ComplexTypeInfoType” is presented in subsection 2.11.8.1, “DatatypePropertyInfoType” is presented in subsection 2.11.8.2, “ElementContainerType” is presented in subsection 2.11.8.3, “ElementInfoType” is presented in subsection 2.11.8.4, and class “ElementRefType” is presented in subsection 2.11.8.5.

2.10.8.1. Class ComplexTypeInfoType

The "ComplexTypeInfoType" Class, which is described in this subsection, captures information about the XML Schema complex types. This information has two aspects: a) Information mapping XML Schema complex types with OWL-DL classes; and b) Information about XML Schema complex types that cannot be directly expressed in OWL syntax. A detailed description of the information encoded in the individuals of the "ComplexTypeInfoType" class, is presented below.

Information mapping XML Schema complex types with OWL-DL classes. A "ComplexTypeInfoType" individual that represents an XML Schema type T, has the datatype properties "typeID" and "classID".

The datatype property "typeID" has cardinality of zero or one and represents the name of the XML Schema type T.

The datatype property "classID", with cardinality value one, has an value the ID of the OWL class that represents XML Schema type T in main ontology.

Information about XML Schema complex types that cannot be directly expressed in OWL syntax. The information about XML Schema complex types that cannot be represented by OWL constructs includes the datatype property "abstract", the datatype property "final", the object property "DatatypePropertyInfo" and the object property "ElementContainer".

The datatype property "abstract" is of Boolean type and has a cardinality of zero or one. It represents the value of the "abstract" attribute of the XML Schema complex type T, which specifies if T is abstract and thus has no valid instances. If the datatype property "abstract" is absent, the implied default value is "false".

The datatype property "final" is of Boolean type and has a cardinality of zero or one. It represents the value of the "final" attribute of the XML Schema complex type T, which specifies if T is final and thus no further type derivations are possible. If the datatype property "final" is absent, the implied default value is "false".

The datatype property "DatatypePropertyInfo" is of type "DatatypePropertyInfoType" and has no cardinality restrictions. It allows the representation of information about datatype properties of the class that represent the XML Schema complex type T in the ontology.

The object property "ElementContainer" is of type "ElementContainerType" and has no cardinality restrictions. It allows the representation of information about the cardinality and the order of the sequence and choice elements of of the XML Schema complex types.

2.10.8.2. Class DatatypePropertyInfoType

The class “DatatypePropertyInfoType”, which is described in this subsection, captures information about the datatype properties of main ontology. Every datatype property do this information is specify in the values datatype of the properties “datatypePropertyID”, “XMLConstructID”, “datatypePropertyType” and “defaultValue”.

The datatype property “datatypePropertyID” is of type string, has a cardinality of one and has the value of the ID of the datatype property DP.

The datatype property “XMLConstructID” is of type string, has cardinality of zero or one and has the value of the name of the XML Schema construct (attribute or element of simple type) that is represented by the datatype property DP. The absence of datatype property “XMLConstructID” means that the type represented by the domain class of is extension of a simple type.

The datatype property “datatypePropertyType” is of type string, has cardinality of one and has the value of the source of the datatype property DP. It may have one of the following values:

- “Attribute” if the construct represented by DP is an attribute.
- “Element” if the construct represented by DP is an element.
- “Extension” if datatype property DP does not representing a construct, but specifies that the type represented by the domain class of DP is an extension of the simple type that forms the range of property.

The datatype property “defaultValue” is of type string, has cardinality of zero or one and holds the optional default value of the construct represented by DP. The absence of “defaultValue” means that DP does not have a default value.

2.10.8.3. Class ElementContainerType

The class “ElementContainerType”, which is described in this subsection, captures information about XML Schema sequences and choices. For every sequence or choice C this inform includes the exact cardinality of C, the members of C and the position of C (if C is nested in another sequence or choice).

Information about the exact cardinality of C. The information about the exact cardinality of XML Schema sequence or choice C consist of the values of the datatype properties “minOccurs” and “maxOccurs”.

The datatype property “minOccurs” is of type string, has cardinality of 1 and represents the minimum number of occurrences of C.

The datatype property “maxOccurs” is of type string, has cardinality of 1 and represents the maximum number of occurrences of C.

Information about the members of C. The information about the members of the XML Schema sequence or choice C contained in by the values of the object property “Item”. As far as the members of C, that are sequences or choices, are concerned, the value of the property “Item” is of type “ElementContainerType” and for the members of C that are elements (including members of nested groups, regarded as elements), the value of “Item” the property is of type “ElementRefType”.

Information about the position of C. In case that the XML Schema sequence or choice C is nested in another sequence or choice, information about the position of C is represented by the datatype property “itemPosition”, with a cardinality of one.

The class “ElementContainerType” is extended by the classes “ChoiceInfoType” and “SequenceInfoType”, which capture information about choices and XML Schema sequences respectively.

2.10.8.4. Class ElementInfoType

The Class “ElementInfoType”, which is described in this subsection, captures information about the XML Schema elements. This information has two aspects: a) Information mapping XML Schema elements with OWL-DL properties; and b) Information about the XML Schema elements that cannot be directly expressed in OWL syntax.

Information about mapping XML Schema elements with OWL-DL properties. Let E be an XML Schema element. Information about mapping XML Schema elements with OWL-DL properties, consists of datatype property “elementID”, datatype property “propertyID” and datatype property “isRoot”.

The datatype property “elementID” is of type string, has a cardinality of one and represents the name of the XML Schema element E.

The datatype property “propertyID” is of type string, has a cardinality of one and holds the ID of the property, that represents E in the main ontology.

The datatype property “isRoot” is of type boolean, has a cardinality of zero or one and is true if E is the top element of XML Schema. The absence of the datatype property “isRoot” implies the default value “false”.

Information about the XML Schema elements that cannot be directly expressed in OWL syntax. The information about XML Schema elements that cannot be expressed natively with OWL syntax is captured in the datatype property “block” with a cardinality of zero or one. “block” represents the value of the attribute “block” of E, which prohibits substitutions if it is true. The absence of the datatype property “block” implies the default value “false”.

2.10.8.5. Class ElementRefType

The class “ElementRefType”, which is described in this subsection, depicts information about a referenced element by in an XML Schema sequence or choice. Let E be an XML Schema element references a sequence or choice. The mapping information consists of the datatype properties “refElementID”, “minOccurs” and “maxOccurs”.

The datatype property “refElementID” is of type string, has a cardinality of one and has as value the name of XML Schema element E.

The datatype property “minOccurs” is of type string, has a cardinality of one and has as value the minimum number of element E occurrences.

The datatype property “maxOccurs” is of type string, has a cardinality of one and has as value the maximum number of element E occurrences.

The class “ElementRefType” is extended by the class “SeqElementRefType”, which represents information about the usage of an element in a sequence and has in addition to the properties of the “ElementRefType” class, the datatype property “itemPosition”. The latter datatype property is of type integer, has a cardinality of one and represents the position of the element in the sequence.

2.11. SPARQL2XQuery 1.0 Framework

In this section is presented the SPARQL2XQuery 1.0 framework which provides a formal mapping model for the expression of OWL to XML Schema mappings and a generic formal methodology for SPARQL-to-XQuery translation. The SPARQL2XQuery 1.0 framework is the ancestor of the SPARQL2XQuery 2.0, framework which has been developed in to thesis. SPARQL2XQuery introduces an environment where SPARQL queries are automatically translated in XQuery syntax which is used for accessing XML data across the web. The SPARQL2XQuery 1.0 framework has been integrated with the XS2OWL 1.0 framework for automatic mapping discovery, generation and maintenance. The SPARQL2XQuery 1.0 framework is compatible with the Double Bus Architecture (show in Figure 2.47), which was introduced by Tim Berners Lee [TimBus].

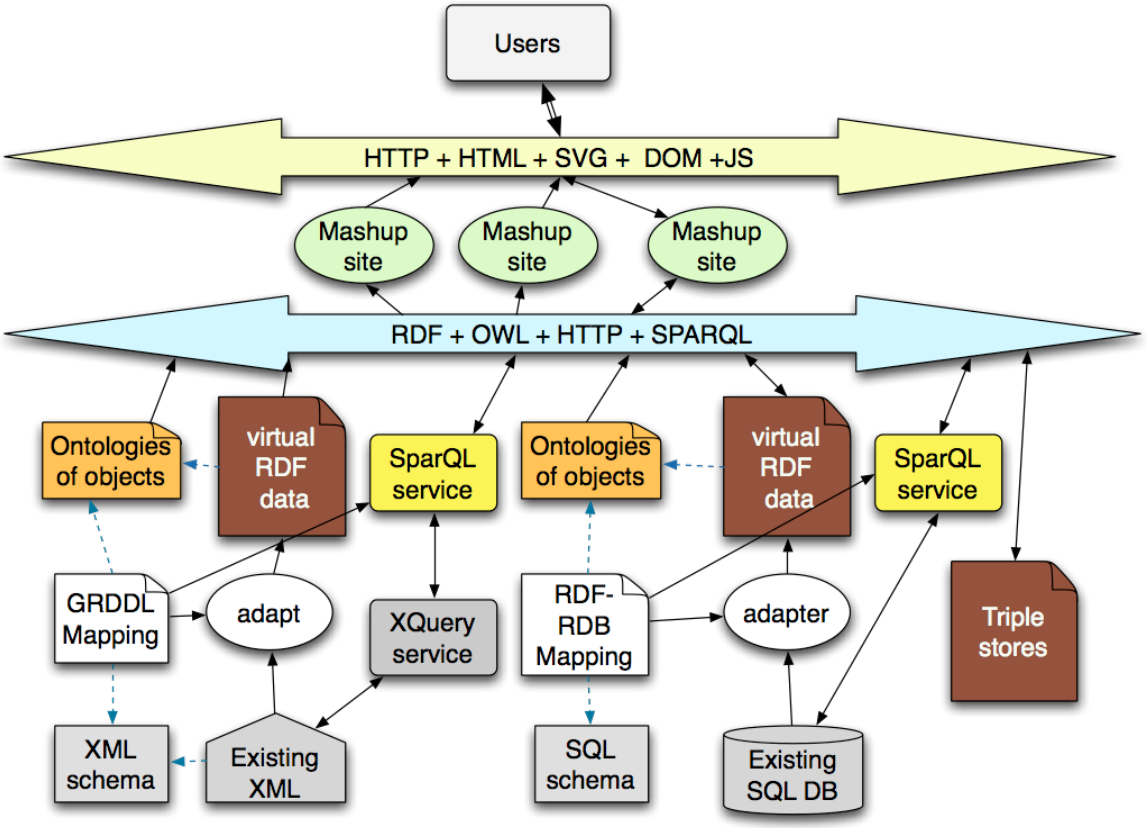

Figure 2.47 Double Bus Architecture

The Double Bus Architecture tries to integrate legacy data sources (Relational and XML) in the Semantic Web, assuming that the Semantic Web users and applications use the SPARQL query language to ask for content from underlying XML data.

The SPARQL2XQuery 1.0 contribution is summarized as follows:

- It introduces a formal mapping model for the expression of OWL – XML Schema mappings that are used for the SPARQL-to-XQuery translation.
- It proposes a generic formal methodology and a set of algorithms that provide a comprehensive SPARQL-to-XQuery translation.
- It integrates the SPARQL2XQuery framework with the XS2OWL framework facilitating (a) the automatic representation of XML Schemas as OWL-DL ontologies and (b) the automatic generation and maintenance of the mappings used in the SPARQL-to-XQuery translation.

In the rest of this section is presented the system architecture in section 2.12.1, the mapping model in section 2.12.2, the GP Normalization in section 2.12.3, the Variable management in

section 2.12.4, the Onto-triple processing in section 2.12.5 and the Graph Pattern translation in section 2.12.6.

2.11.1. System Architecture

This section presents an overview of the SPARQL2XQuery 1.0 framework and outlines the system architecture. The working scenarios involve existing XML data, which follow one or more XML Schemas. Moreover, the SPARQL2XQuery framework supports two different scenarios:

1. **It utilizes an ontology automatically generated by the XS2OWL framework. In that case, the following steps take place:**
   a. Using the XS2OWL framework, the XML Schema is expressed in OWL-DL syntax.
   b. The SPARQL2XQuery framework, taking as input the XML Schema and the generated main ontology, automatically generates and maintains the mappings between them.
   c. The SPARQL queries posed over the ontology are translated to XQuery expressions.
   d. The query results are transformed into the desired SPARQL Query Result XML Format or in RDF format. Thus, the SPARQL2XQuery framework can be utilized as a fundamental component of hybrid Ontology-based integration frameworks, where the schemas of the XML sources are represented as OWL-DL ontologies and these ontologies are further mapped to a global ontology.

2. **It uses an existing ontology. In this case the following steps are take place:**
   a. An existing OWL ontology is manually mapped by a domain expert with the XML Schema.
   b. The SPARQL queries posed over the ontology are translated to XQuery expressions.
   c. The query results are transformed into the desired SPARQL Query Result XML Format or in RDF format.

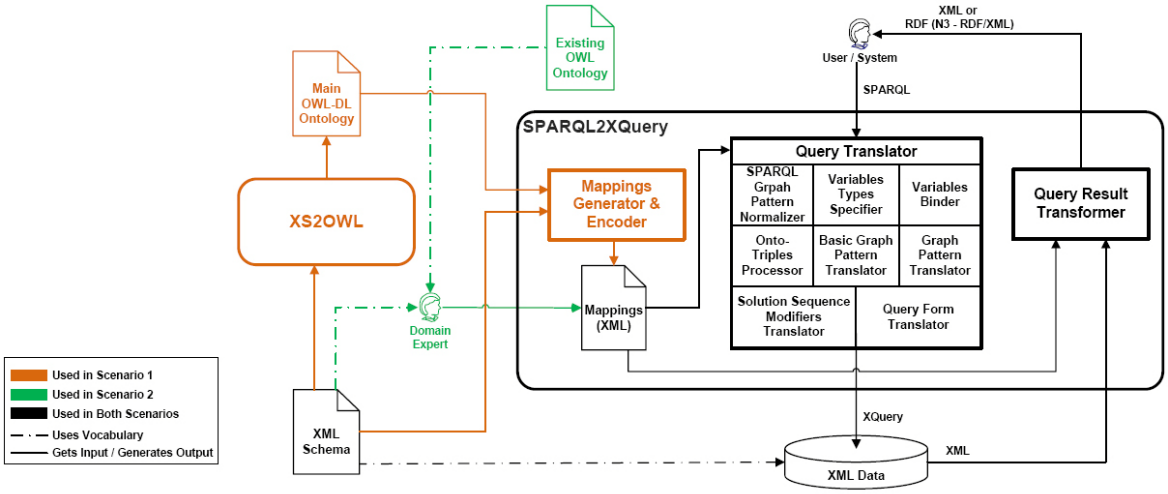

Figure 2.48 SPARQL2XQUERY 1.0 Architecture

2.11.2. The Mapping Model

The translation of SPARQL queries in XQuery syntax require a mapping dictionary, in which have been declared one by one all the relations between the OWL constructs of the ontology used as the knowledge base of the OWL environment and the XML Schema constructs of the XML data existing in the repository and forming the XML environment side of the bridging process. These mappings have to be recorded in a dictionary, which is implemented with an XML document as described in subsection 2.12.2.1, and before that, be analyzed and captured from the ontology and the XML source schema as described in subsection 2.12.2.2. Furthermore, there is a set of XPath operators introduced by SPARQL2XQuery 1.0, which specifies some operations between XPath sets as described in subsection 2.12.2.3.

Representation of Mappings

In the first scenario, the generation and encoding of the mappings are carried out by the Mappings Generator and Encoder component, which takes as input the original XML Schema and the main OWL-DL ontology (generated by the XS2OWL framework). This component parses the input files and obtains the existing correspondences between the XML Schema and the generated ontology using the XS2OWL transformation model of Table 2. The output of the Mappings Generator and Encoder component is an XML document that contains the mappings between all the constructs of the ontology and the XPath sets that address all the corresponding XML instances. In particular, it generates the Class XPath Set XC, the Property XPath Set XPr, the Property Domains XPath Set XPrD and the Property Ranges XPath Set XPrR for all the ontology classes and properties.

Definition 2.21 Mapping

A Mapping in the context of the SPARQL-to-XQuery translation is formally represented as: O≡XS where O is an ontology construct, class or property and XS is an XPath set.

The Figure 2.49 shows the mappings between the Semantic Web and XML Environments. At the Schema level, correspondences between the OWL ontology constructs and the XML Schema constructs exist. At the Data level, the XML data follows the XML Schema and every XML node can be addressed using XPath expressions. Based on OWL and XML Schema correspondences, the ontology constructs are “associated” with the corresponding XPath expressions.

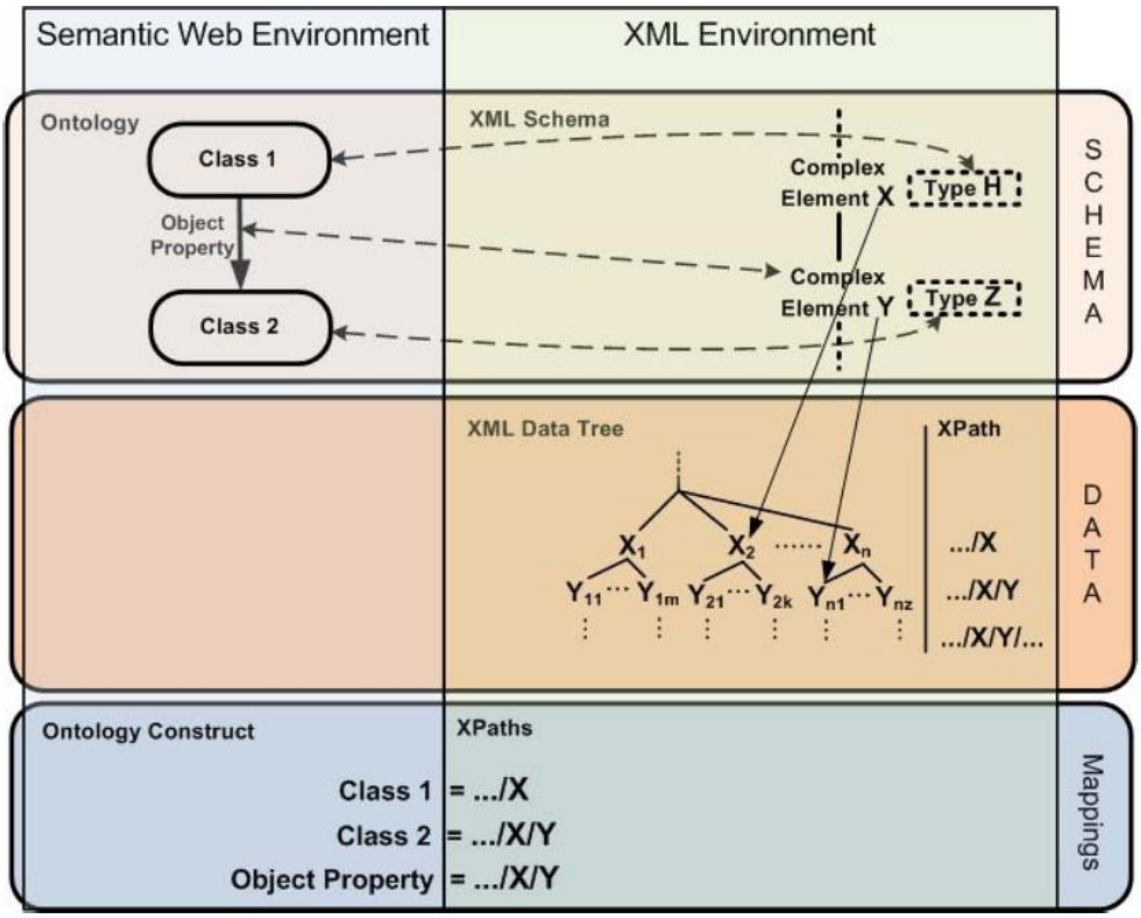

Figure 2.49 Mappings between the Semantic Web and XML Environments

2.11.3. Graph Pattern Normalization

The SPARQL Graph Pattern Normalization activity, which is described here, rewrites the Graph Pattern (*GP*) (Definition 2.8, essentially the Where clause) of the SPARQL query in an equivalent normal form. The SPARQL Graph Pattern normalization is based on the *GP* expression equivalences proved in [PerArGutCom] and on rewriting techniques.

In particular, every *GP* can be transformed in a sequence:

P_1 *UNION* P_2 *UNION* P_3 *UNION...UNION* P_n, where P_i $(1 \le i \le n)$ is a *UF-GP* (Definition 2.19).

The new GP normal form allows for easier and more efficient translation since:

- It contains only a sequence of union-free graph patterns that can be processed independently.
- It contains bigger Basic Graph Patterns (BGP) (Defintion 2.7). These BGPs make more efficient the translation process by reducing the number of the variable bindings as well as the BGP translation processes that are executed. A bigger BPG means also that more joins (i.e., ANDs)

are intrinsically handled by XQuery expressions in the translation of the BGP (by the BGP2XQuery algorithm) and not by solution modifications at a higher level, thus making the translation process more efficient.

2.11.4. Variable Management

As it has been already discussed, a SPARQL query includes within every declaration a set of variables *V*. For each variable in *V,* must be determined the type of the variable according to its position in the triple patterns and its context. Moreover, for every variable in *V*, is explored and determined the set of XPaths with which the variable must be bound in order to make feasible the translation of the SPARQL query in XQuery syntax.

In this section is presented an overview of these activities and how they are implemented in the SPARQL2XQuery 1.0 framework.

Variable type Determination

This activity identifies the type of every variable referenced in a Union-Free Graph Pattern (*see* Definition 2.19) *UF-GP* in order to determine the form of the results and, consequently, the syntax of the Return clause in XQuery. Moreover, the variable types are used by the "Onto-triple processing" (section 4.4.2) and the "Variable Binding" activities (section 4.4.3). Finally, this activity performs consistency checking in variable usage in order to detect possible conflicts (e.g., the same variable name may be used in the definitions of variables of different types in the same *UF-GP*). In such a case, for efficiency reasons the *UF-GP* is not going to be translated, since it is not possible to be matched with any RDF dataset.

Definition 4.7. Variable Types. For the variable type determination, the following variable types are defined:

- The Class Instance Variable Type (*CIVT*), which represents class instance variables.
- The Literal Variable Type (*LVT*), which represents literal value variables.
- The Unknown Variable Type (*UVT*), which represents unknown type variables.
- The Data Type Predicate Variable Type (*DTPVT*), which represents data type predicate variables.
- The Object Predicate Variable Type (*OPVT*), which represents object predicate variables.
- The Unknown Predicate Variable Type (*UPVT*), which represents unknown predicate variables.

Variable binding

The variable binding process aims to declare the bindings between the variables of a BGP and a set of XML paths according to the XML document that contains the data of the XML repository. The

binding of variables with paths results in a dictionary of variables and their corresponding set of paths. This dictionary is essential for the correct and sound execution of the BGP translation process to the equivalent XQuery syntax, as it is necessary to know the possible paths that lead to the instance in the XML document that represents the variable.

The Variable binding process exploits the results of the onto-triple processing (2.12.5). The bindings of variables mentioned in onto-triples are used during the initialization session of the Variable binding process. These bindings are being reffered in this section as initial bindings.

Moreover, the Variable Binding process does not take into consideration any FILTER elements of the BGP, as these elements do not contain anything useful for the binding of variables to XML paths but only restrictions and conditions between variables and values.

2.11.5. Onto-triples

In this section is described the Onto-Triple Processing, i.e. the processing of triples which are composed of constructs that belong to the RDF/S[W3C/RDF(S)] and OWL[W3C/OWL2] vocabularies. This process follows the Variable Type Determination process and its input is a Union-Free graph pattern in order to bind the variables with the XPaths of the xml document that were captured by the onto-triple analysis. This set of bindings is used for the initialization of the Variable Binding process. They are used as initial bindings in our algorithms, as these paths depict the requirements of the xml data according to the ontology semantics that are mentioned in a graph pattern.

An **Onto-triple** is a triple which refers to the ontology structure and/or semantics. Formally, the one and only requirement for a triple pattern to be marked as onto-triple is the existance of at least one construct, among the triple parts, that belongs to the RDF/S and OWL vocabularies.

In some cases the Onto-triple variable can be bound to XPaths without the evaluation of the triples. A list of cases that are supported by the SPARQL2XQuery 1.0 is shown below:

- ?x rdf:type IRI
- ?x rdf:type rdf:Property
- ?x rdf:type owl:ObjectProperty
- ?x rdf:type owl:DatatypeProperty
- ?x rdf:type ?o
- ?x rdf:subClassOf IRI

2.11.6. Graph Pattern Translation

This section describes the Graph Pattern Translation, which translates a graph pattern (GP) into semantically correspondent XQuery expressions. The concept of a GP is defined recursively, thus the Basic Graph Pattern Translation activity translates the basic components of a GP (i.e., BGPs) into semantically correspondent XQuery expressions, which however have to be properly associated in the context of a GP. Thus it applies the SPARQL operators among them using XQuery expressions and functions. These operators are: OPT, AND, UNION and FILTER and are implemented using standard XQuery expressions.

2.11.6.1. Basic Graph Pattern Translation

The Basic Graph Pattern Translation is the translation of the SPARQL Basic Graph Patterns (BGP) to semantically correspondent XQuery expressions, thus allowing the evaluation of a BGP over XML data. This process is based on the results of previous processes, particularly the mappings determination, the variable binding and the variables type determination.

The main and most challenging to translate part of the query, is the *where* clause. The *where* clause must be translated into a format that can be used by the XQuery language afterwards in order to create a sequence of XQuery expressions that articulate the same semantics with the SPARQL expressions. The where clause of the SPARQL query is a graph pattern (Definition 2.8). The simplest case of a Graph Pattern is the Triple Pattern and then the Basic Graph Pattern, which is comprised of triple patterns and filter expressions. This analysis clarifies the target of the needed algorithm, which is the sound and complete translation of a BGP into XQuery expressions that will be afterwards evaluated over XML data.

The core of the Basic Graph Pattern translation is the BGP2XQuery algorithm, which is responsible for the orchestration of the whole activity. The sub-activities of the algorithm include:

- the Subject translation,
- the Predicate translation,
- the Object translation,
- the Filter translation and
- the Building of the Return clause.

These activities are implemented by a set of algorithms that are presented later in this thesis at their latest form. The new algorithms have been modified in order to support the new features that the SPARQ2XQuery 2.0 framework supports.

2.12. Summary

This chapter described the background technologies of both the XML environment and the Semantic Web. Furthermore, as far as the XML Schema and OWL technologies are concerned, Chapter 2 presented the current and the previous version of these W3C technologies in order to clarify the differences that motivated the evolution of the XS2OWL framework to the new XS2OWL

2.0 framework and the evolution of the SPARQL2XQuery framework to the new SPARQL2XQuery 2.0 framework.

Moreover, except from the W3C standards and technologies, chapter 2 presented an overview of the XS2OWL framework in order to provide the readers with the existing context in which the XS2OWL 2.0 framework was developed and to clarify the contribution of the new version to the previous one. Thus, the next chapter (Chapter 3) analyzes the XS2OWL 2.0 framework by desribing the new functionality of the framework.

Finally, Chapter 2 presented an overview of the SPARQL2XQuery framework in order to provide the readers with the existing context in which the SPARQL2XQuery 2.0 framework was developed and to clarify the contribution of the new version to the previous one. Thus, Chapter 3 analyzes the SPARQL2XQuery 2.0 framework by desribing the new functionality of the framework.

3. The XS2OWL 2.0 Framework

The 3rd chapter of this thesis represents the transformation of XML Schemas into OWL 2.0 syntax. The approach of this thesis focuses on capturing the semantics of the XML Schema structures and repreresenting them with OWL 2.0 structures and axioms. The main idea, of the approach presented in this chapter, is to capture the semantics of XML Schema structures and retain any semantics of XML Schema that cannot be represented in OWL 2.0. This is feasible due to an assistant file, which consists of OWL classes individuals that preserve any semantics of XML Schema syntax that are not represented natively by OWL 2.0 axioms. In this way, the XS2OWL 2.0 framework allows to generate a correspodent OWL 2.0 ontology and simultaneously to hold any XML Schema structures irrelevant to OWL 2.0 syntax for further use and exploitation by one that would need the complete semantics of an XML Schema construct that is partially or not represented within the generated OWL 2.0 ontology.

The rest of this chapter describes the architecture of XS2OWL 2.0 framework in section 3.1, the mapping ontology in section 3.2, the Transformation model XS2OWL 2.0 in section 3.3 and a summary in section 3.4.

3.1. Framework Architecture

The XS2OWL 2.0 framework (the name stands for XML Schema-to-OWL) is responsible for the representation of XML Schemas in OWL syntax. In this section is shown in brief how the XS2OWL 2.0 framework, which was developed in the context of this thesis, works. It is used throughout the

section the XML Schema motivating example of Figure 2.12, which describes a sequence of persons with their personal information and relations between them. This XML Schema has been transformed into semantically equivalent OWL syntax using the XS2OWL transforming model as shown in Figure 3.1. The products of this transformation are the main ontology and the OWL2XMLRules mapping ontology. The mapping ontology holds any information provided by the XML Schema that cannot be directly captured in the main ontology due to the expressivity limitations of the OWL 2.0 syntax.

Figure 3.1 The XS2OWL 2.0 Framework

An important difference between XS2OWL 2.0 and XS2OWL 1.0 is the elimination of the additional XML Schema file which was used to save the definitions of the XML Schema Simple Types. This capability is supported automatically in OWL 2.0. A detailed comparison between XS2OWL 1.0 and XS2OWL 2.0 is presented below, in Table 3.1.

	XML Construct	XS2OWL 1.0	XS2OWL 2.0
XML Schema 1.0	Complex Type	✓	✓
	Attribute	✓	✓
	Element	✓	✓
	Attribute	✓	✓
	Annotation	✓	✓
	Sequence	✓	✓
	Choice	✓	✓
	Substitution Group	✓	✓
	Extension	✓	✓
	Simple Type	❖	✓
	Nested Simple Type	❖	✓
	Key	×	✓
	Keyref	×	✓
	Unique	×	✓
	Redefine	×	❖
XML	Error	×	✓

Schema 1.1	SubstitutionGroup 1.1	×	✓
	Alternative	×	❖
	Assert	×	❖
	Override	×	❖

Table 3.1 XS2OWL 1.0 - XS2OWL 2.0 Comparison
(Legend: ✓ supported × not supported ❖ mapping ontology only)

An overview of the XS2OWL 2.0 transformation model is provided in Table 3.2.

XML Schema 1.1 Construct	OWL 2.0 Construct
Complex Type	Class
Simple Datatype	Datatype Definition
Element	(Datatype or Object) Property
Attribute	Datatype Property
Sequence	Unnamed Class – Intersection
Choice	Unnamed Class – Union
Annotation	Comment
Extension, Restriction	subClassOf axiom
Unique (Identity Constraint)	HasKey axiom
Key (Identity Constraint)	HasKey axiom – ExactCardinality axiom
Keyref (Identity Constraint)	Object Property Range
Substitution Group	SubPropertyOf axioms
Alternative	On Mapping Ontology
Assert	On Mapping Ontology
Override, Redefine	On Mapping Ontology
Error	Datatype

Table 3.2 The XS2OWL 2.0 Transformation Model

3.2. The OWL2XMLRules Mapping Ontology

In this section is described the extensions made in the context of XS2OWL 2.0 framework in the OWL2XMLRules mapping ontology. As already been mentioned, the OWL2XMLRules ontology is generated for each XML Schema and allows encoding all the knowledge needed to transform the individuals generated or added later on to the main ontology back to XML syntax valid according to the original XML Schema. The mapping ontology keeps information that is not usable by the Semantic Web tools, but can be of use in other applications like, for example, the transformation of RDF data structured according to the main ontology in XML syntax compliant with the original XML Schemas.

In the context of XS2OWL 2.0 three additional classes are introduced in the OWL2XMLRules mapping ontology which are described in the following subsections: The “OverrideInfoType” class is presented in subsection 3.2.1, the “AlternativeInfoType” class is presented in subsection 3.2.2, the “AssertInfoType” class is presented in subsection 3.2.3, the “SimpleTypeInfoType” class is presented in subsection 3.2.4 and the “IdentityConstraintInfoType” class is presented in subsection 3.2.5. It is necessary to mention that except from these 3 classes, which appear only in the XS2OWL 2.0 framework, there are still the five classes of the mapping ontology of XS2OWL that were described in section 2.11.8. These classes have been modified in order to support the new features introduced by XML Schema 1.1.

3.2.1. Class OverrideInfoType

The “OverrideInfoType” class captures information about the XML Schema “override” and “redefine” elements. As mentioned in subsection 2.3.3, redefine is deprecated in XML Schema 1.1, however XS2OWL 2.0 is designed in order to serve both versions of XML Schema, XML Schema 1.0[W3C/XSD1] and XML Schema 1.1[W3C/XSD1.1a]. “OverrideInfoType” has the datatype properties “overrideType” and “schemaLocation” as well as a rdf:XMLLiteral construct.

The datatype property “overrideType” is of type string, has a cardinality of one and has as value the type of the element that is represented. It has one of the following values:

- “Redefine” if the represented construct is a redefine element.
- “Override” if the represented construct is an override element.

The datatype property “schemaLocation” is of type string, has a cardinality of one and has as value the IRI of the target schema of the override/redefine element.

The rdf:XMLLiteral has as value the exact structure of the element as it appears in the XML Schema definition. Unfortunately, OWL 2.0 does not support the functionality and semantics of the override element of XML Schema; thus, it is represented entirely in the mapping ontology.

In the following example of Figure 3.2 is presented the corresponding individual of an override element that replaces the “ValidAgeType” simple type defined in the “Persons.xsd” XML Schema with the XML structure specified within the rdf:XMLLiteral.

```
<ox:OverrideInfoType>
      <ox:overrideType>override</ox:overrideType>
      <ox:schemaLocation>Persons.xsd</ox:schemaLocation>
      <rdf:XMLLiteral>
            <xs:simpleType name="validAgeType" >
                  <xs:annotation>
                        <xs:documentation>
                        Overrided Simple Type.
                        </xs:documentation>
                  </xs:annotation>
                  <xs:restriction base="xs:integer">
                        <xs:minInclusive value="23"/>
                        <xs:maxInclusive value="55"/>
                  </xs:restriction>
            </xs:simpleType>
      </rdf:XMLLiteral>
</ox:OverrideInfoType>
```

Figure 3.2 The OverrideInfoType individual

3.2.2. Class AlternativeInfoType

The “AlternativeInfoType” class captures information about the XML Schema “alternative” element. As mentioned in subsection 2.3.2, the instances of the Alternative type provide associations between boolean conditions (as XPath 2.0 expressions) and Type Definitions and they are used in conditional type assignment. “AlternativeInfoType” has the datatype properties “PropertyID”, “typeID”, “assertExpr” and “XDefaultNamespace”.

The datatype property “PropertyID” is of type string, has a cardinality of one and has as value the name of the element, in which the alternative element is defined.

The datatype property “assertExpr” is of type string, has a cardinality of one and has as value an XPath expression.

The datatype property “typeID” is of type string, has a cardinality of one and has as value the name of the type that is assigned to the related element, in which the alternative element is defined, if the assert expression of the alternative element is true.

The datatype property “XDefaultNamespace” is of type string, has a cardinality of zero or one and has as value the URI of the Namespace of the element.

In the following example of Figure 3.3 is presented the corresponding individual of the alternative element that was described in Figure 2.7.

```
<ox:AlternativeInfoType>
      <ox:PropertyID> Person_PersonType</ox:PropertyID>
      <ox:typeID> EmployeeType</ox:typeID>
      <ox:assertExpr> @kind eq 'employee'</ox:assertExpr>
</ox:AlternativeInfoType>
```

Figure 3.3 The AlternativeInfoType individual

3.2.3. Class AssertInfoType

The “AssertInfoType” class captures information about the XML Schema “assert” element. As mentioned in subsection 2.3.2, the instances of the Assert type element consist of XPath 2.0 expression, which is evaluated over XML data and instances of complex types. In case an XPath is not satisfied by the XML data, the XML Schema Validator marks the XML Document as invalid. “AlternativeInfoType” has the datatype properties “ComplexTypeID”, “assertExpr” and “XDefaultNamespace”.

The datatype property “assertExpr” is of type string, has a cardinality of one and has as value an XPath expression.

The datatype property “ComplexTypeID” is of type string, has a cardinality of one and has as value the name of the complex type, in which the assert element is defined.

The datatype property “XDefaultNamespace” is of type string, has a cardinality of zero or one and has as value the URI of the Namespace of the element.

In the following example of Figure 3.4 is presented the corresponding individual of the assert element that is defined within the unnamed complex type declared in the “intRange” element as shown in Figure 2.6. In particular, the expression of the “assertExpr” element specifies that the “max” attribute of the complex type is less or equal to 6.

```
<ox:AssertInfoType>
      <ox:ComplexTypeID> NS_intRange_UNType</ox:ComplexTypeID>
      <ox:assertExpr> @max le 6 </ox:assertExpr>
</ox:AssertInfoType>
```

Figure 3.4 The AssertInfoType individual

3.2.4. Class SimpleTypeInfoType

The “SimpleTypeInfoType” class, which is described in this subsection, captures information about the XML Schema simple types. This information has two aspects: a) Information about mapping XML Schema simple types with OWL-DL datatypes; and b) Information about XML Schema simple types that cannot be directly expressed in OWL syntax. A detailed description of the information encoded in the individuals the “SimpleTypeInfoType” class, is presented below.

Information about mapping XML Schema simple types with OWL datatypes. A “SimpleTypeInfoType” individual that represents an XML Schema type T, has the datatype properties “typeID” and “datatypeID”.

The datatype property “typeID” has cardinality of zero or one and represents the name of XML Schema type T.

The datatype property “datatypeID”, with cardinality value one, has as value the ID of the OWL datatype that represents XML Schema type T in main ontology.

Information about XML Schema simple types that cannot be directly expressed in OWL syntax. The information about XML Schema simple types that cannot be represented by OWL constructs includes the datatype property “final”, the datatype property “assert” and the datatype property “definition_type”.

The datatype property “final” is of Boolean type and has a cardinality of zero or one. It represents the value of the “final” attribute of the XML Schema simple type T, which specifies if T is final and thus no further type derivations are possible. If the datatype property “final” is absent, the implied default value is “false”.

The datatype property “assert” is of type string and has no cardinality restrictions. It allows the representation of information about assertion expressions on the XML Schema simple type T. It is used only if the XML Schema simple type T has a restriction element that contains assertions.

The datatype property “definition_type” is of string type and has cardinality of one. It represents the type of the XML Schema simple type T. It has one of the following values:

- “restriction” if T is defined as a type restriction.
- “union” if T is defined based on a union of elements.
- “list” if T is defined as a list.

In the following example of Figure 3.5 is presented the XML Schema information about the “ValidAgeType” simple type that cannot be directly expressed in OWL syntax. The “ValidAgeType” is declared as final, which specifies that no further type derivations are possible.

```
<ox:SimpleTypeInfoType rdf:ID="ValidAgeType_si">
     <ox:datatypeID>ValidAgeType</ox:datatypeID>
     <ox:typeID>ValidAgeType</ox:typeID>
     <ox:final>true</ox:final>
     <ox:definitionType>restriction</ox:definitionType>
</ox:SimpleTypeInfoType>
```

Figure 3.5 The SimpleTypeInfoType individual

3.2.5. Class IdentityConstraintInfoType

In this subsection is described the “IdentityConstraintInfoType” class, which captures information about an XML Schema identity constraint element IC. The “IdentityConstraintInfoType” class has the datatype properties “classID”, “ICName”, “RefersTo”, “HostElementID”, “selectorPath” and “constraintType”.

The datatype property “classID” is of type string and has a cardinality of zero or one. In case of the xsd:keyref identity constraint, this property is missing. In case of the key and unique identity constraints, the property represents the name of the OWL class that contains IC in main ontology.

The datatype property “ICName” is of type string and has a cardinality of one. “ICName” captures the name of IC as it appears in the `name` attribute of the xml schema construct.

The datatype property “RefersTo” is of type string and has a cardinality of zero or one. “RefersTo” captures the name of the key, which is referred by the keyref constraint via the `refer` attribute of the xml schema keyref construct. In the case of the key and unique constraints this datatype property is missing from the definition of the IC instance in the mapping ontology.

The datatype property “HostElementID” is of type string, has a cardinality of one and represents the name of the element under which IC is declared in the xml schema.

The datatype property “selectorPath” is of type string, has a cardinality of one and represents the selector XPath expression of IC.

The datatype property “constraintType” is of type string, has a cardinality of one and represents the type of IC. It has values:

- “unique” if IC is a unique element.
- “key” if IC is a key element.
- “keyref” if IC is a keyref element.

Moreover, the “IdentityConstraintInfoType” containts one individual of the “ICProperties” class. The “ICProperties” class contains one or more individuals of the “ICPropertyType” class. The “ICPropertyType” class has the datatype properties “Property” and “fieldPath”.

The datatype property “fieldPath” is of type string, has a cardinality of one and represents the field XPath expression of IC.

The datatype property “Property” is of type string, has a cardinality of one and represents the correspondent object or datatype property of the XPath expression specified in “fieldPath”.

In the following example of Figure 3.6 is presented the “UniquePerson” unique identity constraint, which is declared within the “Persons” element and specifies that the combination of the “Name” and the “@birthday” of a “Person” element, declared under the “Persons” element, must be unique.

```
<ox:IdentityConstraintInfoType rdf:ID="PersonType_IC">
   <ox:classID> PersonType </ox:classID>
   <ox:ICName> UniquePerson </ox:ICName>
   <ox:HostELementID> Persons </ox:HostELementID>
   <ox:selectorPath> Person </ox:selectorPath>
   <ox:ICProperties>
      <ox:ICPropertyType>
         <ox:Property> Name_NameType </ox:Property>
         <ox:fieldPath> Name </ox:fieldPath>
      </ox:ICPropertyType>
      <ox:ICPropertyType>
         <ox:Property> DateOfBirth_DateType </ox:Property>
         <ox:fieldPath> Age/@birthday </ox:fieldPath>
      </ox:ICPropertyType>
   </ox:ICProperties>
   <ox:constraintType> unique </ox:constraintType>
</ox:IdentityConstraintInfoType>
```

Figure 3.6 The IdentityConstraintInfoType individual

3.3. The XS2OWL 2.0 Transformation Model

This section describes the XS2OWL 2.0 Tranformation Model, which is responsible for the representation of XML Schemas OWL syntax. As it has been already mentioned in section 3.1, this transformation process generates two ontologies based on XS2OWL 2.0 transformation model: a) A main ontology which represents the XML Schema constructs using OWL constructs and b) A mapping ontology that associates the names of the XML Schema constructs to the IDs of the equivalent main ontology constructs and captures the information that cannot be described in the main ontology due to the limitations of the OWL 2.0 sytnax.

In subsection 3.3.1 is described the representation of the XML Schema elements, in subsection 3.3.2 is described the representation of the XML Schema simple types, in subsection 3.3.3 is described the representation of the XML Schema identity constraints, in subsection 3.3.4 is described the representation of the XML Schema override and redefine elements, in subsection 3.3.5 is described the representation of the XML Schema alternative elements and in subsection 3.3.6 is described the representation of the XML Schema assert elements.

These XML Schema constructs either are treated in a different way from XS2OWL 1.0 and XS2OWL 2.0 or they are not taken into account in the XS2OWL 1.0 transformation procedure. In order to save space, the analysis presented below does not include the subsections of the XS2OWL 2.0 transformation model that are identical with the XS2OWL 1.0 transformation model. Thus, the representation of the XML Schema Complex Types, Attributes, Groups, Model groups and References are not included here.

3.3.1. XML Schema Element Representation

This section describes the XML Schema Element representation in the main ontology. This part of the transformation process has not changed from XS2OWL 1.0, except for the difference in the declaration of the “substitutionGroup” attribute. In order to save space, it is not included the detailed description of the XML Schema element representation, but only the representation of the “substitutionGroup” attribute.

The XML Schema 1.1 specification changed the syntax of the `substitutionGroup` attribute. The XML Schema 1.0 allowed an element to be substitutable from only one element. XML Schema 1.1 permits an element to be substitutable for multiple elements. This is similar to multiple inheritance.

The XS2OWL 2.0 Transformation Model supports this new feature of XML Schema 1.1. In order to achieve this, the list of names that appear in `substitutionGroup` attribute are tokenized and then treated in the same way that was individually treated in XS2OWL 1.0. Each member of `substitutionGroup` corresponds to a super property of the current property p. This relation is represented by the `rdfs:subPropertyOf` construct in the main ontology for every element of the `substitutionGroup`. For example the XML elements of Figure 3.7 are represented by the OWL syntax shown in Figure 3.8.

```
<xs:element name="Person" type="PersonType"/>
<xs:element name="Employee" type="EmployeeType"/>
<xs:element name="staff" substitutionGroup="Person Employee"/>
```

Figure 3.7 XML elements compliant with the Persons schema

```
<owl:ObjectProperty rdf:ID="staff__NS_staff_UNType">
      <rdfs:subPropertyOf rdf:resource="#Person__PersonType"/>
      <rdfs:subPropertyOf rdf:resource="#Employee__EmployeeType"/>
      <rdfs:label>staff</rdfs:label>
</owl:ObjectProperty>
```

Figure 3.8 OWL declaration of an XML Schema element with substitutionGroup in main ontology

3.3.2. XML Schema Simple Type representation

In this section, is described how XS2OWL 2.0 treats the simple type definitions. Simple types can be either built in the XML Schema language or defined by an XML Schema author. All the simple types created by a schema author are derived datatypes. The built in XML Schema datatypes include both primitive datatypes and derived datatypes.

The previous version of the XS2OWL transformation model used an external XML Schema file to store the simple type definitions. The datatypes represented in this file were used in XS2OWL 1.0 main ontology by referencing them. OWL 1.0 does not allow the definition of new datatypes. As mentioned in section 2.9, the new recommendation of OWL, OWL 2.0 permits the definition of new datatypes through the axiom DatatypeDefintion. This new feature motivated us to include the Simple Type definitions in the main ontology and the mapping ontology without storing them separately in an XML Schema.

XML Schema allows the schema authors to create new simple types by restriction, union or as lists. These three cases are presented in the following subsections.

3.3.2.1. XML Schema Simple Type representation derived by Restriction

In this subsection is described the representation of XML Schema Simple Types derived by restriction from existing types. The base type is specified in the `base` attribute of the `restriction` element or from the `simpleType` among the children of `restriction`. The restriction element has either a `base` attribute or a `simpleType` element, but not both of them.

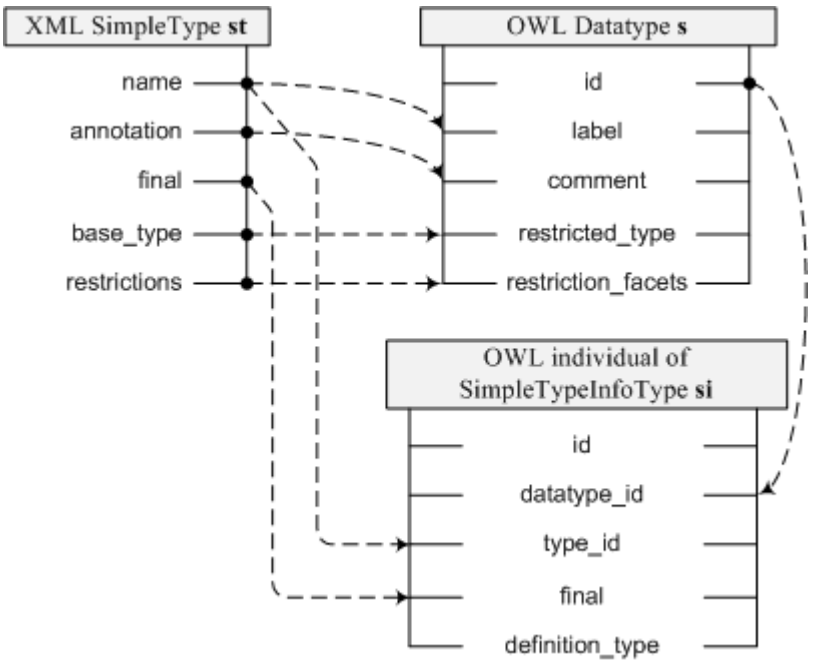

Figure 3.9 OWL representation of the XML Schema Simple Types derived by Restriction

Let `st` be an XML Schema Simple Type that is described by expression (3.1), where:

st (name, annot, final, base, restrictions) (3.1)

- `name` is the value of the "`name`" attribute of the XML Schema simple type `st`, which represents its name.
- `annot` is an optional annotation of the XML Schema simple type `st`, which represents the value of its optional "`annotation`" element.
- `final` is the value of the optional "`final`" attribute of the XML Schema simple type `st`. It specifies the possibility of extending type `st`.
- `base` is the value of the optional "`base`" attribute of the XML Schema simple type `st`. It specifies the type being restricted by type `st`.
- `restrictions` is the set of the restriction expressions declared in the XML Schema simple type `st`.

Let s be the OWL datatype that represents the XML Schema simple type st in the main ontology. The datatype s is described by the following expression (3.2), where:

s (id, label, comment, restricted_type, restriction_facets) (3.2)

- id is the identity of the datatype s, which is represented by rdf:ID and has as value:
  - The value of name, if st is a top level type.
  - If st is an unnamed type nested in element e, the value of concatenate(ct_name, '_', element_name, '_UNType'), where:
    - The algorithm concatenate() takes as input a list of strings and returns their concatenation.
    - ct_name has the value:
      - If the element e is nested in a complex type ct, the value of the "name" attribute of type ct.
      - If e is a top level element, the value "NS".

- label is the label of s, which has as value name and is implemented using the OWL construct rdfs:label.

- comment is an optional comment for s, has the value of annot and is implemented using the OWL construct rdfs:comment.

- restricted_type is the OWL user defined datatype that corresponds to the simple type which is being restricted by the restriction expressions set in order to define the range of the new simple type st. The restricted_type has the value of base and is implemented by the OWL construct owl:onDatatype.

- restriction_facets comprises of the restriction expressions that apply on an existing datatype in order to specify the range of the new datatype st. It is worthwhile to mention that the restriction expressions of OWL 2.0 syntax are inherited from the XML Schema 1.1 namespace. Thus, the restriction_facets has the values of restrictions and are implemented with the OWL construct owl:withRestrictions.

Let si be an individual of the "SimpleTypeInfoType" OWL class. It is described by expression (3.3), where:

si (id, datatype_id, type_id, final) (3.3)

- id is the identity of si, has the value of name and is represented by rdf:ID.

- datatype_id is the identity of the OWL datatype s, which represents simple type st in the main ontology. The datatype_id is represented by the datatype property "datatypeID".

- `type_id` is the name of the XML Schema simple type `st`, has the value of `name` and is represented by the datatype property “typeID”.

- `final` specifies if `st` may be extended. The `final` construct is represented by the datatype property “final”.

The following figures present a sample of an XML Schema (Figure 3.10), which defines a simple type and the results of the transformation process respectively in the main ontology (Figure 3.11) and in the mapping ontology (Figure 3.12).

```
<xs:simpleType name="ValidAgeType">
      <xs:restriction base="xs:float">
            <xs:minInclusive value="0.0"/>
            <xs:maxInclusive value="150.0"/>
      </xs:restriction>
</xs:simpleType>
```

Figure 3.10 XML Schema Simple Type derived by restriction

```
<rdfs:Datatype rdf:ID="ValidAgeType">
<owl:equivalentClass>
   <rdfs:Datatype>
     <owl:onDatatype rdf:resource="&xsd;#float"/>
        <owl:withRestrictions rdf:parseType="Collection">
          <rdf:Description>
             <xsd:maxInclusive
               rdf:datatype="&xsd;#float">150.0</xsd:maxInclusive>
          </rdf:Description>
          <rdf:Description>
             <xsd:minInclusive
                 rdf:datatype="&xsd;#float">0.0</xsd:minInclusive>
          </rdf:Description>
        </owl:withRestrictions>
    </rdfs:Datatype>
</owl:equivalentClass>
</rdfs:Datatype>
```

Figure 3.11 OWL representation of an XML Schema Simple type defined using restriction in the main ontology

```
<ox:SimpleTypeInfoType rdf:ID="ValidAgeType_si">
     <ox:datatypeID>validAgeType</ox:datatypeID>
     <ox:typeID>validAgeType</ox:typeID>
     <ox:definitionType>restriction</ox:definitionType>
</ox:SimpleTypeInfoType>
```

Figure 3.12 Individual of type "SimpleTypeInfoType" representing in the mapping ontology an XML Schema Simple type defined using restriction

The Figure 3.11 presents the OWL 2 syntax of the new Datatype in the main ontology according to the rules that were described through the representation analysis of the XML Schema Simple Type with Restriction. The next subsection describes the transformation rules that are applied on the XML Schema Simple Type constructs which are comprised of unions.

3.3.2.2. Representation of XML Schema Simple Types defined using Union

In this subsection is described the definition of XML Schema Simple Type defined as the union of existing types. The union members are specified in the `memberTypes` attribute of the `union` element. Alternatively, the union members may be specified using nested `simpleType` elements defined within the `union` element. Union has either a `memberTypes` attribute or a set of `simpleType` elements, but these cases are mutually exclusive.

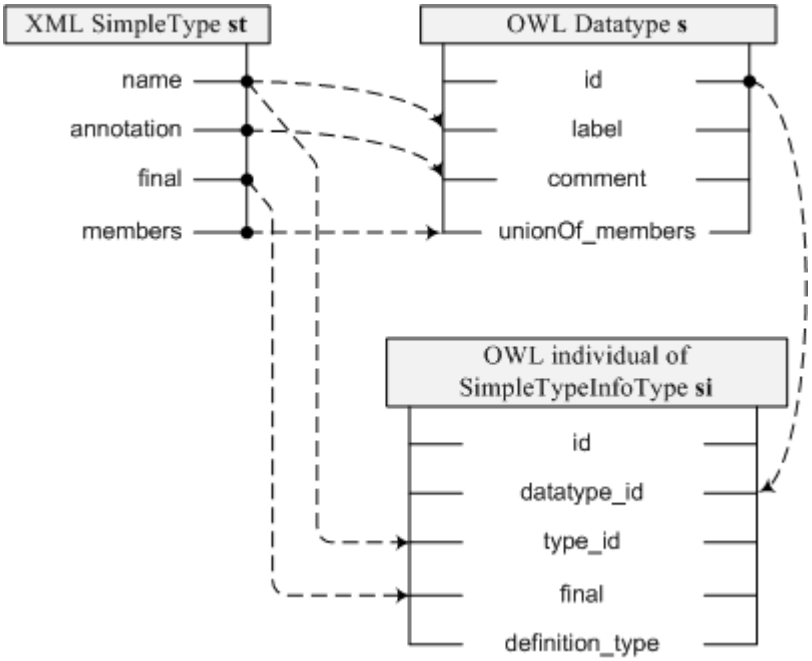

Figure 3.13 OWL representation of XML Schema Simple Types with Union

Let st be the XML Schema Simple Type that is described in expression (3.4), where:

st (name, annot, final, union_members) (3.4)

- name is the value of the "name" attribute of st, which represents its name.
- annot is an optional annotation of st, which has the value of its optional "annotation" element.
- final is the value of the optional "final" attribute of st. It specifies the possibility of extending type st.
- union_members is the value of the optional "memberTypes" attribute of st. It specifies the types that contribute to the union forming st. In case that the "memberTypes" attribute is missing, union_members is the set of the simpleType elements defined within the union element.

Let s be the OWL datatype that represents st in the main ontology. The datatype s is described by expression (3.5), where:

s (id, label, comment, unionOf_members) (3.5)

- id is the identity of s, which is represented by rdf:ID and has value:
  - The value of name, if st is a top level type.
  - If st is an unnamed type nested in element e, the value of concatenate(ct_name, '_', element_name, '_UNType'), where:
    - The algorithm concatenate() takes as input a list of strings and returns their concatenation.
    - ct_name has the value:
      - of the "name" attribute of type ct, if the element e is nested in a complex type ct.
      - The value "NS", if e is a top level element.
- label is the label of s, which has value id and is implemented using the OWL construct rdfs:label.
- comment is an optional comment for datatype s, has the value of annot and is implemented by the OWL construct rdfs:comment.

- unionOf_members is the set of the datatypes, which comprise the union of simple types defined in the context of s. It has the value of union_members and is implemented by the OWL construct owl:unionOf.

Let si be an individual of the "SimpleTypeInfoType" OWL class, described by expression (3.6), where:

si (id, datatype_id, type_id, final, definition_type) (3.6)

- id is the identity of si, has the value of name and is represented by rdf:ID.
- datatype_id is the identity of s, which represents the simple type st in the main ontology. The datatype_id is represented by the datatype property "datatypeID".
- type_id is the name of st, has the value name and is represented by the datatype property "typeID".
- final specifies if the simple type st may be extended and it is represented by the datatype property "final".
- definition_type is the kind of the simple type st. It is a string and has the value "union".

The following figures present a sample of an XML Schema (Figure 3.14), which defines a simple type and the results of the transformation process respectively in the main ontology (Figure 3.15) and in the mapping ontology (Figure 3.16).

```
<xs:simpleType name="UnionAgeType">
      <xs:union memberTypes=”xs:string xs:int”/>
</xs:simpleType>
```

Figure 3.14 XML Schema Simple Type derived by union

```
<rdfs:Datatype rdf:ID="UnionAgeType">
     <owl:equivalentClass>
         <rdfs:Datatype>
             <owl:unionOf rdf:parseType="Collection">
                <rdf:Description rdf:about="&xsd;#float"/>
                <rdf:Description rdf:about="&xsd;#int"/>
             </owl:unionOf>
         </rdfs:Datatype>
     </owl:equivalentClass>
</rdfs:Datatype>
```

Figure 3.15 OWL representation of an XML Schema Simple type defined using union in the main ontology

```
<ox:SimpleTypeInfoType rdf:ID="UnionAgeType_si">
     <ox:datatypeID>UnionAgeType</ox:datatypeID>
     <ox:typeID>UnionAgeType</ox:typeID>
     <ox:definitionType>union</ox:definitionType>
</ox:SimpleTypeInfoType>
```

Figure 3.16 Individual of type "SimpleTypeInfoType" representing in the mapping ontology an XML Schema Simple type defined using union

The Figure 3.15 presents the OWL 2 syntax of the new Datatype in the main ontology according to the rules that were described through the representation analysis of the XML Schema Simple Type with Union. The next subsection describes the transformation rules that are applied on the XML Schema Simple Type constructs which define a list of values that follow a specific simple type.

3.3.2.3. Representation of XML Schema Simple Types as Lists

In this subsection is described the representation of the XML Schema Simple Type defined as a list of a certain datatype. Member's type of list is derived from attribute `itemType` of the `list` element. Alternatively, the list members may be specified using one nested `simpleType` element defined within the `list` element. The `list` has either an `itemType` attribute or a nested `simpleType element`, but cannot have both of them.

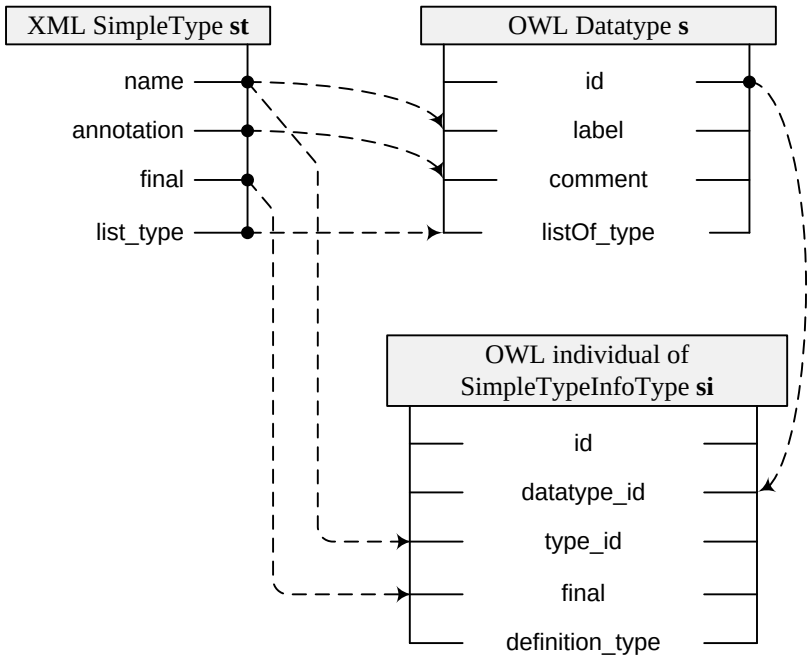

Figure 3.17 OWL representation of XML Schema Simple Types with List

Let `st` be the XML Schema Simple Type is described by expression (3.7), where:

st (name, annot, final, list_type) (3.7)

- `name` is the value of the "`name`" attribute of `st`, which represents its name.
- `annot` is an optional annotation of `st`, which represents the value of its optional "`annotation`" element.
- `final` is the value of the optional "`final`" attribute of `st`. It specifies the possibility of extending type `st`.
- `list_type` is the value of the optional "`itemType`" attribute of `st`. It specifies the type of the list members. If the "`itemType`" attribute is missing, `list_type` is the `simpleType` element defined within the `list` construct.

Let `s` be the OWL datatype that represents `st` in the main ontology. The datatype `s` is described by expression (3.8), where:

s (id, label, comment, listOf_type) (3.8)

- `id` is the identity of `s`, which is represented by rdf:ID and has as value:
  - The value of `name`, if `st` is a top level type.
  - If `st` is an unnamed type nested in element `e`, the value of concatenate(`ct_name, '_', element_name, '_UNType'`), where:
    - The algorithm concatenate() takes as input a list of strings and returns their concatenation.
    - `ct_name` has as value:
      - If the element `e` is nested in a complex type `ct`, the value of "name" attribute of type `ct`.
      - If `e` is a top level element, the value "NS".

- `label` is the label of `s`, which has value `id` and is implemented using the OWL construct `rdfs:label`.

- `comment` is an optional comment for `s`, has the value of `annot` and is implemented by the OWL construct `rdfs:comment`.

- `listOf_type` is the type of the list elements, has value `list_type` and is represented by `owl:onDatatype`.

Let `si` be an individual of the "`SimpleTypeInfoType`" OWL class. It is described by expression (3.9), where:

si (id, datatype_id, type_id, final, definition_type) (3.9)

- `id` is the identity of `si`, has the value of name and is represented by `rdf:ID`.

- `datatype_id` is the identity of the `s` datatype, which represents `st` in the main ontology. The `datatype_id` is represented by the datatype property "datatypeID".

- `type_id` is the name of `st`, has the value of name and is represented by the datatype property "typeID".

- `final` specifies if `st` may be extended and it is represented by the datatype property "final".

- `definition_type` is the kind of `st`. It is a string and has the value "list".

The following figures present a sample of an XML Schema (Figure 3.18), which defines a simple type and the results of the transformation process respectively in the main ontology (Figure 3.19) and in the mapping ontology (Figure 3.20).

```
<xs:simpleType name="ListIntegers">
    <xs:annotation>
      <xs:documentation>
         SimpleType containing a List.
      </xs:documentation>
    </xs:annotation>
    <xs:list itemType="xs:int"/>
</xs:simpleType>
```

Figure 3.18 XML Schema Simple Type defined as list

```
<rdfs:Datatype rdf:ID="ListIntegers">
    <owl:equivalentClass>
        <rdfs:Datatype>
            <owl:onDatatype rdf:resource="&xsd;#int"/>
        </rdfs:Datatype>
    </owl:equivalentClass>
    <rdfs:comment>
         SimpleType containing a List.
    </rdfs:comment>
</rdfs:Datatype>
```

Figure 3.19 OWL declaration in main ontology of XML Schema Simple type with list

```
<ox:SimpleTypeInfoType rdf:ID="ListIntegers_si">
      <ox:datatypeID>ListIntegers</ox:datatypeID>
      <ox:typeID>ListIntegers</ox:typeID>
      <ox:definitionType>list</ox:definitionType>
</ox:SimpleTypeInfoType>
```

Figure 3.20 Individual of type "SimpleTypeInfoType" in mapping ontology of XML Schema Simple type with list

3.3.2.4. Representation of Unnamed XML Schema Simple Types

In this section present the representation of XML Schema Unnamed Simple Types according to the XS2OWL 2.0 transformation model. In the three previous subsections was described how an XML Schema Simple Type is represented in XS2OWL 2.0 according to its type (restriction, union, list). An unnamed Simple Type also belongs to one of those three cases.

Every XML Schema simple type that is nested in an XML construct is unnamed and valid in the scope of that XML construct. This is the reason that it has no name. We cannot refer to this simple type from somewhere else in the XML Schema. Unnamed simple types can be found inside declarations of elements, groups, attribute groups, attributes, simpleType restrictions, simpleType unions, simpleType lists and alternative.

A key point of is the nesting of simple types inside other simple types. This makes it possible for an XML Schema author to declare a simple type which has a nested unnamed simple type, the second has another nested simple type and so forth. This chain is infinite.

For every single simple type, as mentioned in the three previous subsections, is created a Datatype Definition in the main ontology and an individual of the "`SimpleTypeInfoType`" OWL class in the mapping ontology. These requirements and the way nested simple types interact with their named XML construct parents, forced us to implement a recursive algorithm which is responsible for the creation of Datatype Definitions in the main ontology and individuals that represent the nested simple types in the mapping ontology.

The unnamed simple types are assigned a unique ID in the context of the Datatype Definitions in the main ontology. This ID follows a set of naming convention rules.

It is worthwhile to mention that the unnamed simple types defined other XML constructs (except from simple types) are handled in the same way with the unnamed nested types of the simple type declarations. The only difference is the naming convention rule that applies to each case.

In the following is presented the naming convention rules followed in the generation of the IDs of the constructs that represent the unnamed simple types:

- The IDs of the SimpleTypes nested in an element, an attribute, a restriction construct of a named simpleType or a list construct of a named simpleType have as value:

  ```
  concat (parent_name,'_',el_name,'_UNType')
  ```

- The IDs of the SimpleTypes nested in a group or an attribute group have as value:

  ```
  concat (parent_name,'_',el_name,'_',group_name,'_UNType')
  ```

- The IDs of the SimpleTypes nested in a union construct of a named simpleType have as value:

  ```
  concat (parent_name,'_',el_name,'_UNType','_',position)
  ```

- The IDs of the SimpleTypes nested in an alternative element have as value:

```
concat (parent_name,'_',el_name,'_',position'_UNType')
```

If the XML construct, in which a simpleType is nested, is a top level one `parent_name = 'NS'`

In Figure 3.21 is presented an activity diagram, that describies the workflow of the SimpleType representation process. Afterwards, a detailed example is presented showing the way a simple type is represented in the main ontology and the mapping ontology in the case that contains nested unnamed types.

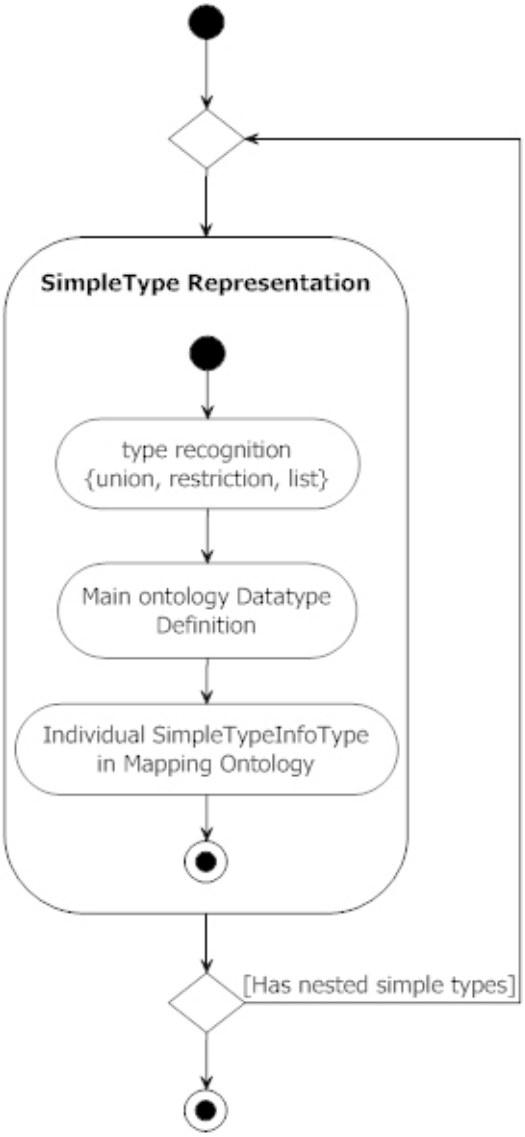

Figure 3.21 Activity diagram of the simpleType Representation process

```
<xs:simpleType name="UnionSimpleTypes" >
      <xs:union>
         <xs:simpleType>
             <xs:restriction base="xs:int">
                  <xs:totalDigits value="3"/>
             </xs:restriction>
         </xs:simpleType>
         <xs:simpleType>
             <xs:restriction>
                  <xs:simpleType>
                        <xs:restriction base="xs:float">
                              <xs:minExclusive value="3.4"/>
                        </xs:restriction>
                  </xs:simpleType>
                  <xs:length value="5"/>
             </xs:restriction>
         </xs:simpleType>
      </xs:union>
</xs:simpleType>
```

Figure 3.22 An XML Schema Simple Type defined as a union definition that includies unnamed simple types

```
<rdfs:Datatype rdf:ID="UnionSimpleTypes">
      <owl:equivalentClass>
      <rdfs:Datatype>
      <owl:unionOf rdf:parseType="Collection">
            <rdf:Description
rdf:about="NS_UnionSimpleTypes_UNType_1"/>
            <rdf:Description
rdf:about="NS_UnionSimpleTypes_UNType_2"/>
      </owl:unionOf>
      </rdfs:Datatype>
      </owl:equivalentClass>
</rdfs:Datatype>
…
<rdfs:Datatype rdf:ID="NS_UnionSimpleTypes_UNType_1">
     <owl:equivalentClass>
         <rdfs:Datatype>
               <owl:onDatatype rdf:resource="&xs;int"/>
               <owl:withRestrictions rdf:parseType="Collection">
                   <rdf:Description>
                       <xs:totalDigits>3</xs:totalDigits>
                   </rdf:Description>
               </owl:withRestrictions>
```

```
         </rdfs:Datatype>
     </owl:equivalentClass>
</rdfs:Datatype>

…
<rdfs:Datatype rdf:ID="NS_UnionSimpleTypes_UNType_2">
   <owl:equivalentClass>
      <rdfs:Datatype>
         <owl:onDatatype
rdf:resource="UnionSimpleTypes_NS_UnionSimpleTypes_UNType_2_UNType"/>
         <owl:withRestrictions rdf:parseType="Collection">
             <rdf:Description>
                  <xs:maxExclusive>21</xs:maxExclusive>
             </rdf:Description>
         </owl:withRestrictions>
      </rdfs:Datatype>
   </owl:equivalentClass>
</rdfs:Datatype>
…
<rdfs:Datatype rdf:ID="UnionSimpleTypes_NS_UnionSimpleTypes_UNType_
      2_UNType">
    <owl:equivalentClass>
      <rdfs:Datatype>
             <owl:onDatatype rdf:resource="&xs;float"/>
             <owl:withRestrictions rdf:parseType="Collection">
                 <rdf:Description>
                      <xs:minExclusive>3.4</xs:minExclusive>
                </rdf:Description>
             </owl:withRestrictions>
      </rdfs:Datatype>
    </owl:equivalentClass>
</rdfs:Datatype>
```

Figure 3.23 OWL representation of an XML Schema Simple type defined as a union that includies unnamed simple types in the main ontology

```
<ox:SimpleTypeInfoType rdf:ID="UnionSimpleTypes_si">
      <ox:classID>UnionSimpleTypes</ox:classID>
      <ox:typeID>UnionSimpleTypes</ox:typeID>
      <ox:definitionType>union</ox:definitionType>
</ox:SimpleTypeInfoType>


<ox:SimpleTypeInfoType rdf:ID="NS_UnionSimpleTypes_UNType_1_si">
      <ox:classID>NS_UnionSimpleTypes_UNType_1</ox:classID>
      <ox:typeID>NS_UnionSimpleTypes_UNType_1</ox:typeID>
      <ox:definitionType>restriction</ox:definitionType>
</ox:SimpleTypeInfoType>

<ox:SimpleTypeInfoType rdf:ID="NS_UnionSimpleTypes_UNType_2_si">
      <ox:classID>NS_UnionSimpleTypes_UNType_2</ox:classID>
      <ox:typeID>NS_UnionSimpleTypes_UNType_2</ox:typeID>
      <ox:definitionType>restriction</ox:definitionType>
</ox:SimpleTypeInfoType>

<ox:SimpleTypeInfoType
      rdf:ID="UnionSimpleTypes_NS_UnionSimpleTypes_UNType_2_UNType_si
      ">
      <ox:classID>
         UnionSimpleTypes_NS_UnionSimpleTypes_UNType_2_UNType
      </ox:classID>
      <ox:typeID>
         UnionSimpleTypes_NS_UnionSimpleTypes_UNType_2_UNType
      </ox:typeID>
      <ox:definitionType>restriction</ox:definitionType>
</ox:SimpleTypeInfoType>
```

Figure 3.24 Individuals of type “SimpleTypeInfoType” in the mapping ontology for the representation of XML Schema Simple type defined as a union that includies unnamed simple types

3.3.3. Representation of XML Schema Identity Constraints

In this section is described the representation of the XML Schema Identity constraints according to the XS2OWL 2.0 transformation model. In the section 2.9 is described the new features of OWL 2.0 including the identity constraints express by the HasKey axiom. This OWL 2.0 feauture allowed us to represent the XML Schema identity constraints in OWL 2.0 syntax.

In the XS2OWL design, the main ontology automatically generated, captures the XML schema semantics and represents them using semantically equivalent OWL constructs.

The XML Schema identity constraints are Unique, Key and Keyref. Unique asserts uniqueness, key asserts uniqueness like unique and it further asserts that all selected content actually has such tuples, and keyref asserts a correspondence with the referenced key. Our implementation aims to capture these semantics by transferring the meaning of uniqueness and the referencing meaning of elements via the key-keyref relation to the structures of the main ontology. It is worthwhile to mention, at this point, some differences between XML Schema and OWL as far as identity constraints are concerned.

The XML Schema identity constraints contain: (a) A selector element, which specifies the XML Schema elements on which the identity constraint is applied; and (b) one or more field elements, where the XML Schema constructs (elements or attributes) that form the constraint value are specified. Both the selector and field elements specify the construct(s) they refer to via their xpath attribute. The xpath attribute uses a set of XPath expressions (subset), which should be evaluated over the XML Schema in order to locate the XML constructs they refer to. On the other hand, the OWL syntax applies the key axiom on some properties (datatype or object) in the scope of a class. This gap is bridged by capturing the semantics of the XML Schema identity constraints and translating them to the semantics of OWL respective axioms.

The activity diagram of Figure 3.25 shows the process of transforming an XML identity constraint to in OWL syntax. The first step includes isolation of the XPaths of the identity constraints. These paths have to be evaluated over the XML Schema document. This is a complex process and is presented in details in the following sections. The evaluation process will lead to the representation of an XML construct by a datatype property or an object property in the main OWL ontology in the XS2OWL 2.0 transformation model. Finally, the necessary OWL identity constraint is created on the properties that represent XML elements on which the XML identity constraintis defined.

This section consists of three subsections. In the first one (3.3.3.1), is described the process of evaluating the XPath of the selector and field elements and discovering the resulting XML tree. In the following subsections (3.3.3.2), (3.3.3.3), (3.3.3.4) present, respectively the representation of the XML Schema identity constraints unique, key and keyref in OWL syntax.

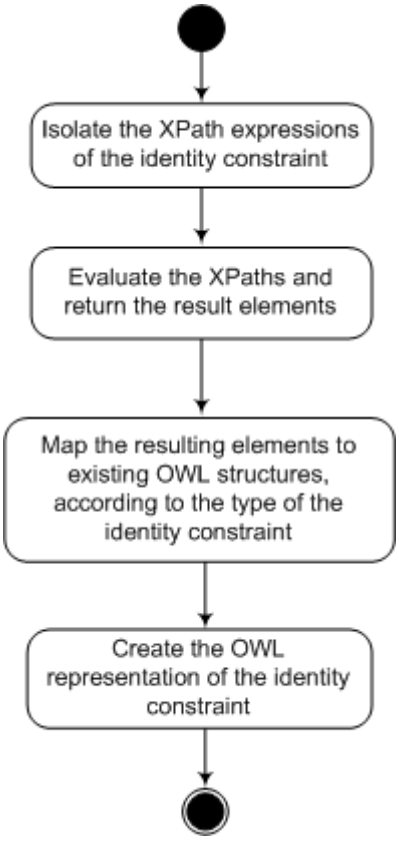

Figure 3.25 Activity diagram of the identity constraint representation process

3.3.3.1. XPath evaluation for the selector and field elements

It has already been mentioned that the XPath expressions of the identity constraints are expressed wing a subset at Xpath language. The selector and field elements have the "`xpath`", attribute, the value of which an XPath expression that follows the syntax of this subset of XPath. This subset includes expressions that contains named and the '/' character. Such an expression indicates a path through an XML document. In addition to the path characters, a field element XPath expression may include the character `at (@)` in order to point to an attribute rather than to an element. Furthermore, a selector element may point to more than one paths by using the OR operator '|' between the path expressions. In conclusion, complex XPath expressions with conditions, loops, predicates, comparisons are prohibited in the `xpath` attribute of the selector and field elements. Only pure paths are allowed.

As shown in Figure 3.25, constaint representation algorithm has to evaluate the XPath expression in order to indicate which elements contribute to the identity constraint. The XSLT processor is capable of evaluating an XPath expression on an XML document. At this point a tricky issue arises. The XPath expression of `xpath` attribute is referring to the XML data document which is generated by the input XML Schema since. It is not written, by an author being aware of the node

hierarchy of the input XML Schema but of the XML node tree of XML data. This fact makes it impossible for XSLT to evaluate the XPath expression and return consistent results. In Figure 3.26 is shown an example of a unique identity constraint and described the problem using figures.

```
<xs:unique name="NameAddrUnique">
    <xs:selector xpath="Person"/>
    <xs:field xpath="Name"/>
    <xs:field xpath="Address/@city"/>
</xs:unique>
```

Figure 3.26 XML Schema Unique identity constraint

The declaration of Figure 3.26 has a unique identity constraint. The selector element points to a node tree, through the xpath attribute, which is the domain of the field elements.The XPath attribute of the field elements is evaluated over this domain. Figure 3.27 presents the XML data document in which the XPath expressions refer.

```
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <ID>
        <IdentityCardID>F123456</IdentityCardID>
    </ID>
    <Name>
        <FirstName>Ioannis</FirstName>
        <Surname>Stavrakantonakis</Surname>
    </Name>
    <Address country="GR" city="Chania" postCode="73100">
        <Street>Hrwwn Polutexneiou</Street>
        <Number>50</Number>
    </Address>
    <Age>24</Age>
</Person>
```

Figure 3.27 XML data document generated from Persons' schema

The concatenated of the XPath expression of the second field of the unique constraint with the selector results in the following path expression:

```
Person/Address/@city
```

This XPath expression points to the attribute “city” with value “Chania” in the example of Figure 3.27. The next step of our analysis will be an attempt to find the equivalent XPath of the corresponding XML Schema. In Figure 3.23 is shown the XML Schema node tree under the Person element and the required actions in order to detect the declaration of attribute `city`. One realizes immediately that the documents of Figure 3.27 and Figure 3.28 have nothing common from an XPath point of view.

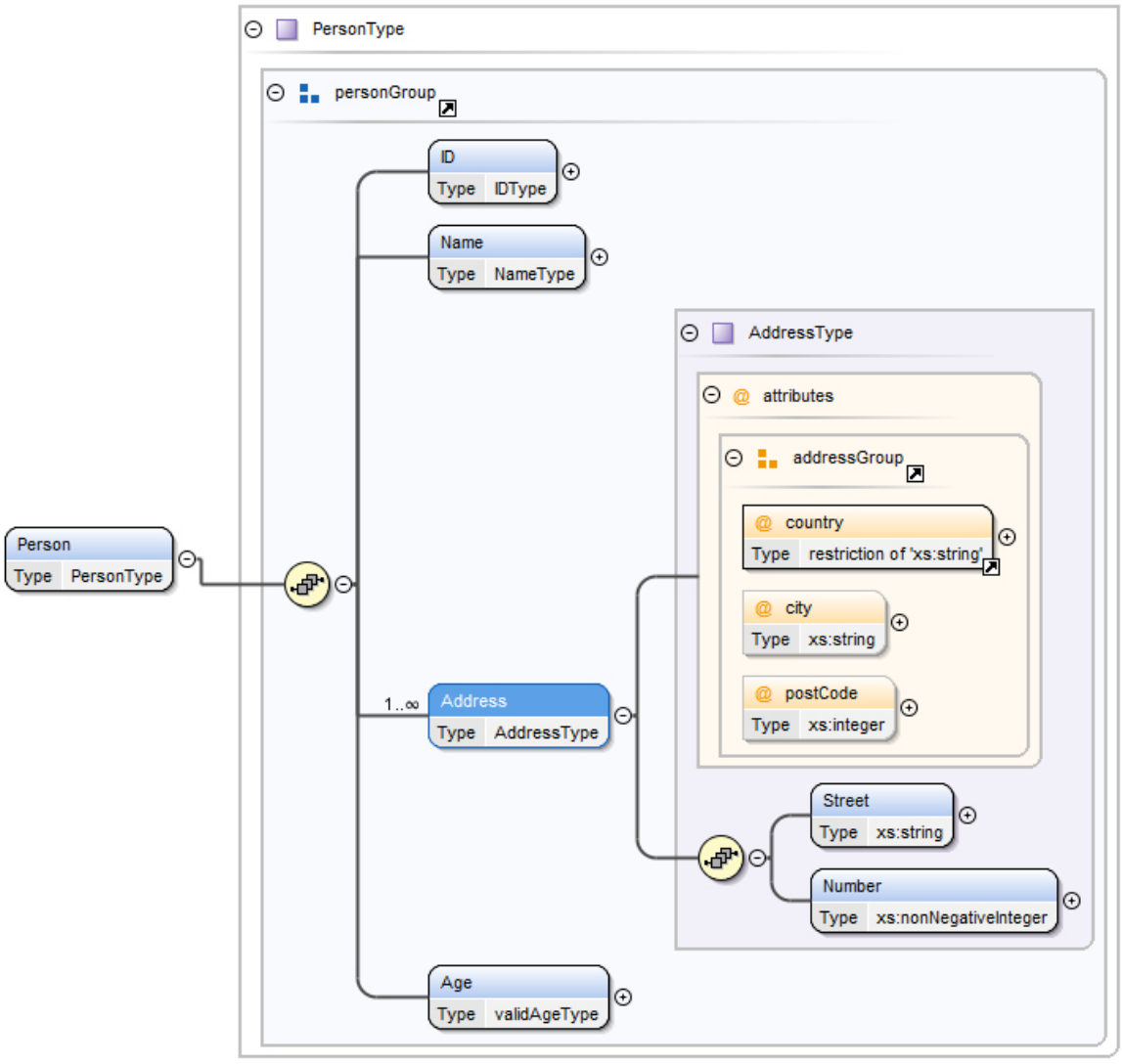

Figure 3.28 XML Persons schema Tree view

Starting the navigation from the Person element the following steps should be taken:

- Locate the type of the Person element, which is “PersonType”.

- Navigate to the “PersonType” declaration. It has no children elements except from a `group` reference to “personGroup” through the attribute `ref`.
- Locate the “Address” element in “personGroup”.
- Navigate to the “AddressType” declaration.
- Locate the attribute group “addressGroup”, which is declared in the `ref` the attribute of the “attributes” element.
- Finally, the “addressGroup” definition includes the desired attribute “city”.

XpathEvaluator algorithm

Since the XPath expressions do not refer to the node hierarchy of the XML Schema but in the node structure of the XML data following it, the “XPathEvaluator” algorithm (see the activity diagram of Figure 3.29) has been developed for XPath expression evaluation. The XSLT variable restrictions, described in section 5.1, force a recursive design and implementation of our algorithm. The inputs of the recursive algorithm include:

- `Xpath,` – which holds the remaining path across the XML Schema.
- `pre_xpath_nodeTree,` – which holds the node tree that has been detected since last call.
- `group_name` – holds the name of a group element if the algorithm comes up against a group element during the execution phase.

The “`group_name`” parameter is called tunnel parameter, because it is passed on by the algorithm during the execution of the “XpathEvaluator” algorithm without having every execution to be aware of the parameter.

Figure 3.29 presents the activity diagram that depicts the idea and the outlines of the “XpathEvaluator” algorithm and afterwards presents the pseudo code of the same algorithm. In an identity constraint *IC(S,F)* the selector S represents the XML Schema element that has a unique combination of values of the constructs specified in the Fields of set F of IC, where F_i is a field of F set, $F_i \in F$.The “XpathEvaluator” is used for the XPath expression evaluation of both selector and fields. For this reason it has been designed in a more abstract way to cover the needs of both cases.

The activity diagram has two input parameters, “xpath” and “Sequence of element nodes”. The former is the XPath expression that should be evaluated over the XML Schema and discover the definition of the element/attribute that refers to. The latter is the node tree, which comprises the initial position within the XML Schema tree. If the XPath Selector of the (S) is evaluated, the “Sequence of element nodes” takes as value the path of the element that hosts the identity constraint (IC). If the XPath of a field evaluated, the “Sequence of element nodes” takes as value the result tree of the Selector XPath evaluation. Finally, if the “XpathEvaluator” function is recursively called inside the algorithm, the “Sequence of element nodes” takes as value the element nodes according to the context element allocation in the schema and the “xpath”

parameter holds the remaining path, of the previous “xpath” input, after the node that represents the context element.

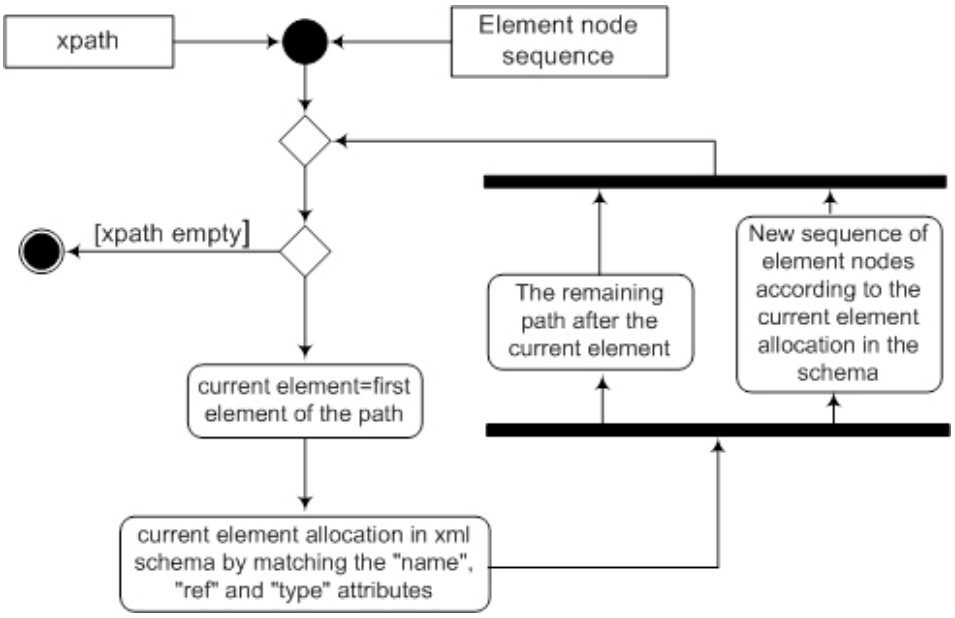

Figure 3.29 Activity diagram of algorithm “XPathEvaluator”

In Table 3.3 is described the application of the algorithm over the example of Figure 3.26. This example show only the workflow for the field “Address/@city”, which takes place after the evaluation of the Xpath Selector’s that results in the path “/Persons/Person”. This path is the pointer to the tree loaded into the “Sequence of nodes” parameter at the initialization.

Algorithm state	XPath	Sequence of nodes[1]
Initial state	Address/@city	/Persons/Person
1st recursion	Address/@city	/PersonType/
2nd recursion	Address/@city	/personGroup/
3rd recursion	Address/@city	/personGroup/Address/
4th recursion	@city	/AddressType/
5th recursion	@city	/AddressType/addressGroup
6th recursion	@city	/addressGroup/
7th recursion	@city	/addressGroup/city
Recursion exits	-	-

Table 3.3 XpathEvaluator example of the workflow

[1] It refers to a path. The path points to the tree of nodes that is passed in for each recursion.

The Xpath Evaluator Algorithm

Input: Selector Xpath, Pre-xpath nodes, Group Name

Output: Sequence of Nodes, Group Name

```
1.  if (xpath != ∅)
2.      Let root_dashes be a boolean
3.      root_dashes ← contains(xpath, './/')
4.      xpath_wo_root ← replace(xpath, './/','')
5.      Let xpath_current_element be the first node of the xpath
6.      Let xpath_remainder be the remaining path after xpath_current_element
7.      Let xpath_current_element_wo_at be the current element without attribute symbol '@'
8.      if (xpath has more than one nodes)
9.          xpath_current_element ← substring-before( xpath_wo_root , '/' )
10.         xpath_remainder ← substring-after( xpath_wo_root , '/')
11.     else
12.         xpath_current_element ← xpath_wo_root
13.         xpath_remainder ← ''
14.     end if
15.     if (xpath_current_element contains '@')
16.         xpath_current_element_wo_at ← replace( xpath_current_element ,'@', ' ' )
17.     else
18.         xpath_current_element_wo_at ← xpath_current_element
19.     end if
20.     if ( root_dashes )
21.         return XPathEvaluator ( xpath_remainder , schema_element.append(//node()
            [./@name = xpath_current_element ]) , group_name)
22.     else if (pre_xpath_nodes != ∅)
23.         temp_result ← pre_xpath_nodes.append(//node()[./@name=xpath_current_element])
24.         if (temp_result != ∅)
25.             return XPathEvaluator (xpath_remainder, pre_xpath_nodes.append
                (//node()[./@name=xpath_current_element ]) , group_name)
26.         else
27.             temp_type ←pre_xpath_nodes.append (/@type)
28.             temp_ref ← pre_xpath_nodes.append (/@ref)
```

```
29.          temp_group_ref ← pre_xpath_nodes.append ( /child::node()[local-name() =
             'group' ]/@ref)
30.          temp_att_group_ref ← pre_xpath_nodes.append ( /child::node()[local-name()=
             'attributeGroup' ]/@ref )
31.          if (schema_element.append(//node()[./@name=temp_type]))
32.              return XPathEvaluator (xpath, schema_element.append ( //node()
                 [./@name=temp_type] ), group_name)
33.          else if (schema_element.append ( //node()[./@name=temp_ref ] ))
34.              return XPathEvaluator (xpath, schema_element.append ( //node()
                 [./@name=temp_ref ] ) , group_name)
35.          else if (schema_element.append ( //node()[./@name=temp_group_ref ]))
36.              return XPathEvaluator (xpath, schema_element.append ( //node()
                 [./@name=temp_group_ref ] ) , temp_group_ref )
37.          else if (schema_element.append (//node()[./@name=temp_att_group_ref ]))
38.              return XPathEvaluator (xpath, schema_element.append ( //node()
                 [./@name=temp_att_group_ref ] ) , temp_att_group_ref)
39.          else
40.              return XPathEvaluator (xpath_remainder, schema_element.append (
                 //node()[./@name=xpath_current_element ] ) , group_name)
41.          end if
42.      end if
43.  else
44.      return (pre_xpath_nodes , group_name)
45.  end if
```

Figure 3.30 The Xpath Evaluator Algorithm

The XPath expression evaluation returns a set of XML Schema constructs, which are represented as (object or datatype) properties in the main ontology. Then, depending on the constraint type, the constraints themselves are expressed in OWL 2.0 syntax, as it is explained in the following paragraphs.

3.3.3.2. Representation of the XML Schema Unique Identity Constraint

In this subsection is described the representation of the XML Schema Unique identity constraint in the main ontology generated by the XS2OWL framework. The members of the uniqueness

constraint are specified in the `field` elements among the children of the `unique element`. The representation in of the main unique constraint ontology requires some preprocessing on the XML Schema. The `selector` is used to define the element on which a uniqueness constraint or reference is applied. Preprocessing evaluates the XPath of the selector in order to indicate the element(s) that serve(s) as the location(s) that forms the context from which the xs:field XPath expressions are resolved. Next the XPath expressions of the field elements are evaluated over the node tree of the selector. The activity diagram of Figure 3.31 describes the entire work flow of the XPath expression evaluating of an IC of unique type. This procedure applies to both the unique and key identity constraints.

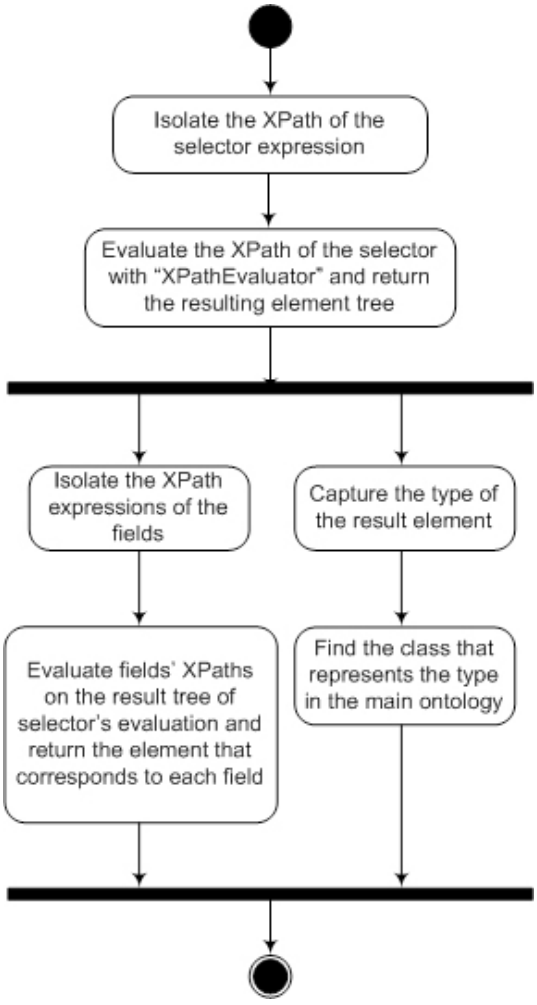

Figure 3.31 Activity diagram of the XPath evaluation process

After analyzing the XPath expression, the resulting element is used to locate the value of the ID of the OWL property that corresponds to this element, in main ontology.

In a unique identity constraint U(S, F) the selector S represents the XML Schema element that has a unique combination of values of the constructs specified in the fields F of U.
For the representation of U in the main ontology, the following actions are performed: (a) The OWL class $C_{SelType}$, which corresponds to the type of S, is located in the main ontology; and (b) A HasKey axiom is defined from the class $C_{SelType}$ on the set P of the properties that represent the XML constructs specified in F (see Figure 3.35).
U is represented in the mapping ontology by an IdentityConstraintInfoType individual (see Figure 3.37).

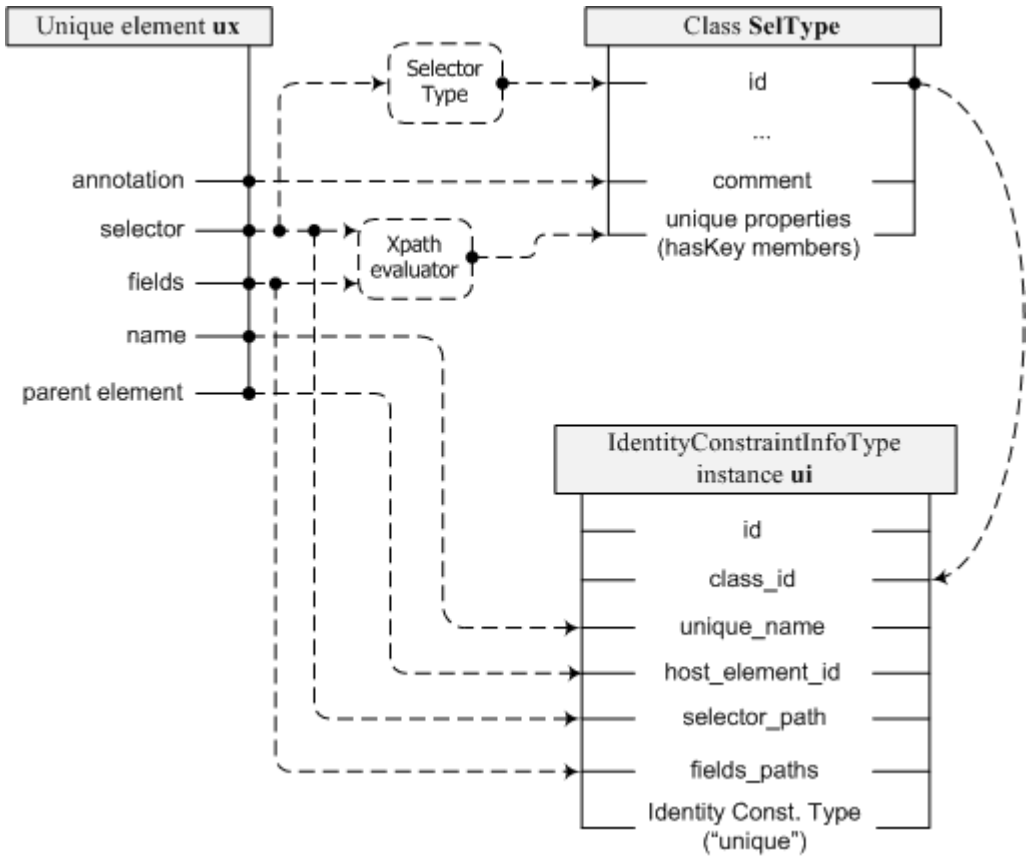

Figure 3.32 OWL representation of an XML Schema Unique Identity constraint

Let ux be an XML Schema Unique constraint that is described by expression (3.10), where:

ux (name, annotation, selector, fields, parent element) (3.10)

- `name` is the value of the "name" attribute of ux, which represents its name.
- `annotation` is an optional annotation of ux, which represents the value of its optional "annotation" element.
- `selector` is the value of the "xpath" attribute of selector element of ux.
- `fields` is the set of the "xpath" attribute of the field elements of ux.
- `parent element` is the name of the element that hosts the ux constraint.

Let `SelType` be the OWL class that represents the XML Schema Type of the selector of ux in the main ontology. This class exists regardless of the unique constraint, since it represents a complex

type in the main ontology. In order to locate the element referred by the path of the selector, the **XPathEvaluator** algorithm (Figure 3.30) is used within the process of mapping an Xpath to an OWL class. This procedure is supported by the **XPathSelector2ClassID** (Figure 3.33).

The Class `SelType` is enriched with the necessary OWL constructs for the representation of the unique `ux` identity constraint, described by expression (3.11), where:

SelType (id, comment, unique_properties) (3.11)

- `id` is the identity of `SelType` and is represented by `rdf:ID`.
- `comment` is an optional comment for `ux`, has the value of `annotation` and is implemented using the OWL construct `rdfs:comment`.
- `unique_properties` is the list of properties indicated in `fields`. These are used to generate the identity constraint using `owl:haskey`.

The mapping ontology keeps information about the name of the constraint, the element that hosts the declaration of the unique constraint and the actual paths of the selector and field elements. Let `ui` be an individual of the "`IdentityConstraintInfoType`" OWL class, representing the additional information needed for a unique constraint. It is described by the expression (3.12), where:

ui (id, class_id, unique_name, host_element_id, selector_path, fields_paths, constraint_type) (3.12)

- `id` is the identity of `ui`, has as value `concatenate(element_name, '_', name, '_', type, '_',ui)` and is represented by `rdf:ID`.
- `class_id` is the identity of the related class `SelType` of `ui`, has as value the `id` of `SelType` and is represented by `the` datatype property `"classID"`.
- `unique_name` is the name of the unique element `ux`, has value the `name` attribute of `ux` and is represented by datatype property `"ICName"`.
- `host_element_id` is the identity of the element that hosts the declaration of `ui` and is represented by `the` datatype property `"HostElementID"`.
- `selector_path` is the XPath expression of the selector of `ux`, has as value `selector` and is represented by the datatype property "selectorPath".
- `fields_paths` are the XPath expressions of the field elements of `ux`, have the values of `fields` and are represented by the datatype property "fieldPath".

- `constraint_type` is the kind of the XML Schema identity constraint, has as value “unique” and is represented by the datatype property “constraintType”.

XPathSelector2ClassID

The XPathSelector2ClassID is responsible for bridging the gap between the constructs that are specified as unique in OWL and XML Schema. As it is already presented, the XML Schema identity constraint specifies an element using the XPath of the selector element. Let the specified element be of the *ct_type* complex type. The *ct_type* has been represented to an OWL class according to the XS2OWL transformation model; let it be the *ct_class* OWL class. The XPathSelector2ClassID is responsible for the estimation of the correspondent *ct_class* OWL class, in which the HasKey axiom will be declared in the context of the XML Schema unique/key identity constraint representation, of the *ct_type* complex type. The XPathSelector2ClassID is based on the naming rules of the XS2OWL transformation model, where the name of an OWL class is the same with the name of the related XML Schema complex type.

The activity diagram of Figure 3.34 presents the XPathSelector2ClassID algorithm. The algorithm has two input parameters, “selector xpath” and “Sequence of element nodes”. The former represents the XPath expression that had to be evaluated over the XML Schema in order to discover the definition of the element that is referred by the selector element of the identity constraint. The latter takes as value the path of the element that hosts the identity constraint (IC). As a result of *XPathEvaluator,* the algorithm retrieves the type of the resulting element and associates it with a pre-existing class of the main ontology.

The XPath Selector 2 Class ID algorithm is presented in Figure 3.33.

Xpath Selector 2 Class ID Algorithm

Input: *selector Xpath*, *pre-xpath nodes*

Output: Selector Type *SelType*

1. *selElement* ← *XPathEvaluator* (*selector xpath*, *pre-xpath nodes*, “”)
2. *SelType* ← *selElement*.**append**(/@type)
3. **return** *SelType*

Figure 3.33 The XPath Selector 2 Class ID Algorithm

The class returned from this process is used to host the HasKey axiom, which is composed by taking in consideration the mapping of the identity constraint fields to the object/datatype properties of the main ontology (see Figure 3.31) as it was described in expression 3.11.

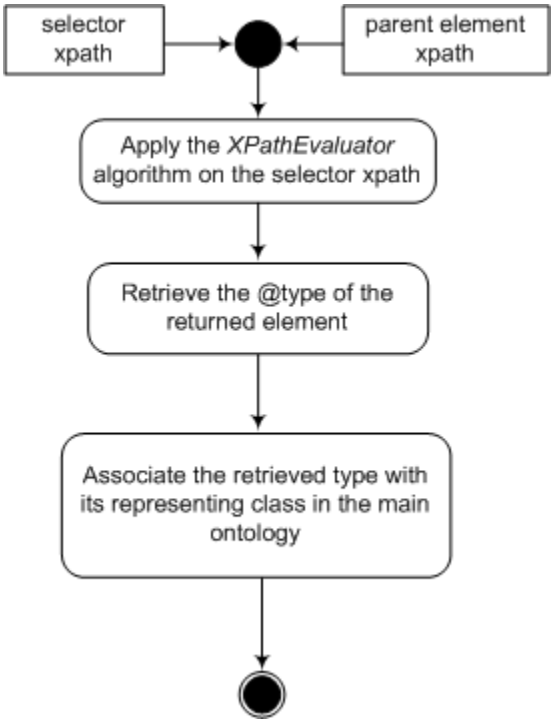

Figure 3.34 Activity diagram of XPathSelector2ClassID algorithm

An example of a unique identity constraint is shown in Figure 3.35, and its representations in the main ontology and the mapping ontology are shown, respectively, in Figure 3.36 and Figure 3.37.

```
<xs:element name="Persons" type="PersonsType">
   <xs:unique name="UniquePerson">
      <xs:selector xpath="Person"/>
      <xs:field xpath="Name"/>
      <xs:field xpath="Address/@city"/>
   </xs:unique>
</xs:element>
```

Figure 3.35 XML Schema Unique identity constraint

The representation of the OWL class in the main ontology, shown in Figure 3.36, presents the additional syntax with which the XPathSelector2ClassID algorithm enriches a pre-existing class in order to capture the semantics of the unique identity constraint.

```
<owl:Class rdf:ID="PersonType">
      ...
   <owl:hasKey rdf:parseType="Collection">
      <rdf:Description rdf:about="#FirstName_xs_string "/>
      <rdf:Description rdf:about="#city_addressGroup__xs_string"/>
   </owl:hasKey>
</owl:Class>
```

Figure 3.36 OWL representation of XML Schema Unique Identity constraint

In the mapping ontology the transformation process creates an individual of IdentityConstraintInfoType to handle the XML Schema constructs that cannot be directly expressed in OWL syntax.

```
<ox:IdentityConstraintInfoType
rdf:ID="Persons_UniquePerson_PersonsType_ui">
      <ox:classID> PersonType </ox:classID>
      <ox:ICName> UniquePerson </ox:ICName>
      <ox:HostELementID> Persons </ox:HostELementID>
      <ox:selectorPath> Person </ox:selectorPath>
      <ox:fieldPath> Name </ox:fieldPath>
      <ox:fieldPath> Address/@city </ox:fieldPath>
      <ox:constraintType> unique </ox:constraintType>
</ox:IdentityConstraintInfoType>
```

Figure 3.37 The individual ui of class IdentityConstraintInfoType in the mapping ontology that represents the XML Schema Unique Identity constraint

3.3.3.3. Representation of the XML Schema Key Identity Constraint

In this subsection is described the representation of XML Schema Key identity constraints according to the XS2OWL transformation model. The key identity constraint K(S,F) has the same semantics with the unique identity constraint, that was presented in subsection 3.3.3.2, with the additional requirement that the XML Schema constructs of the field elements of the key are mandatory. Thus, the representation of the key identity constraint is almost the same with the one

of unique. The only difference is that the additional requirement is satisfied with the definition of an ExactCardinality axiom on the object and datatype properties of the class $C_{SelType}$, which represents the type of S in the main ontology.

As it has been already mentioned, the transformation process is similar to the one described in subsection 3.3.3.2. Thus, the algorithm XPathSelector2ClassID is reused and the activity diagrams describing the transformation process are the same. In Figure 3.38 is presented an overview of the representation of K in the main ontology and the mapping ontology.

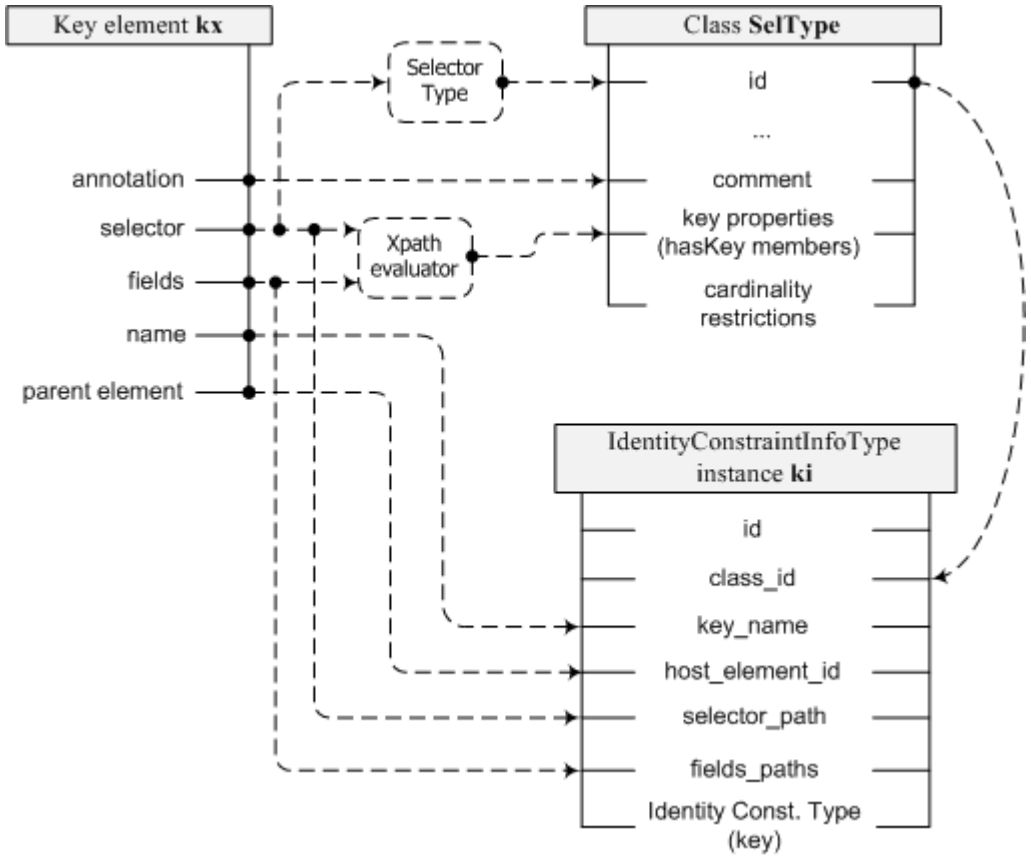

Figure 3.38 OWL representation of an XML Schema Key Identity constraint

Let kx be an XML Schema Key constraint that is described by expression (3.13), where:

kx (name, annotation, selector, fields, parent element) (3.13)

- name is the value of the “name” attribute of kx, which represents its name.

- annotation is an optional annotation of kx, which represents the value of the optional "annotation" element.
- selector is the value of the "xpath" attribute of the selector element of kx.
- fields is the set of the "xpath" attribute of the field elements of kx.
- parent element is the name of the element that hosts the kx constraint.

Let SelType be the OWL class that represents the XML Schema Type of the selector of kx in the main ontology. This class exists regardless of the key constraint, since it represents a complex type in the main ontology. In order to locate the element referred by the path of the selector, the **XPathEvaluator** algorithm (Figure 3.30) is used within the process of mapping Xpaths to OWL classes. This procedure is supported by the **XPathSelector2ClassID** algorithm (Figure 3.33). Moreover, the requirement of the existence of the key properties is satisfied by declaring an exact cardinality of 1 on the object and datatype properties that represent the field elements.

The Class SelType is enriched with the OWL structure for the key kx identity constraint as it is described by the expression (3.14), where:

SelType (id, comment, key_properties, cardinality restrictions) (3.14)

- id is the identity of SelType and is represented by rdf:ID.
- comment is an optional comment for kx, has the value of annotation and is implemented by the OWL construct rdfs:comment.
- key_properties is the list of properties indicated in fields. These are used to generate the identity constraint using owl:haskey.
- cardinality restrictions is the set of the exact cardinalities constraints of value 1 on the object and datatype properties that represent the field element of kx.

The mapping ontology keeps information about the name of the constraint, the element that hosts the declaration of the key constraint and the actual paths of the selector and field elements. Let ki be an individual of the "IdentityConstraintInfoType" OWL class, representing the addition information needed for a key constraint. It is described by expression (3.15), where:

ki (id, class_id, key_name, host_element_id, selector_path, fields_paths, constraint_type) (3.15)

- id is the identity of ki, has as value concatenate(element_name, '_', name, '_', type, '_',ki) and is represented by rdf:ID.

- `class_id` is the identity of `ki`, has as value the `id` of the class `SelType` and is represented by the datatype property "`classID`".
- `key_name` is the name of `kx`, has as value the `name` attribute of `kx` and is represented by datatype property "`ICName`".
- `host_element_id` is the identity of the element that hosts the declaration of the key constraint and is represented by the datatype property "`HostElementID`".
- `selector_path` is the XPath expression of the selector of `kx`, has the value of `selector` and is represented by the datatype property "selectorPath".
- `fields_paths` are the XPath expressions of the fields of `kx`, have the values of `fields` and are represented by the datatype properties "fieldPath".
- `constraint_type` is the kind of XML Schema identity constraint, has value "key" and is represented by the datatype property "constraintType".

An example of a key identity constraint is shown in Figure 3.39, and its representations in the main ontology and the mapping ontology are shown, respectively, in Figure 3.40 and Figure 3.41.

```
<xs:element name="Persons" type="PersonsType">
      <xs:key name="KeyPerson">
            <xs:selector xpath="Person"/>
            <xs:field xpath="ID"/>
      </xs:key>
</xs:element>
```

Figure 3.39 XML Schema Key identity constraint

The representation of the OWL class in the main ontology in Figure 3.40 shows the additional syntax with which the algorithm enriches a pre-existing class in order to capture the semantics of the key identity constraint.

```
<owl:Class rdf:ID="PersonType">
      ...
  <owl:hasKey rdf:parseType="Collection">
      <rdf:Description rdf:about="#ID__IDType"/>
  </owl:hasKey>

  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Restriction>
      <owl:onProperty rdf:resource="#ID__IDType"/>
      <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">
      1</owl:cardinality>
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>
```

Figure 3.40 OWL representation of an XML Schema Key Identity constraint

In the mapping ontology the transformation process creates an individual of the class IdentityConstraintInfoType to handle the XML Schema constructs that cannot be directly expressed in OWL syntax.

```
<ox:IdentityConstraintInfoType
rdf:ID="Persons_KeyPerson_PersonsType_ui">
      <ox:classID> PersonType </ox:classID>
      <ox:ICName> KeyPerson </ox:ICName>
      <ox:HostELementID> Persons </ox:HostELementID>
      <ox:selectorPath> Person </ox:selectorPath>
      <ox:fieldPath> ID </ox:fieldPath>
      <ox:constraintType> key </ox:constraintType>
</ox:IdentityConstraintInfoType>
```

Figure 3.41 Individual ki of the class IdentityConstraintInfoType in the mapping ontology that represents the XML Schema Key Identity constraint

3.3.3.4. Representation of the XML Schema Keyref Identity Constraint

In the presentation of the representation of the XML Schema identity constraints were discussed the semantics of the key-keyref pair of constraints. The representation of key was comprehensivly analyzed in subsection 3.3.3.3, where the HasKey axiom was used to capture its semantics. The rich set of the OWL 2.0 axioms has a limitation as far as the citation of a key is concerned, lacking a respective axiom to represent a keyref in the ontology. In this subsection the representation of the XML Schema Keyref identity constraint is desrcibed.

A keyref identity constraint KR(R,S_{KR},F_{KR}) declaration asserts a correspondence, with respect to the content identified by selector S, of the tuples resulting from the evaluation of XPath expression(s) of the fields F, with those of the referenced key K(S_K,F_K) mentioned by attribute refer R. For the representation of KR in the main ontology, the following actions are performed: (a) The OWL property OP_{KR}, which corresponds to the content identified by S_{KR}, is located in the main otnology; and (a) the OWL class $C_{SelType}$, which corresponds to the type of S_K, is asserted to the Range of OP_{KR}. Moreover, KR is represented in the mapping ontology by an IdentityConstraintInfoType individual (see Figure 3.43).

This representation gives added value to HasKey axiom of key's representation, as it makes feasible to capture the underlying meaning of the XML Schema key-keyref pair, which is the referencing of a unique and mandatory element by another one within the same document. Furthermore, the XML Schema key-keyref identity constraints can now be accurately represented in OWL 2.0 syntax, which was not possible in the XS2OWL 1.0 framework.

An example of this extension of the OWL main ontology is shown in Figure 3.42.

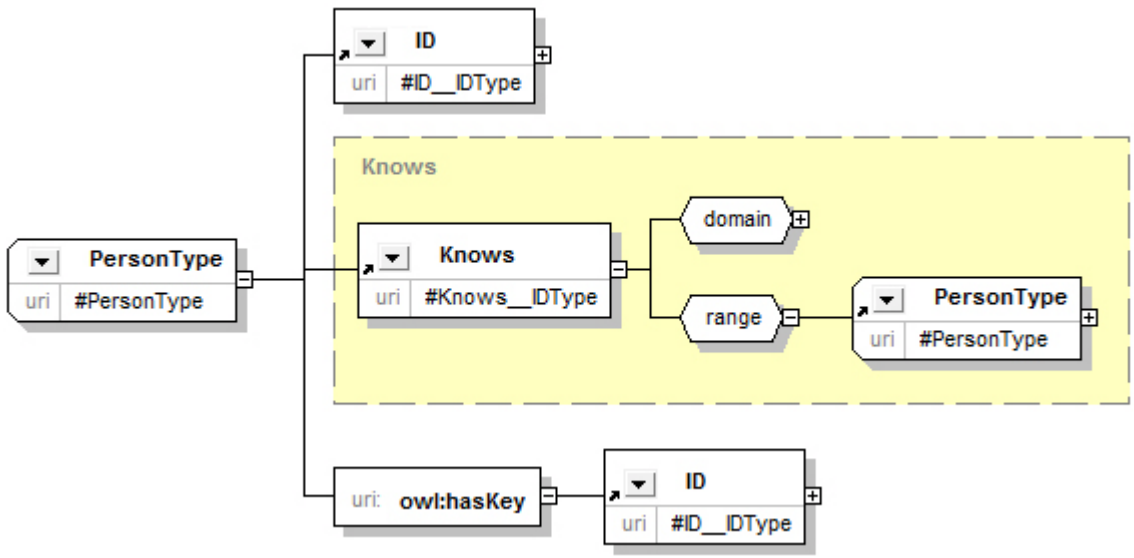

Figure 3.42 Key - Keyref representation in the main ontology

The detailed representation rules of an XML Schema keyref constraint `krx` from the Object Property `op` of the main ontology and the individual `kri` of the mapping ontology are presented in Figure 3.43.

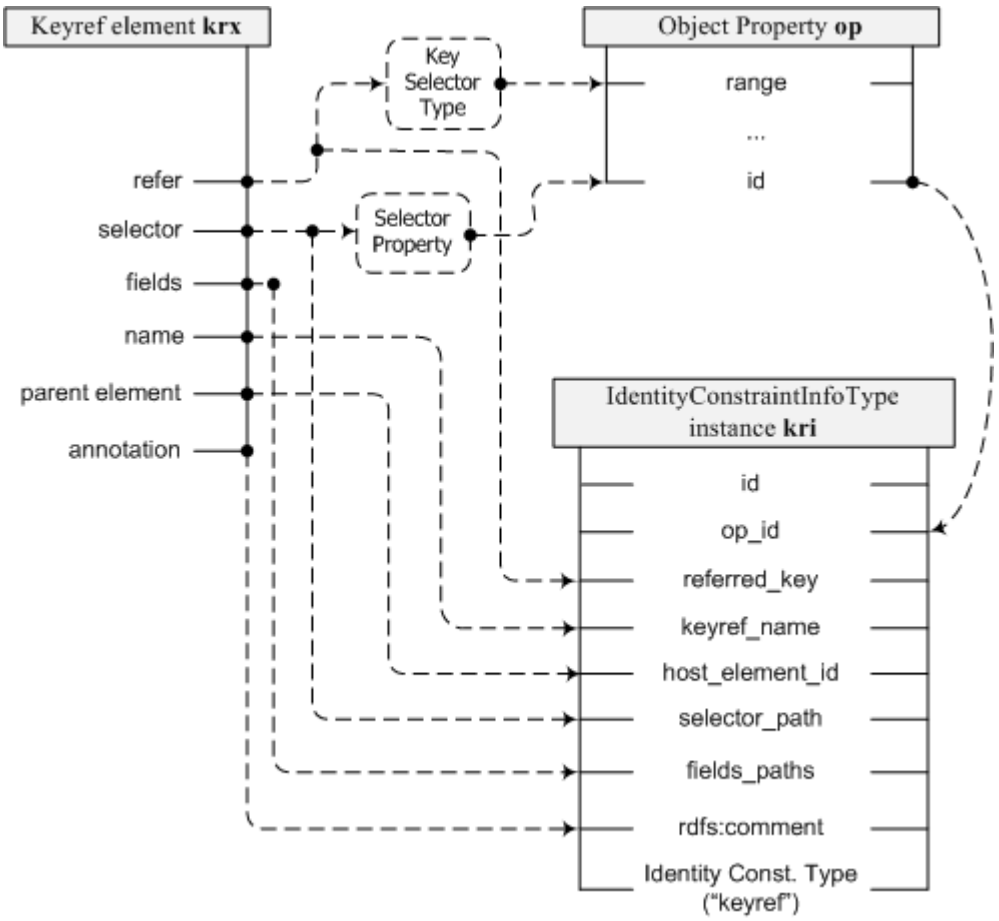

Figure 3.43 OWL representation of XML Schema Keyref Identity constraint

Let `krx` be the XML Schema Keyref constraint that is described by expression (3.16), where:

kx (name, annotation, selector, fields, refer, parent element) (3.16)

- `name` is the value of the "name" attribute of `krx`, which represents its name.
- `annotation` is an optional annotation of `krx`, which represents the value of the optional "annotation" element.
- `selector` is of the "xpath" attribute of the selector element of `krx`.

- `fields` is the set of the “xpath” attribute of the field elements of `krx`.
- `parent element` is the name of the element that hosts the `krx` constraint.
- `refer` is the value of the “refer” attribute of `krx`.

Let `OP` be the OWL object property that represents the XML Schema element of the selector of the referred key in the main ontology. This property exists regardless of the keyref constraint, because it represents an element in the main ontology. If this element is represented by a datatype property due to the type of the element, it is transformed to an object property, so that the range of the property can be assigned a class. In order to locate the element referred by the path of the selector, the **XPathEvaluator** algorithm is used (Figure 3.30) within the process of mapping the Xpath of the selector to a property. This procedure is supported by the **XPathSelector2PropertyID** algorithm (Figure 3.44).

The object property `OP` is enriched with the OWL Range structure for the keyref’s `krx` referred key as it is described by expression (3.17), where:

op (id, range) (3.17)

- `id` is the identity of the object property `op` and is represented by `rdf:ID`.
- `range` is the range of the object property `op`, which has as value the type of the selector of the referred key implemented using the OWL construct `rdfs:range`.

The mapping ontology keeps information about the name of the constraint, the element that hosts the declaration of the keyref constraint, the name of the referred key and the actual paths of the selector and field elements. Let `kri` be an individual of the “`IdentityConstraintInfoType`” OWL class, representing the additional information needed for a keyref constraint. It is described by expression (3.18), where:

kri (id, op_id, keyref_name, referred_key, host_element_id, selector_path, fields_paths, constraint_type, comments) (3.18)

- `id` is the identity of `kri`, has as value `concatenate(element_name, ‘_’, name, ‘_’, type, ‘_’,ki)` and is represented by `rdf:ID`.
- `op_id` is the identity of the object property `op`, has value the `id` of the object property `OP` and is represented by the datatype property “`opID`”.
- `keyref_name` is the name of the keyref element `krx`, has value the `name` attribute of `krx` and is represented by the datatype property “`ICName`”.

- `referred_key` is the name of the key element `kx` referred by `krx`, has value the `refer` attribute of `krx` and is represented by the datatype property "`RefersTo`".

- `host_element_id` is the identity of the element that hosts the declaration of the keyref constraint and is represented by the datatype property "`HostElementID`".

- `selector_path` is the selector's XPath expression of XML Schema key element `kx`, has the value of `selector` and is represented by the datatype property "selectorPath".

- `fields_paths` are the XPath expressions of the fields of `kx`, have the values of `fields` and are represented by the datatype property "fieldPath".

- `constraint_type` is the kind of XML Schema identity constraint, has as value "key" and is represented by the datatype property "constraintType".

- `comments` are the optional annotations of `krx`, has the value of `annotation` and is implemented by the OWL construct `rdfs:comment`.

XPathSelector2PropertyID

The XPathSelector2PropertyID algorithm is used for bridging the gap between the structures that are marked as references of keys in OWL and XML Schema. In XML Schema the identity constraint keyref declares that an element is a reference of another element in contrast to OWL syntax that does not support the ability to apply a keyref-like axiom to the main ontology.

In Figure 3.44, is presented the XPath Selector 2 Property ID algorithm.

Xpath Selector 2 Property ID Algorithm

Input: Selector Xpath *xpath*, Pre-XPath Nodes *pNodes*

Output: Property Name/ID *PN*

1. *selElement* ← *XPathEvaluator* (*xpath*, *pNodes*, "")
2. *selElementName* ← *SelElement*.**append** (/@name)
3. *selElementType* ← *SelElement*.**append** (/@type)
4. *PN* ← *guesstimatePropertyName* (*selElementName*, *selElementType*)
5. **return** *PN*

Figure 3.44 The XPath Selector 2 Property ID Algorithm

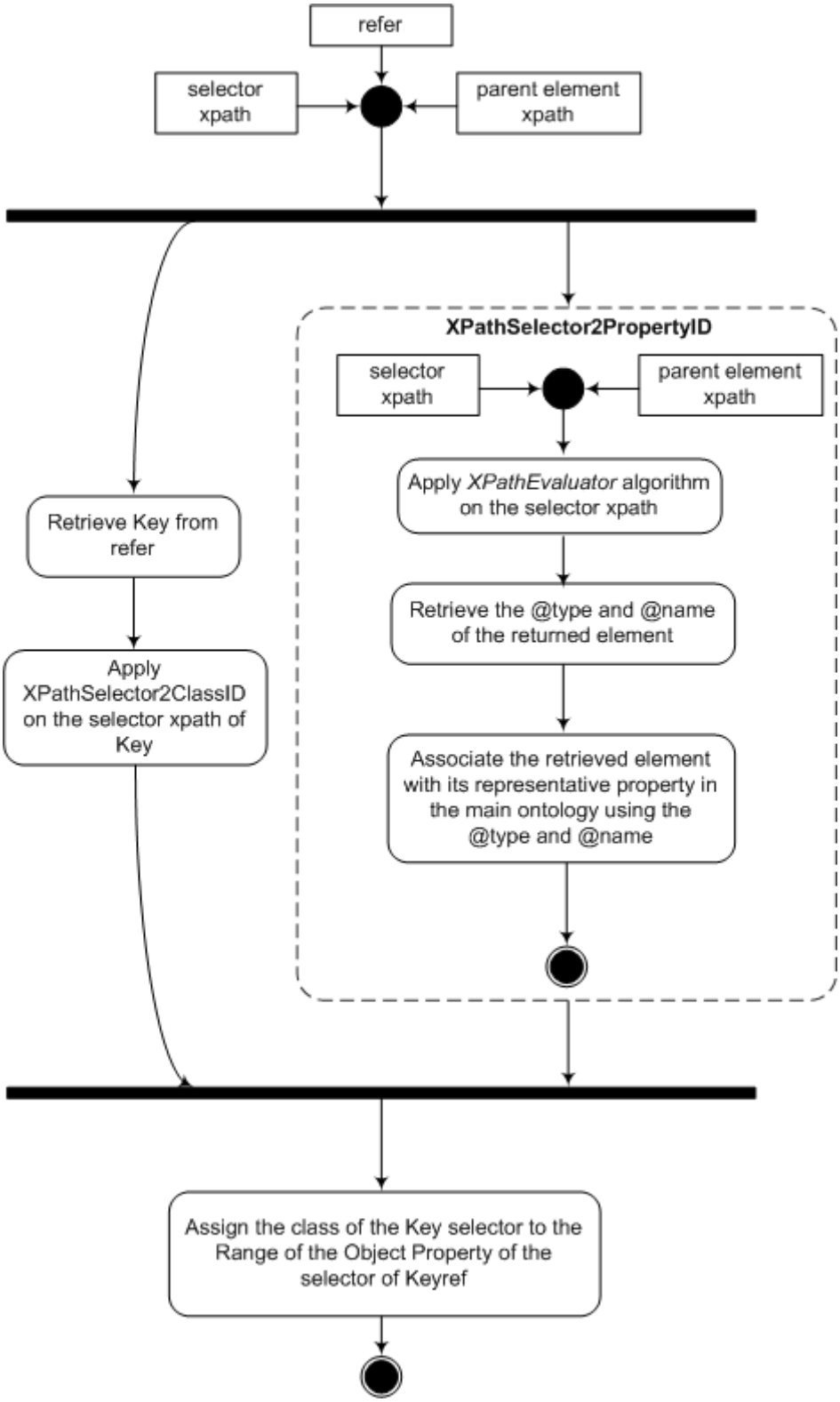

Figure 3.45 Activity diagram of the Keyref representation algorithm

The activity diagram of Figure 3.45 presents the two parallel processes taking place during the representation of the XML Schema Keyref identity constraint in OWL syntax. The first process retrieves the type of the referred key selector and the second one is the *XPathSelector2PropertyID* algorithm. The *XPathSelector2PropertyID* algorithm has two input parameters, "selector xpath" and "Sequence of element nodes". The former represents the XPath expression that should be evaluated over the XML Schema and locates the definition of the element that is referred by the selector element of the identity constraint. The latter takes as value the path of the element that hosts the identity constraint (IC). After the execution of *XPathEvaluator,* the algorithm retrieves the name and the type of the resulting element and associates it with a pre-existing property of the main ontology.

The Range of the returned property is assigned to the class that represents the type of the selector of the referred key. This is feasible using the *XPathSelector2ClassID* algorithm (Figure 3.33).

An example of a keyref identity constraint is shown in Figure 3.46, and its representations in the main ontology and the mapping ontology are shown, respectively, in Figure 3.47 and Figure 3.48.

```
<xs:element name="Persons" type="PersonsType">
      <xs:key name="KeyPerson">
            <xs:selector xpath="Person"/>
            <xs:field xpath="ID"/>
      </xs:key>
      <xs:keyref name="KeyrefKnows" refer="KeyPerson" >
            <xs:selector xpath="Person/Knows"/>
            <xs:field xpath="."/>
      </xs:keyref>
</xs:element>
```

Figure 3.46 XML Schema key and keyref identity constraints

```
<owl:ObjectProperty rdf:ID="Knows__IDType">
      ...
      <rdfs:range rdf:resource="#PersonType"/>
      <rdfs:label>Knows</rdfs:label>
</owl:ObjectProperty>
```

Figure 3.47 OWL representation of an XML Schema Keyref Identity constraint

```
<ox:IdentityConstraintInfoType rdf:ID="Persons_KeyrefKnows_PersonsType_ui">
        <ox:opID> Knows__IDType </ox:opID>
        <ox:ICName> KeyrefKnows </ox:ICName>
        <ox:RefersTo> KeyPerson </ox:RefersTo>
        <ox:HostELementID> Persons </ox:HostELementID>
        <ox:selectorPath> Person/Knows </ox:selectorPath>
        <ox:fieldPath> . </ox:fieldPath>
        <ox:constraintType> keyref </ox:constraintType>
</ox:IdentityConstraintInfoType>
```

Figure 3.48 The Individual kri of the class IdentityConstraintInfoType in the mapping ontology for the representation of an XML Schema Keyref Identity constraint

3.3.4. Representation of XML Schema Override and Redefine

In this subsection is described the OWL representation of the XML Schema Override and Redefine elements according to the XS2OWL 2.0 transformation model.

The OWL syntax does not provide constructs with semantics equivalent or similar to the override and redefine semantics, since this does not allow the representation of these constructs in the main ontology. As a consequence the representation of these constructs exists only in the mapping ontology.

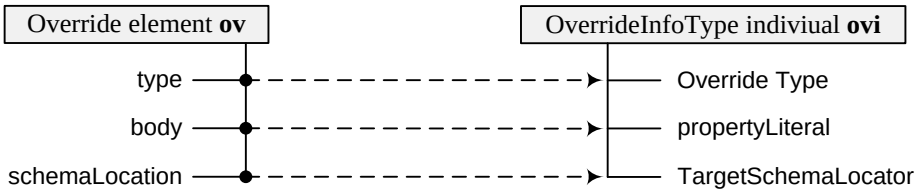

Figure 3.49 OWL representation of XML Schema Override

Let ov be an XML Schema override or redefine element that is described by expression (3.19), where:

ov (type, body, schemaLocation) (3.19)

- `type` is the value of the element name of XML Schema override or redefine element `ov`. Thus, if the element is an override element, `type` has the value "override" and if the element is a redefine element, `type` has the value "redefine".

- `body` is the main part of the definition of `ov`.

- `schemaLocation` is the value of attribute "`schemaLocation`" that appears in the definition of `ov`. This attribute is an anyURI[(1)].

Let `ovi` be an individual of the "`OverrideInfoType`" OWL class. It is described by expression (3.20):

ovi (overrideType, TargetSchemaLocation, propertyLiteral) (3.20)

- `overrideType` is the type of `ov`, has the value of `type` and is represented by the datatype property "overrideType".

- `TargetSchemaLocator` is the target XML Schema of `ov`, has the value of `schemaLocation` and is represented by the datatype property "schemaLocation".

- `propertyLiteral` is the body of `ov`, has the value of `body` and is represented using the OWL construct `rdf:XMLLiteral`.

```
<xs:override schemaLocation="Persons.xsd">
      <xs:simpleType name="validAgeType" >
            <xs:annotation>
                  <xs:documentation>
                        Overrided Simple Type.
                  </xs:documentation>
            </xs:annotation>
            <xs:restriction base="xs:integer">
                  <xs:minInclusive value="23"/>
                  <xs:maxInclusive value="55"/>
            </xs:restriction>
      </xs:simpleType>
</xs:override>
```

Figure 3.50 XML Schema Override element that overrides the simpleType "validAgeType" declared in the XML Schema presented in Figure 2.12

[1] anyURI represents an Internationalized Resource Identifier Reference (IRI). An anyURI value can be absolute or relative, and may have an optional fragment identifier (i.e., it may be an IRI Reference).

In Figure 3.50 is presented an override element. In particular, the sample of the XML Schema shown in the figure specifies that the “validAgeType” simple type of the XML Schema described in the “Persons.xsd” file will be replaced by the definition that is contained within the override element for the current XML Schema. Thus, this schema reuses the corporate schema “Persons.xsd”, but customizes the "validAgeType" with the definition that is particular to the current XML Schema.

The XS2OWL 2.0 transformation Model represents it with the “OverrideInfoType” individual shown in Figure 3.51 in the mapping ontology. The “OverrideInfoType” individual captures the location of the XML Schema that is being modified by the override/redefine element as well as the XML constructs that replace the correspondent structure of the target XML Schema.

```
<ox:OverrideInfoType>
      <ox:overrideType>override</ox:overrideType>
      <ox:schemaLocation>Persons.xsd</ox:schemaLocation>
      <rdf:XMLLiteral>
            <xs:simpleType name="validAgeType" >
                  <xs:annotation>
                        <xs:documentation>
                        Overrided Simple Type.
                        </xs:documentation>
                  </xs:annotation>
                  <xs:restriction base="xs:integer">
                        <xs:minInclusive value="23"/>
                        <xs:maxInclusive value="55"/>
                  </xs:restriction>
            </xs:simpleType>
      </rdf:XMLLiteral>
</ox:OverrideInfoType>
```

Figure 3.51 OWL representation of an XML Schema Override in the mapping ontology

3.3.5. Representation of XML Schema Alternative

In this subsection is described the OWL representation of the XML Schema Alternative elements, according to the XS2OWL 2.0 transformation model.

The OWL syntax does not support the definition of structures with semantics equivalent or similar to alternative semantics. OWL does not implement such a mechanism. This restriction prevents the

usage of main ontology in this representation. Thus, these elements have to be transferred to the mapping ontology without interfering, in any way, with the main ontology.

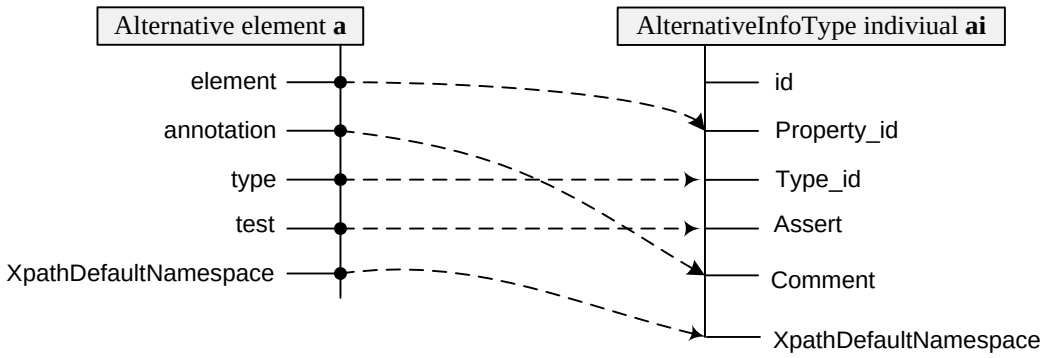

Figure 3.52 OWL representation of XML Schema Alternative

As shown in Figure 3.52, the alternative element “a” contains the “type” attribute and the “test” attribute, which are the main parts of an alternative element as described in section 2.3.2. Moreover, the “element” construct shown in Figure 3.52, represents the element’s name in which the alternative element is defined and applies to.

In Figure 3.53, is presented an example of the XML Schema alternative element. In particular, the “Person” element has the “PersonType” complex type. Moreover, the alternative element assigns a different complex type to the “Person” element according to the value of the “kind” attribute, where in case the “kind” has the “employee” value then the type of the “Person” element is set to the “EmployeeType” complex type.

```
<xs:complexType name="PersonType">
    <xs:group ref="personGroup"/>
    <xs:attribute name="kind" type="xs:string" />
</xs:complexType>
<xs:element name="Person" type="PersonType">
    <xs:alternative  test="@kind eq 'employee' "  type="EmployeeType" />
</xs:element>
```

Figure 3.53 XML Schema alternative

The XS2OWL 2.0 transformation Model represents the alternative element with the `ai` “AlternativeInfoType” individual shown in Figure 3.54 in the mapping ontology. The “AlternativeInfoType” individual captures the main ontology property that reprsesents the element, in which the alternative element is defined according to the XML Schema. Furthermore, `ai` captures the XPath expression that is being evaluated as well as the new type of the element if the evaluation is successful.

```
<ox:AlternativeInfoType>
      <ox:PropertyID> Person_PersonType</ox:PropertyID>
      <ox:typeID> EmployeeType</ox:typeID>
      <ox:assertExpr> @kind eq 'employee'</ox:assertExpr>
</ox:AlternativeInfoType>
```

Figure 3.54 OWL representation of an XML Schema Alternative in the mapping ontology

3.3.6. Representation of XML Schema Assert

In this subsection is described the OWL representation of XML Schema Assert elements, based on XS2OWL 2.0 transformation model.

OWL syntax does not support the definition of structures with semantics equivalent or similar to assert semantics. OWL does not implement such a mechanism. This limitation prevents the usage of main ontology in this representation. Thus, these elements have to be transferred to the mapping ontology without interfering, in any way, with the main ontology.

In the individual of the "`AssertInfoType`" class is stored the information about the complex type in which the assert element "a" is defined in the source XML Schema via the element "`ComplexType`".

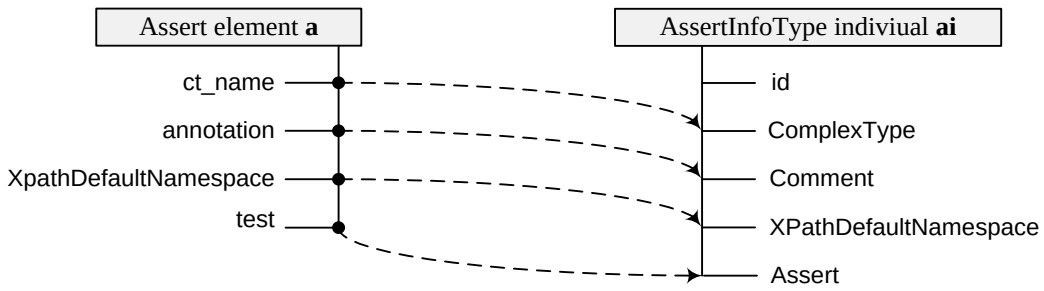

Figure 3.55 OWL representation of XML Schema Assert

Let a be an assert element nested in a complex type `ct`, with name `ct_name`. In Figure 3.55, is described the representation of the assert element a in the mapping ontology by using an individual of the "`AssertInfoType`" class.

In **Σφάλμα! Το αρχείο προέλευσης της αναφοράς δεν βρέθηκε.**, is presented an example of the XML Schema assert element. In particular, it is specified that the value of the “max” attribute of the complex type defined within the “intRange” element must be less or equal to 6.

```
<xs:element name="intRange">
<xs:complexType>
  <xs:attribute name="min" type="xs:int"/>
  <xs:attribute name="max" type="xs:int"/>
  <xs:assert test="@max le 6"/>
</xs:complexType>
</xs:element>
```

Figure 3.56 XML Schema Assert element

The XS2OWL 2.0 transformation Model represents the assert element with the ai “AssertInfoType” individual shown in Figure 3.57 in the mapping ontology. The “AssertInfoType” individual captures the main ontology class that reprsesents the complex type, in which the assert element is defined according to the XML Schema, as well as the XPath expression that is being evaluated.

```
<ox:AssertInfoType>
      <ox:ComplexTypeID> NS_intRange_UNType</ox:ComplexTypeID>
      <ox:assertExpr> @max le 6 </ox:assertExpr>
</ox:AssertInfoType>
```

Figure 3.57 OWL representation of an XML Schema Assert in the mapping ontology

3.4. Summary

This chapter described the transformation of XML Schemas 1.1 into equivalent OWL 2.0 syntax via the XS2OWL 2.0 framework. The architecture of the framework and the correspondeces between the XML Schema structures and OWL 2.0 syntax are presented in the subsections of this chapter. The transformation process of an XML Schema into OWL syntax is described through UML activity diagrams. The rules that apply on the representation of XML Schema structures with OWL syntax are described in both an abstract way and a formal way in order to clarify the main idea of the representation rules and procedures. Moreover, this chapter presented the mapping ontology and the rules that apply on an XML Schema in order to generate the mapping ontology.

Finally, it is worthwhile to mention that the generated main ontology captures the semantics of the source XML Schema and is used through the SPARQL2XQuery framework in order to the translation of SPARQL queries in correspondent XQuery syntax. Thus, the next chapter (Chapter 4) analyzes the SPARQL2XQuery 2.0 framework and the way in which these two frameworks interact and cooperate.

4. The SPARQL2XQuery 2.0 Framework

The Semantic Web infrastructure includes the creation and management of large OWL/RDF [W3C/RDFS][W3C/RDF][W3C/OWL1][ZiDaCeCoEl] knowledge bases accessed using the SPARQL query language [W3C/SPARQL]. Furthermore, it aims to make the semantics of the data explicit and available to semantically enabled web applications and services. In the Semantic Web environment it is essential to have transparent, semantic-based access to information stored in heterogeneous information sources, since the Semantic Web applications and services have to coexist and interoperate with other software and in particular with software that allows access to legacy systems like the very large audiovisual digital libraries of the digital video broadcasters, the digital libraries of the cultural heritage institutions etc.

The XML/XML Schema [W3C/XML][W3C/XSD1] are currently the dominant standards for information exchange in the Web and for the representation of semi-structured information. In addition, many international standards have been expressed in XML Schema syntax like, for example, the MPEG-7 [ChSiPuMPEG7] standard in multimedia content description, the METS

[METS] standard in digital libraries, etc. The above have resulted in the storage of large XML data sets which are accessed using the XQuery query language.

Since the Semantic Web and XML environments have different data models, different semantics and use different query languages to access them, it is very important to develop tools and methodologies that will allow to the bridge the Semantic Web with the XML data, thus facilitating XML data retrieval from within the Semantic Web environment.

Through this thesis we have extended the SPARQL-to-XQuery translation ([BiGiTsCho9]) has been extended in the contex of this thesis, in order to exploit the OWL 2.0 semantics and the new constructs introduced by XML Schema 1.1. In particular, the SPARQL-to-XQuery translation has been extended in order to support the XML Schema datatypes and the XML Schema Identity constraints.

The SPARQL2XQuery 2.0 has been implemented as an extension of the SPARQL2XQuery 1.0 framework, using Java related technologies (Java 2SE, Axis2 and Jena) and the Oracle Berkeley DB XML database.

The rest of this chapter describes in brief the new features of the SPARQL2XQuery 2.0 in section 4.1, the architecture of SPARQL2XQuery 2.0 framework in section 4.2, the Mappings between the ontology and the XML Schema of the XML DB in section 4.3, the Query translation process of SPARQL queries in XQuery syntax in section 4.4 and a summary of the chapter in section 4.5.

4.1. New Features

In this section is presented an overview of the new features that SPARQL2XQUERY 2.0 introduces. As already mentioned, the SPARQL–to–Xquery translation of SPARQL2XQUERY 1.0 ignored the XML Schema simple types and identity constraints due to the OWL 1.0 limitations of the OWL 1.0 syntax which resulted in limitation of XS2OWL 1.0. In subsections 4.1.1 and 4.1.2 is discussed the enhancement in the presentation of the XML Schema datatypes (simple types) and the XML Schema identity constraints respectively.

4.1.1. XML Schema datatypes

The SPARQL query language supports queries, in which datatype references could be exploited in order to define the type of a literal. For example, consider the literal "42". Using the syntax "42"^^xsd:integer, "42" is stated to be an integer, while with the syntax "42"^^xsd:string, "42" is stated to be a string and with the syntax "42"^^ns:ValidAgeType "42" is stated to be of a user-defined type (ValidAgeType). According to the SPARQL specification, literals and datatype references could appear either in the object part of a Triple Pattern or in a Filter Expression of an SPARQL query.

Using the XS2OWL 2.0 transformation model, the XML Schema simple datatypes are represented in OWL 2.0 syntax as presented in Section 3. Exploiting the information of the generated mappings between the main ontology produced by XS2OWL 2.0 and the initial XML Schema, the SPARQL2XQuery framework can handle queries that include datatype references in their literals.

4.1.2. XML Schema identity constraints

Since the OWL 2.0 allows the representation of the XML Schema identity constraints and the XS2OWL 2.0 transformation model supports their representation, the SPARQL-to-XQuery translation of the SPARQL2XQuery 1.0 framework has been extended in SPARQL2XQuery 2.0 to exploit the XML Schema identity constraints during the translation. The identity constraints can be exploited in queries which contain the same variables between more than one Triple Patterns in a SPARQL query Graph Pattern. The SPARQL2XQuery 2.0 framework, can handle this class of SPARQL queries, since it exploits the identity constraint information and the mappings between the generated ontology and the XML Schema.

4.2. Framework Architecture

In this section, is outlined the architecture of SPARQL2XQUERY 2.0 in brief, as it is the same with the one of SPARQL2XQUERY 1.0 except for the changes inside some components in order to exploit the new features of the standards handed by XS2OWL 2.0. The changes inside the framework modules, are presented in the following sections.

SPARQL2XQUERY 2.0 includes three main components, i.e. the mappings generator & encoder, the query translator (SPARQL-to-XQUERY translator) and the result transformer. The SPARQL2XQuery 2.0 Query Translator component comprises of the following sub-components:

- The SPARQL Graph Pattern Normalizer, which rewrites the Graph-Pattern (GP) of the SPARQL query in an equivalent normal form, based on equivalence rules. This makes the GP translation process simpler and more efficient.

- The Variable Type Specifier, which identifies the types of the variables in order to detect any conflict arising from the syntax provided by the user as well as to identify the form of the results for each variable. Moreover, the variable types are used by the Onto-triple Processor and the Variable Binder sub-components.

- The Onto-Triple Processor, which processes onto-triples (actually referring to the ontology structure and/or semantics) against the ontology (using SPARQL) and based on this analysis binds the correct XPaths to variables contained in the onto-triples. These bindings are going to be used in the next steps as input to the Variable Binder sub-component.

- The Variable Binder, which is used in the translation process for the assignment of the correct XPaths to the variables referenced in a given Basic Graph Pattern (BGP, a

sequence of triple patterns and filters), thus enabling the translation of BGPs to XQuery expressions.

- The Basic Graph Pattern Translator, which performs the translation of a BGP into semantically equivalent XQuery expressions, thus allowing the evaluation of a BGP on a set of XML data. The translation is based on the BGP2XQyuery algorithm (available at TO DO). The algorithm takes as input the mappings between the ontology and the XML schema, the BGP, the determined variable types and the variable bindings and generates XQuery expressions.

- The Graph Pattern Translator, which translates a graph pattern (GP) into semantically equivalent XQuery expressions. The concept of a GP is defined recursively. The Basic Graph Pattern Translator sub-component translates the basic components of a GP (i.e. BGPs) into semantically equivalent XQuery expressions, which however have to be properly associated in the context of a GP. This means to apply the SPARQL operators (i.e. AND, OPT, UNION and FILTER) among them using XQuery expressions and functions.

- The Solution Sequence Modifiers Translator, which translates the SPARQL solution sequence modifiers using XQuery clauses (Order By, For, Let, etc.) and XQuery built-in functions. The solution Modifiers are applied on a solution sequence in order to create another, user desired, sequence. The modifiers supported by SPARQL are Distinct, Reduced, OrderBy, Limit, and Offset.

- The Query Forms Translator, which is responsible for the final step of the translation of the SPARQL query in XQuery expressions. SPARQL has four forms of queries (Select, Ask, Construct and Describe). Accord-ing to the query form, the structure of the final result is different. In particular, after the translation of the solution modifiers, the generated XQuery is enhanced with appropriate expressions in order to achieve the desired structure of results (e.g. to construct an RDF graph, or a result set) according to the query form.

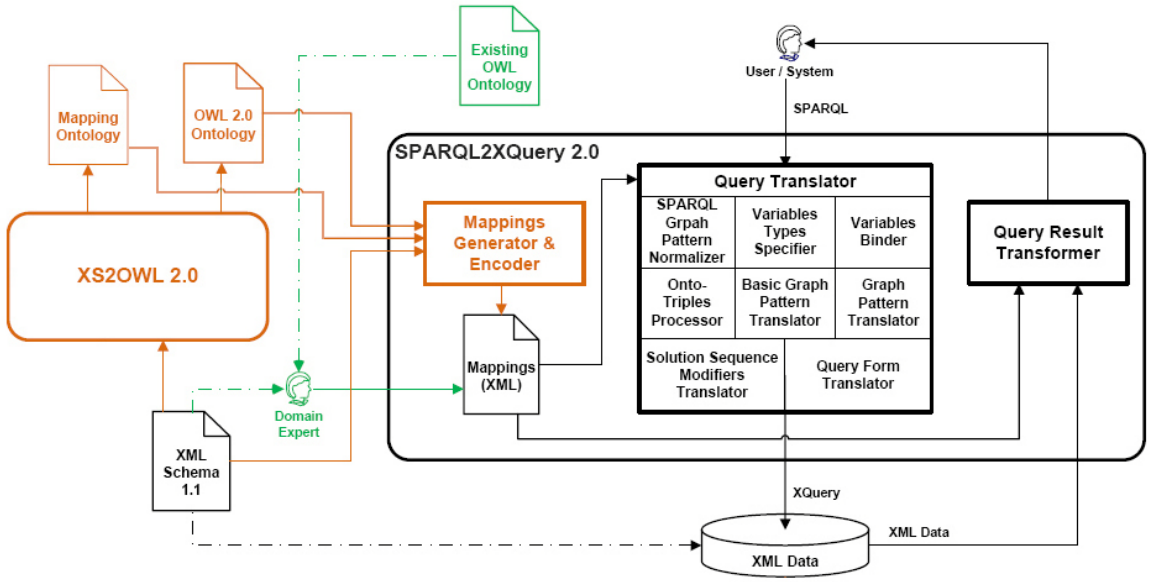

Figure 4.1 The SPARQL2XQuery 2.0 Framework

Figure 4.1 presents the SPARQL2XQUERY 2.0 framework and it's working environment including the required inputs and the different scenarios supported by this thesis. The SPARQL2XQUERY 2.0 framework allows the evaluation of SPARQL Queries over XML Data. If the queries are posed on top of existing ontologies, mappings between the ontologies and the underlying XML Schema(s) should be manually specified. These mappings play a significant role in the SPARQL query translation in XQuery syntax. If it is not uses in conjunction with an existing ontology, the XS2OWL 2.0 Transformation Model is applied on the XML Schema syntax, expresses it in OWL 2.0 syntax and the mappings are automatically generated.

In the rest of this subsection is described the workflow of the SPARQL2XQUERY framework in order to clarify the interaction between the outputs and the inputs of the different components through out the translation process of a SPARQL query into semantically equivalent XQuery syntax.

4.3. Mappings

It has already been mentioned section 2.12, that the mapping model allows the expression of mappings between OWL ontologies and XML Schemas in order to be able to translate SPARQL queries to XQuery expressions, since such mapping are necessary for the translation to be feasible.

The process of mapping generation may be automatic or manual. The former is used in the scenario of querying XML data based on an automatically generated by using XS2OWL 2.0 framework OWL ontology from the XML Schema; The latter is used to the scenario of querying

XML data based on an existing OWL ontology. It is worth-while that both these scenarios conclude in one common mapping file that follows the rules of the mappings specification process that is presented in this section. This way, a flexible and scenario independent translation process of SPARQL queries to XQuery syntax was develop.

Overall, the mappings and enhancement achieved in the context of this thesis are list below:

- Definition of the new XPath set operator;
- Exploitation of the mapping ontology in case of automatically generated mappings;
- Modifications in order to handle the multi-valued substitutionGroup elements supported by of XML Schema 1.1;
- Support for XML Schema simple types;
- Support for XML Schema identity constraints (key-keyref).

The mapping ontology is used only during mapping generation. This approach makes feasible to avoid any doubts about dissimilarities in the efficiency of the query translation process between the two scenarios of mappings, manual and automatic.

In this section is described in detail the specification of the mappings. The process of specifying the mappings between the XML Schema and the OWL 2.0 ontology is described in subsection 4.3.1, the mappings between the XML Schema Simple Types and the datatypes of the OWL 2.0 ontology are described in subsection 4.3.2, the mappings between the XML Schema Unnamed Simple Types and datatypes of the OWL 2.0 ontology are described in subsection 4.3.3 and the mappings between the OWL 2.0 identity constraints ontology and the XML Schema identity constraints (unique, key, keyref, ID, IDREF, IDREFS) are described in subsection 4.3.4.

4.3.1. XPath set operators

In this subsection, is defined a set of operators that are applied on XPaths. These operators are used during the specification of the mappings and also through the query translation. The operator are both unary operators and binary operators.

Definition 4.1 Leaf Node operator. The unary Leaf Node operator ^{LN}, is applied on a set of XPaths X and returns the a set containing the distinct names of all the leaf nodes (i.e. the last nodes) of X.

Example

- Let $\mathbf{X}$={ /a/b , /c/b , /e/f/g } then $\mathbf{X}^{LN}$={ b , g }.

Definition 4.2 Parent operator. The unary Parent operator ^{P},is applied on a set of XPaths X and returns the set of the distinct parent XPaths (i.e. the same XPaths without the leaf nodes of X). When applied to the root node Rx, the operator returns Rx.

Example

- Let **X**={ /a , /a/b , /c/d , /e/f/g , /b/@f } then $\mathbf{X}^{P}$={ /a , /c , /e/f , /b }.

Definition 4.3 Union operator. The binary Union operator ∩, is applied on two XPath sets X and Y and has a special behavior in comparison with the classic set theory union operator: If some member of X or Y has the wildcard character (*), then in the resulting set the more specific XPaths are excluded.

Examples

- Let **X**={ /a , /a/b , /c/d } and **Y**={ /a/b/c } then **X**∪**Y**={ /a , /a/b , /c/d , /a/b/c }.
- Let **X**={ /a/*, /c/d } and **Y**={ /a/b, /m } then **X**∪**Y**={ /a/*, /c/d, /m }.
- Let **X**={ /a , /* , /c/d } and **Y**={ /m } then **X**∪**Y**={ /* }.

Definition 4.4 Intersection operator. The binary Intersection operator ∪, is applied on two XPath sets X and Y and has a special behavior in comparison with the classic set theory intersection operator: if some member of X or Y has the wildcard character (*) then in the resulting set the more general XPath is excluded.

Examples

- Let **X**={ /a , /a/b , /c/d } and **Y**={ /a/b/c, /c/d } then **X**∩**Y**={ /c/d }.
- Let **X**={ /a/*, /c/d } and **Y**={ /a/b, /m } then **X**∩**Y**={ /a/b }.
- Let **X**={ /a , /* , /c/d } and **Y**={ /m } then **X**∩**Y**={ /m }.

Definition 4.5 Append operator. The binary Append operator ***/*** is applied on an XPath set X and a set of node names N, resulting in a new set of XPaths Y formed by appending each member of N to each member of X. Thus, the elements (XPaths) of the resulting set have as fathers the elements of the left operand set and as leaf nodes the elements of the right operand set.

$$\mathbf{X/N = \{\ xpaths : xpaths^{P} \in X \text{ and } xpaths^{LN} \in N\}}$$

Example

- Let **X**={ /a/b , /c/d } and **Y**={ k , m } then **X/Y**={ /a/b/k , /c/d/k , /a/b/m , /c/d/m }.

Definition 4.6 Right Child operator. The binary Right Child operator ⊕, is applied on two XPath sets X and Y and returns the members (XPaths) of the right set Y, the parent XPaths of which are contained in the left set X.

Examples

- Let **X**={ /a/b , /c/d } and **Y**={ /a/b/c , /b/m } then **X**⊕**Y**={ /a/b/c }.
- Let **X**={ /a/b , /c/* } and **Y**={ /a/b/k, /b/k , /c/m, c/t/y } then **X**⊕**Y**={ /a/b/k, /c/m, c/t/y }.

Definition 4.7 Right Child Isolation operator. The binary Right Child Isolation operator ⊖, is applied on two XPath sets X and Y and returns the members (XPaths) of the right operand set Y detached from the parent XPaths, which are contained in the left operand set X.

Examples

- Let **X**={ /a/b , /c/d } and **Y**={ /a/b/c , /b/m } then **X**⊖**Y**={ /c }.
- Let **X**={ /a/b } and **Y**={ /a/b/c/d , /a/b/c/@e } then **X**⊖**Y**={ c/d, /c/@e }.

4.3.2. Specification of the Mappings

During the mapping process, the XML Schema is regarded as a set **XMLS={CT,ST,CE,SE,AT}**, where CT stands for Complex Types, ST stands for Simple Types, CE stands for Complex Elements, SE stands for Simple Elements and AT stands for Attributes.

During the mapping process, an ontology is regarded as a set **O={ CL, DT, DTP, OP }**, where CL stands for Classes, DT stands for Datatypes, DTP stands for Datatype properties and OP stands for Object Properties.

The mapping of an **Ontology O** with an **XML Schema XMLS** is a set of mappings that capture the semantic correlations between the members of O and XMLS.

$$M_{[O,XMLS]} = \{\mu(o_i, xmls_i) | o_i \in O \text{ and } xmls_i \in XMLS\}$$

A mapping $\mu(o_i , xmls_i)$ between two members of the O and XMLS sets, captures a semantic correlation between them. Every single member of these sets may have more than one mappings.

The set $M_{[O,XMLS]}$ of the mappings is stored in an XML Document which follows the XML Schema "MappingsSchema", which has been extended since the first version of SPARQL2XQUERY in order to cover the needs of the Simple Types and the Identity constraints. The "MappingSchema" XML Schema is presented in Appendix A.

The general workflow of the mapping process is presented in Figure 4.2. First of all, it captures any useful information and constructs from the XML Schema and the OWL 2.0 ontology. After

doing so it relies on the representation rules of XS2OWL 2.0 (Table 3.2) and on the set of algorithms developed in order to specify the mappings between these two environments. The previously developed set of algorithms, in the context of SPARQL2XQuery 1.0 [Bik08], are the "**findUnnamedComplexTypeXpaths**" algorithm for grabbing an the XPath expression from the name of an unnamed complex type, the "**determSubClassesXpaths**" algorithm for enriching the set of Xpaths of a class with the Xpaths that correspond to its subclasses, the "**determPropertiesXpaths**" algorithm for determining the mappings between properties and Xpaths and the "**determSubPropertiesXpaths**" algorithm for enriching the set of Xpaths of a property with the Xpaths that correspond to its subproperties.

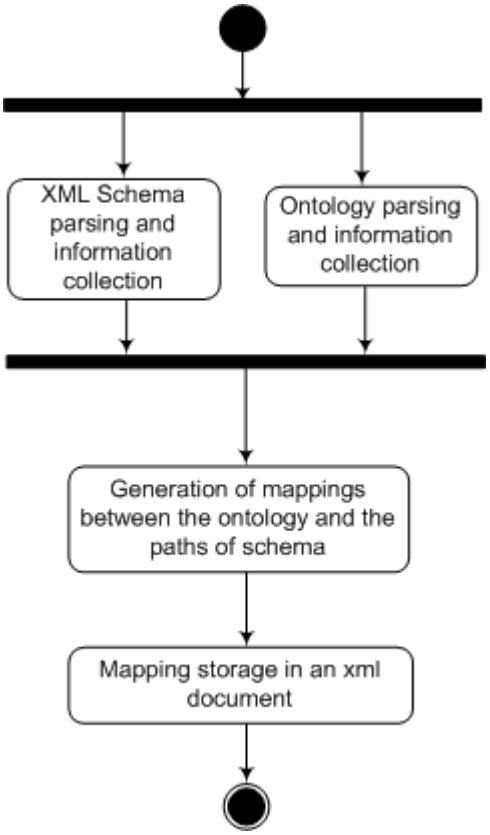

Figure 4.2 Specification of the Mappings

The activity diagram of Figure 4.2, shows the mapping specification process of the. The first step includes parsing the XML Schema and the collection of the needed information. This is achieved by serializing the xml schema using the XML Beans API and saving it as a DOM document. The DOM document is used as input to the XPath Over Schema API [XpathOverXS]. This API may evaluate an XPath over the XML Schema and specify a set of possible paths for the attributes and elements that correspond to simple types and complex types, respectively. In Figure 4.3, is presented an example of the Simple and Complex Type path specification process.

For the simple type “ValidAgeType”: the path `/Persons/Person/Age` and `/Persons/Employee/Age`

For the complex type “PersonType”: the path `/Persons/Person/`

Figure 4.3 Path specification for named Simple and Complex Types

During the first step, the main and the mapping ontologies of XS2OWL 2.0 are parsed in parallel and represented in the main memory as Jena graphs using the Jena API. In case of OWL 2.0 constructs that are not supported by the Jena API, the DOM parser is used. In Figure 4.4 and Figure 4.5, are presented, respectively, the needed information to be collected for datatypes and classes from the main ontology and the needed information to be collected for identity constraints from the mapping ontology.

For the ontology of Persons, the captured information about datatypes:

- ValidAgeType

For the ontology of Persons, the captured information about classes:

- PersonsType
- PersonType
  - “EmployeeType” class is subclass of “PersonType” class
  - Has key on “ID_IDType” property
- EmployeeType
- AddressType

Figure 4.4 Main ontology information collection

For the mapping ontology of Persons, the captured information about identity constraints:

- Knows_IDType
  - Used as key reference to PersonType

Figure 4.5 Mapping ontology information collection

In the rest of the section are described the enhancement of the SPARQL2XQUERY mapping motel. These include the mappings for Simple Types and Identity Constraints.

4.3.3. Mappings specification for XML Schema Simple Types

The mapping specification proscess between OWL Datatypes and XML Shema Simple Types is presented in the Figure 4.6. Each node of the diagram is analyzed afterwards in order to make clear the set of the underlying operations.

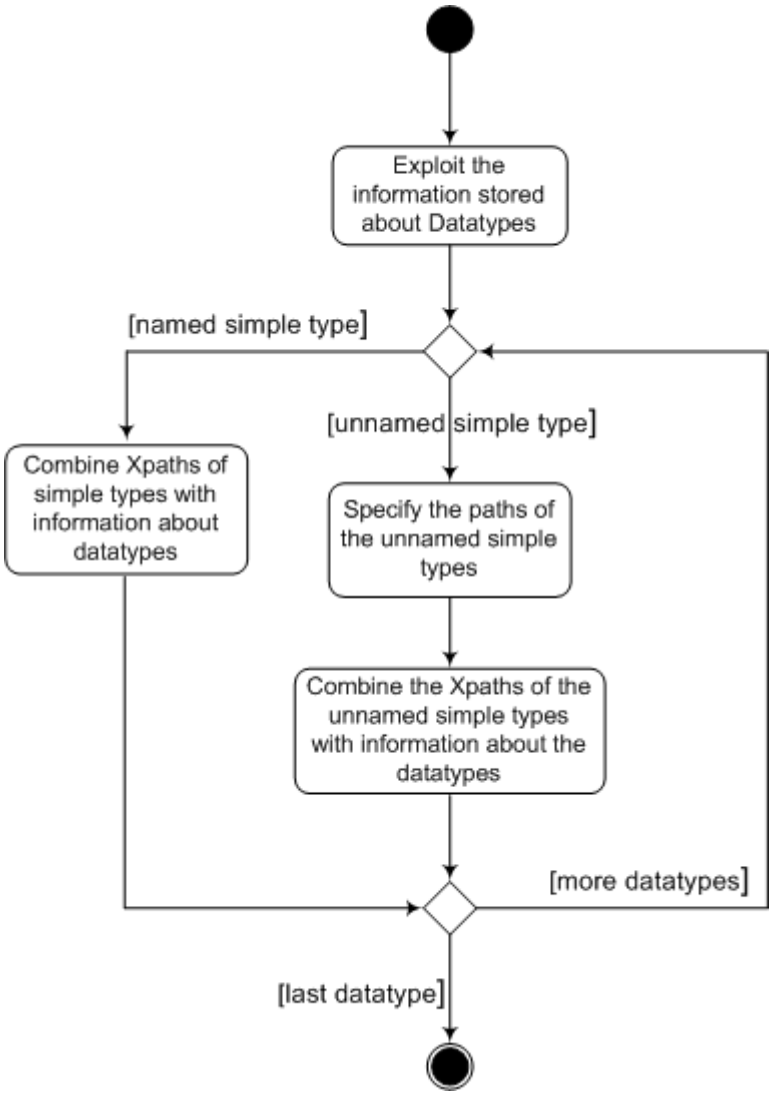

Figure 4.6 Specification of mappings between Datatypes and Simple Types

In the rest of this subsection the specification of mappings between Datatypes and Simple Types is presented. According to Figure 4.6, the following activities take place:

Exploit the information stored about the datatypes

Section 4.3.1 presented the process of capturing and saving the necessary information about the owl datatypes and the xml schema simple types. During this activity the name of each datatype is matched with the set of the corresponding simple type names. The result of this matching process

marks each simple type as being named or unnamed. As already metioned, the name of a datatype is the same with the name of the semantically equivalent named simple type. This distinction is the basis of the next activities of the activity diagram.

Named simple types

If a datatype represents a named simple type, the set of the xpaths of the datatype agrees with the set of xpaths that have been previously assigned to the conserporting simple type by taking in to consideration the elements and attributes that are declared to follow this type.

Specification of the paths of the unnamed simple types

If the datatype represents an unnamed simple type, the set of xpaths of the datatype has to be specified through in this step. The unnamed datatypes were ignored in the xml parsing and exploration process due to the fact that these datatypes do not appear with some name value in the XML schema document. In the context of this activity, the **findUnnamedSimpleTypeXpaths** algorithm has been developed. This algorithm allows locating the unnamed simple type in the xml schema and the XPath set it is assigned to, relying on the XS2OWL transformation model and the naming convention rules. The **findUnnamedSimpleTypeXpaths** algorithm is discussed comprehensively in subsection 4.3.2.2.

4.3.3.1. Mappings for named Simple Types

According to the XS2OWL 2.0 tranformation model the simple XML Shema types are represented as datatypes in the main ontology, since both these constructs refer to simple types. This is a 1:1 mapping, which means that every simple type is associated with one and only one datatype and vice versa every datatype is associated with one and only one simple type. Every named datatype has identify) equal to the name attribute of the simple type.

As far as ontology information capturing is concerned, the required information about the datatypes is the **name** of the datatype, which is represented by the OWL construct rdf:ID.

The mappings SPARQL2XQuery have to be enriched with the needed information about the Datatypes of the main ontology and the corresponding Simple Types of the XML Schema. This need is covered by introducing the map:Datatype in the XML Schema of the mapping file. This construct holds the XPaths that point to instances of a specific Datatype in an XML document. The Datatype declaration in the schema followed by the SPARQL2XQuery mappings is presented in Figure 4.7.

```
<xs:complexType name="Datatype_Type">
  <xs:sequence>
     <xs:element name="Xpath" type="xs:string" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="name" type="xs:string"/>
</xs:complexType>
```

Figure 4.7 Datatype declaration Schema of the mappings

As an example, consider the XML Schema of Figure 2.12, which describes a set of Persons. The Simple Type "ValidAgeType" of this Schema is mapped to the homonym owl:Datatype in the automatically generated by the XS2OWL 2.0 framework main ontology. The needed information about this pair in the mappings is presented in Figure 4.8.

```
<map:Datatype name="ValidAgeType">
      <map:Xpath>/Persons/Person/Age</map:Xpath>
      <map:Xpath>/Persons/Employee/Age</map:Xpath>
</map:Datatype>
```

Figure 4.8 Example of Datatype mapping information

The XPath set of the "ValidAgeType" is used in the translation process in order to answer queries that refer to literals of this simple type.

4.3.3.2. Mappings for unnamed Simple Types

The representation of the unnamed XML Schema simple types was discuss in section 3.3.2.4. Unnamed datatypes are always children of constructs and are valid in the scope of those constructs. Moreover, the unnamed simple types may be nested inside other simple types (unnamed or named), elements, attributes, attribute-groups or groups. In this section is described how the XML Schema unnamed simple types are taken in consideration by the SPARQL2XQUERY 2.0 framework in the query translation and encoding of the mapping. Thus, the rest of this section presents the mapping model extensions needed in order to exploit the unnamed simple types.

As far as unnamed simple types nested in elements, attributes, attribute groups and groups are concerned, a process similar with the one of determining the xpaths of unnamed complex types described in SPARQL2XQuery 1.0 is followed.

```
<xs:simpleType name="Salary">
    <xs:union>
        <xs:simpleType>
            <xs:restriction base="xs:int">
                <xs:minExclusive value="900"/>
            </xs:restriction>
        </xs:simpleType>
        <xs:simpleType>
            <xs:restriction>
                <xs:simpleType>
                    <xs:restriction base="xs:int">
                        <xs:minInclusive value="950"/>
                    </xs:restriction>
                </xs:simpleType>
                <xs:maxInclusive value="2000"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:union>
</xs:simpleType>
```

Figure 4.9 Simple Type comprised of a Union of nested Unnamed Simple types

Furthermore, for unnamed simple types that are nested within named simple types it is more complex to specify their location inside an XML Schema than unnamed simple types of the previously described case. The reason is that the nested simple types defined in elements, attributes etc. can be mapped to the Xpaths of their parent constructs (elements, attributes etc.) as these XPaths describe soundly and with no uncertainty the location of the simple type in an XML Schema. However, a nested simple type of a named simple type cannot be mapped to the same set of XPaths with the one of the named simple type, because the named simple type is an extension of the nested on and some individuals of the named simple type may not belong to the value space of the nested type. For example, consider a simple type comprised of a union of nested unnamed simple types like the one of Figure 4.9. According to that example, a valid value of the simple type “Salary” may be the value “2100”. This value is valid for the first member of the union but not for the second member. Thus, the XPath set of the Simple Type “Salary” cannot be mapped to the unnamed nested simple types of “Salary”. The first member can represent any value greater than 900, but the second member can represent only values in the range [950,2000].

Consequently the XPaths of the named simple type should be and assigned to the unnamed simple types in order to be able to complete the mappings between the XML Schema and the OWL 2.0 ontology generated by the XS2OWL 2.0 framework. The limitation of the XPaths is implemented using (boolean) predicates over the XPaths as described in Figure 2.17. The expression inside the braces of a predicate is determined according to the restriction facets applied over the base datatype of the unnamed simple type and according to the base type of the unnamed simple type.

Let **v** be the value of the constraining facet. Let · be the context of the XPath. The set of the XPath 2.0 expressions have been determined that can be used within a predicate is different from the set of restriction facets that are used within an unnamed simple type. The mappings between these two sets of expressions and are presented in Table 4.1.

	Constraining facet	**XPath 2.0 function**
XML Schema 1.0	length	fn:string-length(.)= v
	minLength	fn:string-length(.)> v
	maxLength	fn:string-length(.)< v
	pattern (reg.ex.)	fn:matches(.,v)
	enumeration	op:union
	whitespace – preserve	no need for predicate
	whitespace – replace	fn:matches(. , '[^\t\n\r]*')
	whitespace – collapse	fn:matches(. , '(\w*[^\t\n\r]\w)+([^\t\n\r]\w+)')
	maxInclusive	.<= v
	maxExclusive	.< v
	minInclusive	.>= v
	minExclusive	.> v
	totalDigits	not supported
	fractionDigits	fn:matches(. , '^[-+]?\d+(\.\d{v})?$')
XML Schema 1.1	maxScale	not supported
	minScale	not supported
	assertions	not supported
	explicitTimezone – required	fn:matches(. , '.*Z')
	explicitTimezone – prohibited	fn:matches(. , '[^Z]*')
	explicitTimezone – optional	no need for predicate

Table 4.1 Mappings between constraining facets and XPath 2.0 functions

The mappings described in Table 4.1 are created taking in to account the form of the data that follow the constraining facets of the second column of the table. For example, an unnamed simple type with integer as the base type and constraining facet the expression `minExclusive=900`

(see the example of Figure 4.9) is mapped with the operator >=900, as the of this expression state that the value of the instances of the unnamed simple type should be greater than or equal to 900.

- The semantics of the constraining facets guided the mapping process with the XPath 2.0 functions and are described as follows:
- The facets **length**, **minLength** and **maxLength** are referring, respectively, to strings of exact length of the facet value, of minimum length equal to the facet value and of maximum length of the facet value.
- **The pattern** facet includes a regular expression, which limits the value of data and is represented by the matches function of XPath2.0.
- The facets **maxInclusive**, **maxExclusive**, **minInclusive**, and **minExclusive** set boundaries at the datatype values. In Figure 4.10 are presented the semantics and the relations of these arithmetic limitations on the number line.

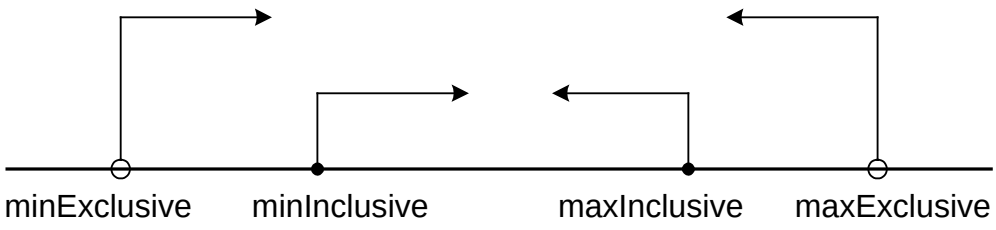

Figure 4.10 MinExclusive, minInclusive, maxInclusive and maxExclusive relations

Whitespace specifies how should be handled the while space in types derived from string. The value whiteSpace must be one of {preserve, replace, collapse}. In case of the preserve value no normalization is done, the value is not changed and consequently there is no need of having any predicate to limit the XPath. In case of replace, all the occurrences, inside the string, of tab, line feed and carriage return are replaced with space. Thus the resulting string may contain one or more spaces but no tabs, linefeeds and carriage returns. The regular expression `[^\t\n\r]*,` as is used shown in Table 4.1, to match that pattern of string values. In case of collapse the processing implied by replace, is first performed contiguous sequences of spaces are collapsed to a single space, and any space at the start or end of the string is removed. The regular expression `(\w*[^\t\n\r]\w)+([^\t\n\r]\w+),` is used as shown in Table 4.1, to match that pattern of string values.

FractionDigits is the maximum number of decimals digits for a datatype derived from decimal. For example, let fractionDigits have a value 4. Some valid decimals would be the numbers 3.12, 5.1235. The number 32.12349 is invalid. As presented in Table 4.1, this issue is handled using the regular expression `^[-+]?\d+(\.\d{v})?$,` where v is the value of the fractionDigits facet, for determining which decimals follow the restriction of fractionDigits.

Example

In Figure 4.9 is presented a simple type comprised of a union of unnamed simple types. Let X be the XPath set of the "Salary" simple type according to Table 4.1, where:

X= /Persons/Employee/Salary

Then the XPaths of the nested unnamed simple types of the Salary simple type are generated as is shown in Table 4.2.

XML Schema Simple Type definition	XPath
<xs:simpleType> <xs:restriction base="xs:int"> <xs:minExclusive value="900"/> </xs:restriction> </xs:simpleType>	/Persons/Employee/Salary[.>900]
<xs:simpleType> <xs:restriction base="xs:int"> <xs:minInclusive value="950"/> </xs:restriction> </xs:simpleType>	/Persons/Employee/Salary[.>=950]
<xs:simpleType> <xs:restriction> <xs:simpleType> <xs:restriction base="xs:int"> <xs:minInclusive value="950"/> </xs:restriction> </xs:simpleType> <xs:maxInclusive value="2000"/> </xs:restriction> </xs:simpleType>	/Persons/Employee/Salary[.>=950] [.<=2000]

Table 4.2 XPaths for nested simple types

Moreover, the base type of a restriction is another constraint on the value space of a simple type. In case that the base type of restrictions' of nested simple types are different pair wise, the predicates of the XPath expressions must contain a regular expression in order to limit the matching elements according to the type from which the unnamed simple type is derived. In Table 4.3 is presented the equivalent regular expressions of every built-in type in order to use this knowledge during the mapping of unnamed simple types to XPath expressions within predicate expressions.

Datatype	Pattern
xs:float	(\+|-)?([0-9]+(\.[0-9]*)?|\.[0-9]+)([Ee](\+|-)?[0-9]+)?|(\+|-)?INF|NaN

xs:decimal	(\+|-)?([0-9]+(\.[0-9]*)?|\.[0-9]+)
xs:boolean	^(true|false|0|1){1}$
xs:string	.
xs:int	[\-+]?[0-9]+ .<=2147483647 .>=-2147483648
xs:long	[\-+]?[0-9]+ .<=9223372036854775807 .>=-9223372036854775808

Table 4.3 Built-in datatypes matching patterns

According to Table 4.3, the first example (row) of Table 4.2 is transformed to:

/Persons/Employee/Salary[.>900][.<=2147483647]

In the example presented above, the pattern `[\-+]?[0-9]+` was left out as this regular expression ensures that the value is a number with or without a sign but this fact is implied using the arithmetic comparison operators. Furthermore, the greater than operator of xs:int was left out, as it overlaps with the pattern used to limit the value of the simple type as it is specified by iths constraining facet. Any base type and any unnamed simple type are threated in the some way.

The algorithm **find Unnamed Simple Type XPaths**, shown in Figure 4.11, has been developed taking into account the above objerversion.

Find Unnamed Simple Type XPaths Algorithm

Input: Unnamed Simple Type Name *datatypeName*

Output: Array of Xpaths *xpaths*

1. **for each** *element* in **substring**(*datatypeName*,"_")
2. **if** *element* is a Named Simple Type
3. *namedSt* ← *element.Name*

```
4.          break
5.      end if
6.  end for
7.  unnamedPosition ← 1
8.  unnamedPosition ← substring-after (datatypeName,"UNType_")
9.  for each st in DOM
10.     if st.Name=namedSt
11.         restrictionNode ← st [unnamedPosition].append("/restriction")
12.         tmpPredicate ← XPathPredicateGenerator (restrictionNode)
13.         for each stXpath in st.Xpaths
14.         xpaths.add ( stXpath.append(tmpPredicate) )
15.         end for
16.     end if
17.     break
18. end for
19. return xpaths
```

Figure 4.11 The find Unnamed Simple Type Xpaths Algorithm

The **XPath Predicate Generator** algorithm exploits the theoretical base, which was discussed previously in this subsection, about the predicates that can be appended to the paths of a named simple type in order to locate instances of the unnamed simple types declared within this named simple type. The XPath Predicate Generator algorithm is presented in Figure 4.12.

Xpath Predicate Generator Algorithm

Input: Simple Type node *simpleType*

Output: XPath predicates list *predicateList*

```
1.  ConstrHashTable.put('maxInclusive','<=')
2.  ConstrHashTable.put('maxExclusive','<')
3.  ConstrHashTable.put('minInclusive','>=')
4.  ConstrHashTable.put('minExclusive','>')
5.  ConstrHashTable.put('length','=')
6.  ConstrHashTable.put('minLength','>')
```

```
7.  ConstrHashTable.put('maxLength','<')
8.  ConstrHashTable.put('required','.*Z') //timezone
9.  ConstrHashTable.put('prohibited','[^Z]*') //timezone
10. ConstrHashTable.put('replace','[^\t\n\r]*') //whitespace
11. ConstrHashTable.put('collapse','(\w*[^\t\n\r]\w)+([^\t\n\r]\w+)') //whitespace
12. restriction ← simpleType.restriction
13. Let predicateList be a list     // A list to save the predicates that correspond to the constraints.
14. if (restriction.base = null)
15.     predicateList.add(XPathPredicateGenerator(restriction.append("/simpletype/restriction")))
16. end if
17. for each Constraint in restriction.Constraints
18.     Select case Constraint.name
19.     case (maxInclusive|maxExclusive|minInclusive|minExclusive)
20.         tmpPredicate ← concat("[.",ConstrHashTable(Constraint.name),Constraint.value,"]")
21.     case (length|minLength|maxLength)
22.         tmpPredicate ← concat("[string-length(.)", ConstrHashTable(Constraint.name),
            Constraint.value,"]")
23.     case (fractionDigits)
24.         tmpPredicate ← concat("[matches(.,'^[-+]?\d+(\.\d{",Constraint.value,"})?$]")
25.     case (pattern)
26.         tmpPredicate ← concat("[matches(.,'",Constraint.value,"')]")
27.     case (whitespace|explicitTimezone)
28.         if (ConstrHashTable(Constraint.value)!=null)
29.             tmpPredicate ← concat("[matches(.,'",ConstrHashTable(Constraint.value),"')]")
30.         end if
31.     case (enumeration)      // enumeration has more than one value
32.         tmpRegex ← "" //initialize tmpRegex variable
33.         for each v in Constraint.values
34.         if (first iteration)
35.         tmpRegex ← v
36.         else
37.         tmpRegex ← concat(tmpRegex,"|",v)
38.         end if
39.         end for
40.         tmpPredicate ← concat("[matches(.,'",tmpRegex,"')]")
```

41. **end Select**
42. *predicateList*.**add**(*tmpPredicate*)
43. **end for**
44. **return** *predicateList*

Figure 4.12 XPath Predicate Generator algorithm

Consider the XML Schema of Figure 4.9, which describes two unnamed simple types nested in the "Salary" simple type. The "Salary" Simple Type of the Schema of figure is mapped to the homonym owl:Datatype and to the Xpath /Persons/Employee/Salary as presented in the example of figure. The first unnamed simple type of the "Salary" union is named "NS_Salary_UNType_1" according to the rules and algorithms presented in section 3.3.2.4. This unnamed simple type is given as input to the findUnnamedSimpleTypeXpaths and XPathPredicateGenerator algorithms. The application on these algorithms to the unnamed simple type results in the XPath set that the "NS_Salary_UNType_1" is mapped to. The XPath set is used by the mappings binder component in order to create the needed instance of the map:Datatype in the borders of the mapping file as shown in Figure 4.13.

```
<map:Datatype name="NS_Salary_UNType_1">
      <map:Xpath>/Persons/Employee/Salary[.>900]</map:Xpath>
</map:Datatype>
```

Figure 4.13 Example of Dataype mapping information, that represents an unnamed simple type

At this point it is worthwhile to mention the benefit of supporting such mappings in this thesis. The user formulates queries being aware of the main ontology that corresponds to the underlying xml schema of an xml repository. Consequenlty, the user can use the datatypes of the main ontology in his queries without taking in to consideration if the datatype represents a named or an unnamed simple type.

4.3.4. Mappings for XML Schema identity constraints

As already mentioned in sections 2.2.2 and 2.3.6 there are two categories of identity constraints. The first category refers to the ID/IDREF/IDREFS built-in simple types and the second one are the key, keyref and unique.

The identity constraints ID-IDRef-IDRefs are derived types from the XML's DTDs and can be used as the type of any attribute or element. According to the XS2OWL transformation model, the attributes and elements are represented, respectively, by datatype and object properties.

Consequently, there is no extra information to store about their in an XML Schema and there is no need to enrich the mapping document with anything else as far as the ID/IDREF mechanism is concerned.

On the other hand, the second identity constraint category, which includes key-keyref, and unique is not a built-in type and is used in addition to having a type and not instead of it. These identity constraint constructs can be declared within an element (see sections 2.2.2 and 2.3.6). The XML Schema identity constraints can now be accurately represented in OWL 2.0 syntax and consequently in the XS2OWL 2.0. Thus, one of the extensions of SPARQL2XQuery 2.0 compared with SPARQL2XQuery 1.0 is the sound support of identity constraints by exploting the main and mapping ontologies, automatically generated by the XS2OWL 2.0 transformation model.

The added value SPARQL2XQuery 2.0 over SPARQL2XQuery 1.0 is the ability of reasoning on entities that are referenced by other entities via the key-keyref mechanism. SPARQL2XQuery 1.0 could answer queries including properties that particapaied in key-keyref pairs of identity constrains with remarkable limitations. In particular, it could acquire only information about the referring entity in the context of the Range set of the object property that refers to that entity. That could be only a data value of some type and not a node that has a complex structure available for further reasoning, e.g. “PersonType” a node.

The mapping specification proscess for the Identity Constraints is described in the Figure 4.14.

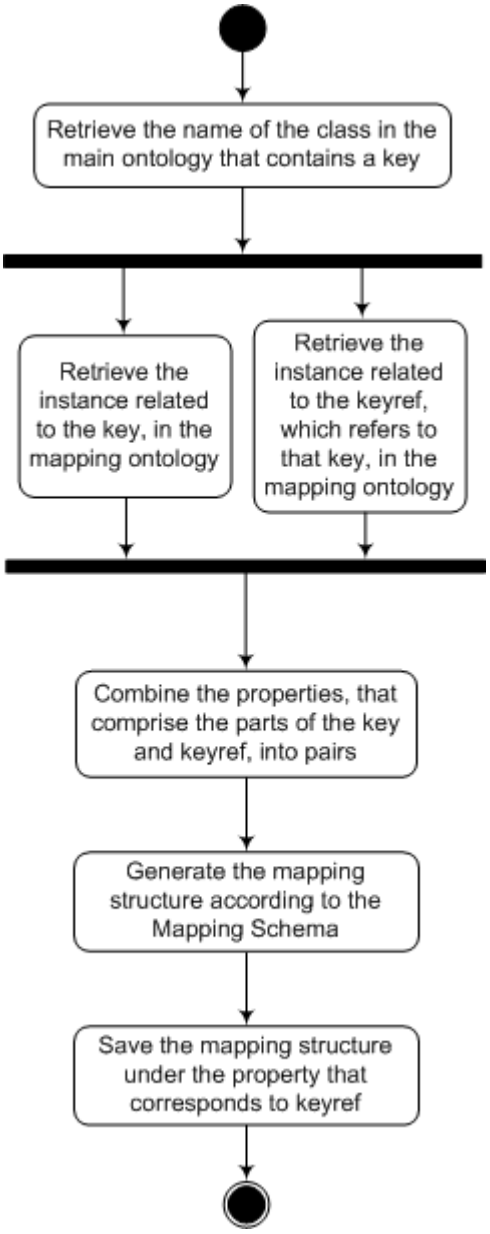

Figure 4.14 Mapping specification of the Identity Constraints

According to Figure 4.6, the following activities take place during the mapping specification of the identity constraints:

Retrieve tha name of the class that contains a key

In an earlier section, was presented the information being captured, through the parsing of main ontology, about the identity constraints (see Figure 4.4). At this step of mappings specification about identity constraints, is being retrieved the name of the classes that contain structures

representing the HasKey axiom. Activity diagram, of the figure presented above, shows in brief which activities are taking place during the mapping session in order to create usable and optimal mappings about identity constraints.

Retrieve the individual that represents the key constraint in the mapping ontology

In this activity the captured information about the mapping ontology is exploited to retrieve the "IdentityConstraintInfoType" individual that corresponds to the class that has just been retrieved in the previous step. This individual contains among others, the set of properties taking place in the combined key or the property comprising the key in case of a single property key.

Retrieve the related individual that represents the keyref constraint in the mapping ontology

This activity, in parallel with the previous one, searches among the "IdentityConstraintInfoType" individuals and retrieves the one that represents the keyref that refers to the current key. This individual contains the set of properties taking place in the combined keyref (or the property comprising the keyref in case of a single property keyref).

Mapping the properties of key-keyref

In the previous activities, the properties that represent the key and keyref constraints were retrieved from the mapping ontology. This activity aims to produce one-to-one mappings between the properties that represent key and the properties that represent keyref. These pairs of properties represent the XPath expressions pairs of key-keyref that are evaluated in order to identify the tuples of the referenced key. This approach makes feasible the correct and without uncertainty matching of the identity constraint datatype or object properties that will be used later on to generate XQueries with references using the key-keyref XML Schema mechanism.

Generate the mapping structure

At this step of the process, the retrieved information should form a meaningful construct. This need is implemented by the "IC" element, which contains the "onClass" attribute. Attribute "onClass" holds the class that repsresents the key in the main ontology. Furthermore, the "IC" element contains a set of "PropertyPair" elements that represent the datatype or object property pairs that represent the key and keyref constraints. For each pair of key and keyref constraints a "PropertyPair" element is defined. Consequently, the "PropertyPair" element includes two elements, one for the name of the property that corresponds to the keyref and one for the name of the property that corresponds to the referenced key.

This is a straightforward approach for the optimal and sound exploitation of the identity constraints in the SPARQL2XQUERY translation process that follows. Also, there is no need to include any paths about the properties that take place in the identity constraints as this need is covered by the definitions of the mappings about the object and datatype properties in the same mapping file.

Mapping construct

Finally, the mapping construct created in the previous step is stored under the object property that correspond to the related keyref. This way, is made available all the required information for the translation of SPARQL queries containing the object property, which represents the keyref selector in the main ontology and makes possible the reasoning as it should be in cases of identity constraints existance.

Moreover, it is worthwhile to mention that this approach of the mapping process encourages the Domain Expert (see Figure 2.48) to make use of the key-keyref mechanism capability and provide the Query translator with that kind of information in an easy and unambiguous syntax.

Example

The following figures present a) the excerpt of an XML Schema with key and keyref elements; b) the main ontology representation of these elements in OWL 2.0 syntax; c) the mapping ontology representation of these elements; and d) the mapping file that was generated to handle the key-keyref semantics. In particular:

Figure 4.15 shows the root element of the XML Schema followed by the data underlying the XML DB repository. The key identity constraint is a combined key, since it contains multiple fields.

Figure 4.16 shows the OWL class and the object property that represent, respectively, the key constraint and the keyref constraint in the main ontology.

Figure 4.17 shows the “IdentityConstraintInfoType” individuals that represent the key and keyref constraints in the mapping ontology.

Figure 4.18 shows an excerpt of the resulting mapping file generated according to the specification process discussed in this subsection.

```
<xs:element name="Persons" type="PersonsType">
      <xs:key name="IDKEY">
            <xs:selector xpath="Person"/>
            <xs:field xpath="firstname"/>
            <xs:field xpath="lastname"/>
      </xs:key>
      <xs:keyref name="KNOWS" refer="IDKEY" >
            <xs:selector xpath="Person/Knows"/>
            <xs:field xpath="Name"/>
            <xs:field xpath="Surname"/>
      </xs:keyref>
</xs:element>
```

Figure 4.15 Excerpt of an XML Schema with Key and Keyref constraints

```
<owl:Class rdf:ID="PersonType">
      ...
      <owl:hasKey rdf:parseType="Collection">
            <rdf:Description rdf:about="#firstname__xs_string"/>
            <rdf:Description rdf:about="#lastname__xs_string"/>
      </owl:hasKey>
      <rdfs:label>PersonType</rdfs:label>
</owl:Class>

<owl:ObjectProperty rdf:ID="Knows__RefType">
      <rdfs:range rdf:resource="#PersonType"/>
      <rdfs:domain rdf:resource="#PersonType"/>
      <rdfs:label>Knows</rdfs:label>
</owl:ObjectProperty>
```

Figure 4.16 Representation of the key and keyref constraints in the main ontology

```
<ox:IdentityConstraintInfoType rdf:ID="Persons_IDKEY_PersonsType_ui">
      <ox:classID>PersonType</ox:classID>
      <ox:ICName>IDKEY</ox:ICName>
      <ox:HostELementID>Persons</ox:HostELementID>
      <ox:selectorPath>Person</ox:selectorPath>
      <ox:ICProperties>
            <ox:ICPropertyType>
                  <ox:Property>firstname__xs_string</ox:Property>
                  <ox:fieldPath>firstname</ox:fieldPath>
            </ox:ICPropertyType>
            <ox:ICPropertyType>
                  <ox:Property>lastname__xs_string</ox:Property>
                  <ox:fieldPath>lastname</ox:fieldPath>
            </ox:ICPropertyType>
      </ox:ICProperties>
      <ox:constraintType>key</ox:constraintType>
</ox:IdentityConstraintInfoType>
<ox:IdentityConstraintInfoType rdf:ID="Persons_KNOWS_PersonsType_ui">
      <ox:opID>Knows__PersonType</ox:opID>
      <ox:ICName>KNOWS</ox:ICName>
      <ox:RefersTo>IDKEY</ox:RefersTo>
      <ox:HostELementID>Persons</ox:HostELementID>
      <ox:selectorPath>Person/Knows</ox:selectorPath>
      <ox:ICProperties>
            <ox:ICPropertyType>
                  <ox:Property>Name__xs_string</ox:Property>
                  <ox:fieldPath>Name</ox:fieldPath>
            </ox:ICPropertyType>
            <ox:ICPropertyType>
                  <ox:Property>Surname__xs_string</ox:Property>
                  <ox:fieldPath>Surname</ox:fieldPath>
            </ox:ICPropertyType>
      </ox:ICProperties>
      <ox:constraintType>keyref</ox:constraintType>
</ox:IdentityConstraintInfoType>
```

Figure 4.17 Representation of key and keyref in the mapping ontology

```
<map:ObjectProperty name="Knows__RefType">
      ...
      <map:IC onClass="PersonType">
            <map:PropertyPair>
                  <map:keyrefMember>Name__xs_string</map:keyrefMember>
                  <map:keyMember>firstname__xs_string</map:keyMember>
            </map:PropertyPair>
            <map:PropertyPair>

      <map:keyrefMember>Surname__xs_string</map:keyrefMember>
                  <map:keyMember>lastname__xs_string</map:keyMember>
            </map:PropertyPair>
      </map:IC>
</map:ObjectProperty>
```

Figure 4.18 Excerpt of the generated mapping file for the constructs of Figure 4.15 and Figure 4.16

4.4. Query Translation

4.4.1. Determination of the Variable Types

The determination of the Variable Types, which is outlined in this section, has not changed since the previous version of the SPARQL2XQuery framework.

This activity identifies the type of every variable referenced in a *UF-GP* (Union-Free GP Definition 2.19) in order to determine the form of the results and, consequently, the syntax of the Return clause in XQuery. Moreover, the variable types are used by the "Onto-triple processing" (section 4.4.2) and the "Variable Binding" activities (section 4.4.3). Finally, this activity performs consistency checking in variable usage in order to detect possible conflicts (e.g., the same variable name may be used in the definitions of variables of different types in the same *UF-GP*). In such a case, for efficiency reasons the *UF-GP* is not going to be translated, because it is not possible to be matched with any RDF dataset.

Definition 4.7. Variable Types. The following variable types are defined for the variable type determination:

- The Class Instance Variable Type (*CIVT*), which represents class instance variables.
- The Literal Variable Type (*LVT*), which represents literal value variables.
- The Unknown Variable Type (*UVT*), which represents unknown type variables.
- The Data Type Predicate Variable Type (*DTPVT*), which represents data type predicate variables.

- The Object Predicate Variable Type (*OPVT*), which represents object predicate variables.
- The Unknown Predicate Variable Type (*UPVT*), which represents unknown predicate variables.

The following sets are defined:

- the Data Type Properties Set (*DTPS*), which contains all the data type properties of the ontology
- the Object Properties Set (*OPS*), which contains all the object properties of the ontology
- the Variables Set (*V*), which contains all the variables that are used in the *UF-GP*
- the Literals Set (*L*), which contains all the literals referenced in the *UF-GP*

The determination of the variable types is based on the rules presented below. These rules are applied iteratively for each triple in the given *UF-GP*. A subset of these rules, which are used to determine the type (T_X) of a variable x, are presented below:

Let *S P O* be a triple pattern.

- If $S \in V \Rightarrow T_S = CIVT$. If the subject is a variable, then the variable type is CIVT.

- If $P \in DTPS$ and O ϵ V $\Rightarrow T_O = LVT$. If the predicate is a datatype property and the object is a variable, then the type of the object variable is LVT.

- If $P \in OPS$ and $O \in V \Rightarrow T_O = CIVT$. If the predicate is an object property and the object is a variable, then the type of the object variable is CIVT.

- $T_P = DTPVT \Leftrightarrow T_O = LVT \mid P,O \in V$. If the predicate variable is of DTPVT type, then the type of the object variable is LVT. The inverse also holds.

- $T_P = OPVT \Leftrightarrow T_O = CIVT \mid P,O \in V$. If the predicate variable is of OPVT type, then the type of the object variable is CIVT. The inverse also holds.

- If $O \in L$ and $P \in V \Rightarrow T_P = DTPVT$. If the object is a literal value, then the type of the predicate variable is DTPVT.

- If S, P, $O \in V$ and $T_P = UPVT$ and $T_O = UVT \Rightarrow T_P = UPVT$ and $T_O = UVT$. If a triple has subject, predicate and object variables, the predicate variable type is *UPVT* and the object variable type is *UVT*, no change is needed since they cannot be specified.

The unknown variable types *UTV* and *UPTV* do not result in conflicts in case that a variable has been also defined to have another type since they can be just ignored. All the variable types are initialized to *UVT* or *UPVT*.

4.4.2. Onto-triple processing

The Onto-Triple Processing, i.e. the processing of triples which is composed of factors that belong to the RDF/S[W3C/RDF(S)] and OWL[W3C/OWL2] vocabularies (see Definition 4.8) is described in this section. This process follows the Variable Type Determination process and its input is a Union-Free graph pattern in order to bind the variables with XPaths of the xml document that were captured by the onto-triples analysis. This set of bindings is used for the initialization of the Variable Binding process (see subsection 4.4.2). They are used as initial bindings in the translation algorithms, as these paths depict the requirements of the xml data according to the ontology semantics that are mentioned in a graph pattern.

Definition 4.8 Onto-triple. An Onto-triple is a triple which refers to the ontology structure and/or semantics. Formally, the only requirement for a triple pattern to be an onto-triple is the existence of at least one construct, among the triple parts, that belongs to the RDF/S and OWL vocabularies.

The focus of this thesis is on the new onto-triple patterns that can be supported by the SPARQL2XQuery 2.0 framework. These patterns have been introduced due to the new features of OWL 2.0.

There are two special cases of onto-triples that are treated in a different way compared to other cases of onto-triple patterns. These are the onto-triples whose predicate is the rdf:type or the rdfs:Datatype.

Rule 4.1. The onto-triples having as predicate the rdf:type property are not processed by the SPARQL engine in those cases that the object is one of the following: variable, IRI, rdf:Property, owl:ObjectProperty, owl:DatatypeProperty or rdfs:Datatype. In this case, the subect of the triple can be mapped directly to XML data.

Rule 4.2. The onto-triples having as predicate the rdfs:Datatype property are not processed by the SPARQL engine in cases that the object is one of the following: variable, IRI. In this case, the subect of the triple can be mapped directly to XML data.

In Rules 4.1 and 4.2 is specified the onto-triple processing of triples that contain the rdf:type or rdfs:Datatype properties in their predicate were specified. These rules are taken in consideration through the examples and cases presented below.

Manipulation of onto triple with property rdf:type

It was discussed in Rule 4.1 that in some cases it is feasible to execute the query in order to create the bindings of the triple variables. This happens in the following query types:

a. ?x rdf:type IRI

In this case the IRI refers to a named class. Thus, the variable x indicates the instances of the class referred by the IRI. Consequently, the variable x is mapped to the paths that are associated with the IRI referred class according to the mapping file that was generated in the mappings specification activity (see section 4.3).

Example 4.1 The following query asks for all the instances of the class "Person_Type".

```
SELECT ?x
WHERE {?x rdf:type :PersonType .}
```

From the mapping specification activity, the following paths have been mapped to the class "PersonType":

$\mathbf{X_{PersonType}}$ = {/Persons/Person, /Persons/Employee}

At this moment, Rule 4.1 is applied. Consequently, this XPath set is equivalent to the one that corresponds to the x variable:

$\mathbf{X_x}$ = {/Persons/Person, /Persons/Employee}

b. ?x rdf:type rdfs:Datatype

In this case the object refers to the set of the Datatypes (defined by the schema author) of the main ontology. Thus, the x variable represents the Datatype instances. Consequently, x is mapped to the XPaths that are associated with the Datatype set, according to the mapping file generated by the mapping specification activity (see section 4.3).

Example 4.2 The following query asks for the instances of all the Datatypes.

```
SELECT ?x
WHERE {?x rdf:type rdfs:Datatype .}
```

The x variable will be mapped to the union of the path set that have been associated with Datatypes in the mapping specification activity. Thus, the following paths have been mapped to the x variable:

$\mathbf{X_x}$ = {/Persons/Person/Age, /Persons/Employee/Age}

This set of paths represents instantiations of the author defined Simple Types in the XML Schema we are referring to and not instantiations of built-in Simple Types.

Manipulation of onto triple with property rdfs:Datatype

It was already mentioned in Rule 4.1, that in some cases it is necessary to execute the query in order to create the bindings of triple variables. This happens in the following query types:

a. ?x rdfs:Datatype IRI

The IRI refers to a named datatype. Thus, the x variable represents the instances of the datatype referred by the IRI. Consequently, the x variable is mapped to the XPaths that are associated with the IRI referred datatype according to the mapping file that was generated by the mapping specification activity (see section 4.3).

Example 4.3 The following query asks for the instances of the datatype "validAgeType".

```
SELECT ?x
WHERE {?x rdfs:Datatype : validAgeType.}
```

From to the mappings specification process, the following paths have been mapped to the datatype "validAgeType":

$\mathbf{X_{validAgeType}}$ = {/Persons/Person/Age, /Persons/Employee/Age}

At this moment Rule 4.2 is applied. Consequently, this XPath set is equivalent to the one of x variable:

$\mathbf{X_x}$ = {/Persons/Person/Age, /Persons/Employee/Age}

b. ?x rdfs:Datatype ?o

In this case the o variable represents datatypes. Consequently, the x variable is mapped to the XPaths that are associated with the Datatype set, according to the mapping file generated by the mapping specification activity.

Example 4.4 The following query asks for the instances of all the Datatypes.

```
SELECT ?x
WHERE {?x rdfs:Datatype ?o .}
```

The triple ?x rdfs:Datatype ?o is reduced to the triple ?x rdf:type rdfs:Datatype, which is the second case (b) of the rdf:type manipulation. Thus, the x variable will be mapped to the union of the XPath sets that have been associated with Datatypes according in the mappings specification activity:

$\mathbf{X_x}$ = {/Persons/Person/Age, /Persons/Employee/Age}

The onto-triple processing before the variable binding process aims to generate a set of initial bindings for the variables that are used with the onto-triples within the graph pattern of the SPARQL query.

4.4.3. Variable Binding

In this section is described the "Variable Binding" activity, which is applied to a BGP (Basic Graph Pattern), i.e. a sequence of triple patterns (see section 2.10.4, Definition 2.7 for details). The Variable binding process aims to declare the bindings between the BGP variables and a set of XML paths of the XML document that contains the data of the XML DB repository. The binding of variables with paths results in a dictionary of variables and their corresponding set of paths. This dictionary is essential for the correct and sound execution of the BGP translation process (see section 4.4.4) to the equivalent XQuery syntax, as it is necessary to know the possible paths that lead to the instance represented by the variable in the XML document.

The Variable binding process exploits the results of the onto-triple processing. The variable bindings referring to onto-triples are used during the initialization session of the Variable binding activity. These bindings are reffered in this section as initial bindings.

The RDF Data Model is a Graph-Based model that consists of a sequence of triple patterns. Every triple pattern represents a relationship between two instances and appears in the RDF model as the link between two nodes. This link is represented by a vector that starts from the subject of the triple pattern and ends at the object of the triple pattern. Every part of the RDF graph has a name (label). On the other hand, the XML Data model is a Tree-based model with named leaves and unnamed branches. Thus, the bindings of BGP variables to XML paths are mandatory in order to evaluate the BGP over XML data and not over RDF graphs.

Moreover, the Variable Binding process does not take into consideration any FILTER elements of the BGP, as these elements do not contain anything useful for the binding of variables to XML paths but only restrictions and conditions between variables and values. Also, the Variables Binding process ignores the Onto-triples, because they were used to extract the initial bindings that will be exploited by the Variable Binding process during the initialization session together with the IRI bindings, that were specified in the parsing of the Ontology and the related XML Schema.

In the rest of this section are represented the path bindings for every single triple pattern and for multiple triple patterns (subsection 4.4.2.1), the variable binding process algorithm (subsection 4.4.2.2) and any dependencies and relations between the path sets of the triple patterns (subsection 4.4.2.3).

4.4.3.1. Variable Binding fundamentals

The notations used and the necessary definitions and rules for the variable binding activity are presented in this section.

Rule 4.3. A triple **(S, P, O) ϵ (I ∪ B ∪ V) x (I ∪ V) x (I ∪ B ∪ L ∪ V)** is called Triple pattern according to Definition 2.2. Moreover, assume that the refered ontologies do not contain class instances. Thus, the above deifinition is formally presented below:

$$(S, P, O) \in (B \cup V) \times (I \cup V) \times (B \cup L \cup V)$$

A Triple pattern represents a part of the RDF graph and is related to a part of the xml tree representation. Thus, for every part of the triple pattern there exists a path in the xml tree. Let **S P O** be a triple pattern:

- Let $\mathbf{X_S}$ be the set of paths that correspond to the subject **S**.
- Let $\mathbf{X_P}$ be the set of paths that correspond to the predicate **P**. Moreover, let $\mathbf{X_{pD}}$ be the domain of X_P and $\mathbf{X_{pR}}$ be the range of X_P.
- Let $\mathbf{X_O}$ be the set of paths that correspond to the object **O**.

The relations among the XPath sets of the subject, predicate and object are summarized in the Rule 4.4 presented below.

Rule 4.4. The set of the paths related to the subject of triple ($\mathbf{X_S}$) equals to the set of paths that is related to the domain of the predicate ($\mathbf{X_{pD}}$). Moreover, $\mathbf{X_S}$ equals to the result set of paths that is related to the range of the predicate when the Parent operator is applied to it ($\mathbf{X_{pR}^P}$). Also, $\mathbf{X_S}$ equals to the result set of paths that is related to the predicate when the Parent operator is applied to it ($\mathbf{X_P^P}$). Finally, $\mathbf{X_S}$ equals to the result set of paths that is related to the object when the Parent operator is applied to it ($\mathbf{X_O^P}$).

$$\mathbf{X_S = X_{pD} = X_{pR}^P = X_P^P = X_O^P}$$

Definition 4.9 Shared Variable. A variable contained in a Union Free Graph Pattern is called a Shared Variable if it is referenced in more than one triple patterns of the same Union Free Graph Pattern regardless of its position in those triple patterns.

In case of shared variables, the algorithm tries to find, using the XPath Set Operators (see Section 4.3.1), the maximum set of bindings that satisfy this relation for the entire set of triple patterns (i.e., the entire BGP). Once this relation holds for the entire BGP, the result is that all the (XML) instances that satisfy the BGP have been addressed.

4.4.3.2. Variable Binding Algorithm

This subsection presents the algorithm that is responsible for binding every BGP variable to an XPath set. The Variable Binding algorithm exploits the relations between the set of XPaths that were described in section 4.4.2.1. Moreover, the algorithm handles any ambiguity that occurs from the different sets of XPaths that appear as possible bindings for the same variable due to conjunctive triple patterns. The resulting set of paths is the broader possible set.

The Variable Binding algorithm ignores any variables of LVT type (i.e. variables of Literal type), because the application of the set operators, that are used by the algorithm, on the path sets result in incorrect path sets.

The algorithm, presented in Figure 4.19, takes as input a basic graph pattern (*BGP*) as well as a set of Initial Bindings (*initBindings*) and the variable types (*varTypes*) determined by the Determination of Variable Types activity. These initial bindings are those produced by the Onto-Triple processing and initialize the bindings of the algorithm (line 1). Then, the algorithm performs an iterative process (lines 4-6) where it determines, at each step, the bindings of the entire BGP (triple by triple). The determination of the bindings of a single triple is implemented within the DetermineBindings() function (line 5), which applies the Variable Binding Rules, that are presented comprehensively in the rest of this subsection, to a single pattern. This iterative process continues until the bindings for all the variables found in the successive iterations are equal (line 7). This means that no further modifications in the variable bindings are to be made and that the current bindings are the final ones. Thus, the Variable Binding algorithm guarantees that all the variables have been bound to the correct XPaths according to: (a) the structure of the ontology; (b) the structure of the XML Schema; and (c) the mappings between them.

Variable Binding Algorithm

Input: Basic Graph Pattern *BGP*, Initial Bindings *initBindings*, Variable Types *varTypes*

Output: Variable Bindings *bindings*

1. *bindings* ← *initBindings*
2. **repeat**
3. *oldBindings* ← *bindings*
4. **for each** *triple* in *BGP*
5. *bindings* ← **DetermineBinding** (*triple*, *oldBindings*, *varTypes*)
6. **end for**
7. **end if**
8. **until** (*oldBindings* = *bindings*)
9. **return** *bindings*

Figure 4.19 The Variable Binding Algorithm

Variable binding Rules

The DetermineBinding function (line5) that takes as input a single triple pattern, any existing bindings at the function call point and finally the dictionary that holds the types of the variables that appear in the BGP is used in Figure 4.19. The DetermineBinding function applies a set of rules to the input triple pattern in order to determine the set of paths which will be binded to it.

These rules are different, according to the combinaton of the variables types among the parts of the triple pattern. The triple patterns are classified in four types, as shown in definition 4.10.

Definition 4.10 Triple Pattern Types

- **VILTP** – A triple pattern whose subject is a variable, the predicate is an IRI and the object is a literal. $S \in V$, $P \in I$, $O \in L$.
- **VIVTP** – A triple pattern whose subject and object are variables and the predicate is an IRI. $S,O \in V$, $P \in I$.
- **VVLTP** - A triple pattern whose subject and predicate are variables and the object is a literal. $S,P \in V$, $O \in L$.
- **VVVTP -** A triple pattern whose subject, predicate and object are variables. $S,P,O \in V$.

For each triple pattern type is defined a different sequence of rules that lead to the determination of variables XPath sets. The contribution of this thesis in the previous work is the exploitation of literals with an optional datatype IRI or prefixed name (^^). In case of literals with a datatype prefixed name, e.g. “25”^^ns:validAgeType, this is declared within the main ontology, it allows exploiting the mappings of the datatype and determining the possible path sets for the subject and the predicate of the triple patterns.

The notations that are used in the Rules shown below are the following:

- The symbol “ ’ ” in XPath sets denotes the new bindings assigned to a set in each iteration.
- The symbol “←” denotes the assignment of a new value to a set.
- The notation “*Not Definable*” is used for variables of type LVT as explained above.

Note: All the XPath sets are considered to be initially set to null. In that case, the intersection operation is not affected by the null set. E.g., of *X={ null }* and *Y= { /a/b/c }* then X ∩ Y ={ /a/b/c }.

Consider the triple **S P O**.

If the triple pattern is of **VILTP** type then:

- $X_S' \leftarrow X_{pD} \cap X_S$

If the triple pattern is of **VIVTP** type then:

- $X_S' \leftarrow X_{pD} \cap X_S \cap (X_O)^P$
- If $P \in OPS$ then
    - $X_O' \leftarrow X_S' \oplus X_O$
- If $P \in DTPS$ then
    - X_O' *Non Definable* (as explained previously)

If the triple pattern is of **VVLTP** type then:

- $X_S' \leftarrow X_{pD} \cap X_S$ and $X_P' \leftarrow X_S' \oplus X_P$
- In case that the subject XPath set (X_S) and the predicate XPath set (X_P) have not been determined (are null), the union of the sets of all the mapped classes is assigned to the subject XPath set (X_S) :
  $X_S \leftarrow X_{C1} \cup X_{C2} \cup ... \cup X_{Cn}$, where X_{Ci} $(1 \leq i \leq n)$ is the XPath set corresponding to class C_i . Moreover, the union of the sets of all the mapped datatype properties is assigned to the predicate XPath set (X_P):
  $X_P \leftarrow X_{Pr1} \cup X_{Pr2} \cup ... \cup X_{Prk}$, where X_{Prj} $(1 \leq j \leq k)$ is the XPath set corresponding to the datatype Property Prj.
- In case that the predicate XPath set (X_P) has not been determined (is null), a set is assigned to the predicate XPath set (X_P) equal to the XPaths from all the datatype properties that their parent XPath is included to the subject XPath set:
  $X_P \leftarrow x_{Pr1} \cup x_{Pr2} \cup ... \cup x_{Prk}$ where $x_{Prj} \subseteq X_{Prj}$, $(x_{Prj})^P \in X_S$ and X_{Prj} $(1 \leq j \leq k)$ is the XPath set corresponding to the datatype property *Prj*.
- In case that the object literal has a datatype prefixed name (e.g. "25"^^ns:validAgeType), a set is assigned to the new subject Xpath set (X_S') the intersection of the parent XPath set of the named Datatype ($X_{ODatatype}$) and the subject XPath set (X_S):
  $X_S' \leftarrow (X_{ODatatype})^P \cap X_S$
  A set is also assigned to the new predicate XPath set (X_P') the intersection of the predicate

XPath set (X_P) and the set of the named Datatype:

$X_P' \leftarrow X_{ODatatype} \cap X_P$

If the triple pattern is of **VVVTP** type:

- $X_S' \leftarrow X_{pD} \cap X_S \cap (X_O)^P$
- $X_P' \leftarrow X_S' \oplus X_P$
  - If T_O = *CIVT* or T_O = *UVT* then
    - $X_O' \leftarrow X_P' \cap X_O$
  - If T_O = *LVT* then
    - X_O' *Non Definable* (as explained previously)
  - In case that the subject XPath set (X_S), the predicate XPath set (X_P) and the object XPath set (X_O) have not been determined (are null), a set is assigned to the subject XPath set (X_S) the union of the sets of all the mapped classes: $X_S \leftarrow X_{C1} \cup X_{C2} \cup ... \cup X_{Cn}$, where X_{Ci} ($1 \leq i \leq n$) is the XPath set corresponding to class *Ci*.
    - If T_P = *DTPVT*, a set is assigned to the predicate XPath set (X_P) the union of the sets of all the mapped datatsype properties:
      $X_P \leftarrow X_{Pr1} \cup X_{Pr2} \cup ... \cup X_{Prk}$, where X_{Prj} ($1 \leq j \leq k$) is the XPath set corresponding to datatype property *Prj*.
    - If T_P = *OPVT*, a set is assigned to the predicate XPath set (X_P) the union of the sets of all the mapped object properties:
      $X_P = X_o \leftarrow X_{Pr1} \cup X_{Pr2} \cup ... \cup X_{Prk}$, where X_{Prj} ($1 \leq j \leq k$) is the XPath set corresponding to object property *Prj*.
    - If T_P = *UPVT,* a set is assigned to the predicate XPath set (X_P) the union of the sets of all the mapped properties:
      $X_P \leftarrow X_{Pr1} \cup X_{Pr2} \cup ... \cup X_{Prk}$, where X_{Prj} ($1 \leq j \leq k$) is the XPath set corresponding to property *Prj*.
  - In case that the predicate XPath set (X_P) has not been determined (is null), a set is assigned to the predicate XPath set (X_P) equal to the XPaths from all the datatype properties or all the object properties or all the properties (for the properties it depends on the predicate variables), that their parent XPath is included in the subject XPath set:
    $X_P \leftarrow x_{Pr1} \cup x_{Pr2} \cup ... \cup x_{Prk}$ where $x_{Prj} \subseteq X_{Prj}$ and $(x_{Prj})^P \in X_S$, where x_{Prj} ($1 \leq j \leq k$) is :

- If T_P = *DTPVT* then the XPath set corresponding to the datatype property *Prj*.
- If T_P = *OPVT* then the XPath set corresponding to the object property *Prj*.
- If T_P = *UPVT* then the XPath set corresponding to the property *Prj*.

A rich set of examples is presented below in order to demonstrate the use cases of the rules already presented.

4.4.3.3. XPath set relations for Triple patterns

The Tree-based model of XML implies a very important relation between the data nodes of the tree and the paths that lead to them. Thus, among the triple pattern XPath sets exist important relations that can be exploited in the development of the XQuery expressions, using the correct for/let clauses, in order to correctly associate data that either have been bound to different triple pattern variables or are not defined by the Variable Binding algorithm, i.e. obejct variables of literal type. The most important relation among triple pattern XPath sets is the extension relation.

> **Definition 4.11 Extension Relation**. An XPath set A is an extension of an XPath set B if all the XPaths in A are descendants of XPaths of B. This relation can be achieved if the Append (/) XPath set operator is applied to the XPath set A having as right operand a set of nodes, and as a result the XPath set B, which will be an extension of A.

The extension relation holds for the results of the Variable Binding Algorithm (Subject-Predicate-Object Relation) and implies that the XPaths bound to the subjects are parents of the XPaths bound to the triple pattern predicates and objects.

In particular, rule 4.4 concludes to a formal relation between the XPath sets of subject, predicate and object:

Consider for instance the triple pattern ?X FirstName__xs_string ?Y (where X corresponds to Persons and Students and Y to their first name(s)). The variable bindings will result in two XPath sets: one for all the Persons and Students, and one for all the First Names. However, the association of persons and names still has to be done. This is performed by the BGP2XQuery algorithm exploiting the extension relation on XQuery variables and using the For and Let clauses.

4.4.4. Basic Graph Pattern Translation

This section describes the process of the Basic Graph Pattern Translation, that is the translation of the SPARQL Basic Graph Patterns (BGP) to semantically correspondent XQuery expressions, thus allowing the evaluation of a BGP over XML data. This process is based on the results of previous

activities, particularly the mapping determination of section 4.3, the variable binding of section 4.4.3 and the variables type determination of section 4.4.1.

The syntax of a SPARQL query has been described in section 2.10. The main and most challenging to translate part of the query, is the *where* clause. The *where* clause must be translated in XQuery syntax in order to create a sequence of XQuery expressions that articulate the same semantics with the SPARQL expressions. The where clause of the SPARQL query is a graph pattern (Definition 2.8). The simplest case of a Graph Pattern is the Triple Pattern and then the Basic Graph Pattern, which is comprised of triple patterns and filter expressions. This analysis clarifies the target of the necessary algorithm, which is the sound and complete translation of the BGP into XQuery expressions that will be afterwards evaluated over XML data.

The needs already discussed are covered by the development of the **BGP2XQuery Algorithm**, that is presented step by step in the rest of this section. The BGP2XQuery Algorithm has changed since the last version. The main objectives of the algorithm design include:

- The development of a general and comprehensible process for the translation of Basic Graph Patterns in XQuery syntax.
- Strict maintenance of the semantics during the translation.
- Development of agile and XQueries with minimal complexity.
- Development of XQueries with easily distinguishable mappings between them and the SPARQL queries.
- Development of optimized XQueries as far as the translation of the SPARQL semantics are concerned. The optimization is not XQuery oriented.
- Exploitation of any identity constraints and relations that appear in the XML Schema of the XML data repository.
- Development of XQuery expressions semantically equivalent to the SPARQL queries, thus allowing the evaluation of a BGP on a set of XML data.

The rest of this section presents the fundamentals of the BGP2XQuery algorithm (subsection 4.4.3.1), the BGP2XQUERY algorithm step by step (subsection 4.4.3.2), the set of changes applied on the BGP2XQuery algorithm in order to handle any issues that arise from the shared variables (subsection 4.4.3.3) and finally the structure of the XQuery result sets (subsection 4.4.3.4).

4.4.4.1. Fundamentals concepts for the Basic Graph Pattern to XQuery translation

The fundamental concepts for the BGP translation to XQuery syntax are presented in this section.

As far as the notations are concerned, the **name of a variable** is represented with the symbol N_V, where V is the referred variable.

Definition 4.12 Return Variables. Return Variables (*RV*) are those variables for which the given SPARQL query would return some information. The set of all the Return Variables of a SPARQL query constitutes the set $RV \subseteq V$, where V is the set of variables in the graph pattern of the query.

In particular, for each type of query is described how the *RV* set is modified in the following distinct query types:

- In case of a **SELECT** type query:
  RV is the set of the variables that appear in the SELECT clause of the query ($RV \subset V$). In case that the SELECT clause contains the wildcard (*) then RV is the whole set V of the variables in the graph pattern of the query ($RV = V$).

- In case of a **DESCRIBE** type query:
  The *RV* set follows the same rules with the SELECT type queries.

- In case of a **CONSTRUCT** type query:
  RV is the set of variables that are mentioned within the graph template except for the blank nodes of the graph template.

- In case of an **ASK** type query:
  There are no return variables, since the queries of this type return a literal “yes” or “no”.

Definition 4.13 Variable $doc. The variable $doc is defined as a variable which represents a text node or a sequence of text nodes. It is initialized with the expression Let $doc := fn:doc(URI) or Let $doc := fn:collection(URI), where URI represents the location of the XML document or the location of the collection of XML documents, respectively.

Note: Any variable that is defined in the algorithms of this thesis, e.g. *$doc*, is given a safe name in the generated XQuery syntax in order to prevent any conflict with the variables existing in the graph pattern of the query. In case that an assistant variable has the same name with a variable of the query graph pattern, the name of the former is changed by appending a serial number to the name.

Definition 4.14 For/Let XQuery Clause Representation. The for and let clauses are represented in the algorithms by the expressions *for $Var in expr* and *let $Var := expr*, respectively. The *fn* namespace prefix refers to the built-in XQuery functions while the *func* namespace prefix refers to the functions that defined in this thesis that are used within the definition of an XQuery.

Thus the XQuery Clause *XC* is comprised of two elements the Var and the expr, which are represented by the *XC.Var* and the *XC.expr* respectively. Moreover, the element type is used to

store the result that is returned by the For/Let Selector Algorithm (Figure 4.26) and is represented by the *XC.type*.

4.4.4.2. Basic Graph Pattern to XQuery Algorithm

The BGP2XQuery algorithm takes as input a basic graph pattern (*BGP*), the mappings between the XML Schema and the main ontology that were generated as described in section 4.3 (*mappings*), the variable bindings to XPaths that were specidfied as described in section 4.4.3 (*variableBindings*), the variable types (*varTypes*), the return variables (*RV*), and the SPARQL query type (*queryTYpe*). The output of the algorithm is a sequence of XQuery expressions semantically correspondent to the basic graph pattern (*BGP*) based on the input parameters.

The BGP2XQuery algorithm is responsible for the **orchestration of the translation process** that is comprised of a set of algorithms, which have to be executed in the right order. The exploitation of the results of the algorithms and the preprocessing of their input parameters is responsibility of the BGP2XQuery algorithm, too. The algorithms that are called by the BGP2XQuery algorithm are listed below:

- *Subject Translation*: Translates the subject part of all the triple patterns of a given BGP.
- *Predicate Translation*: Translates the predicate part of all the triple patterns of a given BGP.
- *Object Translation*: Translates the object part of all the triple patterns of a given BGP.
- *Filter Translation*: Translates the SPARQL FILTERs that may be contained in a given BGP.
- *Build Return Clause*: Builds the XQuery Return Clause.

The BGP2XQuery algorithm is presented in Figure 4.20.

BGP2XQUERY Algorithm

Input: Basic Graph Pattern *BGP*, SPARQL query form *QF*, Return Variables *RV*, Ontology and XML schema *mappings*, Variable Bindings *bindings*, Variable Types *varTypes*

Output: XQuery Expressions *XE*

6. *XE* ← **Subject Translation** (*BGP* , *QF* , *RV* , *bindings*)
7. *XE*.**append** (**Predicate Translation** (*BGP* , *QF* , *RV* , *bindings*))
8. *XE*.**append** (**Object Translation** (*BGP* , *QF* , *RV* , *bindings* , *mappings*))
9. *XE*.**append** (**Filter Translation** (*BGP* , *mappings*))
10. *XE*.**append** (**Build Return Clause** (*BGP* , *QF* , *RV* , *varTypes*))
11. **return** *XE*

Figure 4.20 The BGP2XQuery Algorithm

Subject Translation

The Subject Translation algorithm translates the subject part of all the triple patterns of a given BGP to XQuery expressions. The algorithm takes as input the Basic Graph Pattern (BGP), the SPARQL query form (QF), the SPARQL Return Variables (RV) and the variable bindings (bindings).

The Subject Translation algorithm remains unchanged from the SPARQL2XQuery 1.0 framework and it runs in the way that is described below and in Figure 4.21.

For each triple in which the subject is a variable (line 1,2), the algorithm creates a For or Let XQuery clause *XC* using the For or Let XQuery Clause Selection Algorithm (line 3). According to Definition 4.14 the *for/let XC* is comprised of a *Var* and an *expr*. *XC.Var* has the same value with the name of the subject (line 4) in order to achieve an one to one correspondence between the variable names in the SPARQL query and the XQuery query. *XC.expr* is defined using the variable bindings of the subject and the *$doc* variable (line 5) with the append operator in ordet to create a sequence of operands, on which the union operator is applied. Finally, the algorithm returns the generated For or Let XQuery clause (line 10).

Subject Translation Algorithm

Input: Basic Graph Pattern *BGP,* SPARQL Query Form *QF*, Return Variables *RV*, Variable Bindings *bindings*

Output: For or Let XQuery Clause *XC*

1. **for each** *triple* in *BGP*
2. **if** $S \in V$ *// If subject is a variable*
3. $XC.type \leftarrow$ For or Let XQuery Clause Selection (QF,RV,S) *//Create a For or Let XQuery Clause*
4. $XC.Var \leftarrow N_S$ *// Define an XQuery Variable with the name of SPARQL Variable S*
5. $XC.expr \leftarrow \$doc/x_1$ union $\$doc/x_2$ union ... union $\$doc/x_n$, $\forall x_i \in X_S$
6. *// Set the expr equal to XPath set of Subject prefixed with $doc variable*
7. *//XS is the bindings XPath set for the variable S*
8. **end if**
9. **end for**
10. **return** *XC*

Figure 4.21 The Subject Translation Algorithm

Predicate Translation

The Predicate Translation algorithm translates the predicate part of all the triple patterns of a given BGP to XQuery expressions. The algorithm takes as input the Basic Graph Pattern (BGP), the SPARQL query form (QF), the SPARQL Return Variables (RV) and the variable bindings (bindings).

The Predicate Translation algorithm remains unchanged from the SPARQL2XQuery 1.0 framework and it runs in the way that is described below and in Figure 4.22.

For each triple in which the predicate is a variable (line 1,2) the algorithm creates a For or Let XQuery clause *XC* using the For or Let XQuery Clause Selection Algorithm (line 3). *XC.Var* has the same value with the name of the predicate (line 4) in order to achieve an one to one correspondence between the variable names in the SPARQL query and the XQuery query. *XC.expr* is defined using the variable bindings of the predicate, the XQuery variable (N_S) that corresponds to the subject of the triple and the extension relation (Definition 4.11) for triple-patterns, in order to associate the subject and predicate bindings (line 5). Finally, the algorithm returns the generated For or Let XQuery clause (line 10).

Predicate Translation Algorithm

Input: Basic Graph Pattern *BGP*, SPARQL Query Form *QF*, Return Variables *RV*, Variable Bindings *bindings*

Output: For or Let XQuery Clause *XC*

1. **for each** *triple* in *BGP*
2. **if** $P \in V$ *// If predicate is a variable*
3. $XC.type \leftarrow$ For or Let XQuery Clause Selection (*QF*,*RV*,*P*) *//Create a For or Let XQuery Clause*
4. $XC.Var \leftarrow N_P$ *// Define an XQuery Variable with the name of SPARQL Variable S*
5. $XC.expr \leftarrow \$N_S/x_1$ union $\$N_S/x_2$ union ... union $\$N_S/x_n$, $\forall x_i \in (X_P)^{LN}$
6. *// Set the expr equal to the variable corresponding to triple's subject variable suffixed with*
7. *// Leaf nodes of Predicate XPath set. X_P is the bindings XPath set for the variable P*
8. **end if**
9. **end for**
10. **return** *XC*

Figure 4.22 The Predicate Translation Algorithm

Object Translation

The Object Translation algorithm translates the object part of all the triple patterns of a given BGP to XQuery expressions. The algorithm takes as input the Basic Graph Pattern (BGP), the SPARQL query form (QF), the SPARQL Return Variables (RV), the variable bindings (bindings) and the mappings between the ontology and the XML schema (mappings).

For the objects that are literals (lines 2-13) the algorithm uses XPredicates in order to translate them. The XPredicate restriction is applied to the (For or Let) XQuery clause created during the translation of the predicate variable of the triple (lines 4-5 & 10-11). If the predicate is not a variable, the appropriate restrictions are applied to the (For or Let) XQuery clause, created during the translation of the subject of the triple (lines 6-8).

For the objects that are variables (lines 14-33), if the predicate is also a variable (lines 15-21) the algorithm creates a Let XQuery clause (lines 16-18), in order to assign the predicate XQuery variable to the XQuery variable of the object. If the predicate is an IRI (lines 22-31) the algorithm creates a For or Let XQuery clause XC using the For or Let XQuery Clause Selection Algorithm (line 23). In this case the algorithm uses the variable bindings of the subject, the mappings of the predicate and the extension relation (Definition 16) for triple-patterns, in order to associate the subject, the predicate and the object bindings (line 25).

Object Translation Algorithm

Input: Basic Graph Pattern *BGP*, SPARQL Query Form *QF*, Return Variables *RV*, Variable Bindings *bindings,* XML Schema *mappings*

Output: Xquery Where clause *XC*

1. **for each** triple in BGP
2. **if** $O \in I$ *// If object is a literal*
3. **if** $P \in V$ *// If predicate is a variable*
4. Create XPredicate in expr's Xpaths of For/Let clause of P
5. *XPredicate* ← [.= "O"]
6. **if** Let XQuery Clause created for P
7. Create "Bindings Assurance Condition" for P [see "Biding Assurance Condition" Section]
8. **end if**
9. **else** *// predicate is not a variable-is a IRI*
10. Create XPredicate ? xi ? XS in expr's of For/Let clause of S
11. XPredicate ← [./y1 = "O" or ./y2 = "O" or … or ./yn = "O"] $\forall$ y_i ϵ ($x_i \oplus$ XP) LN
12. *// XS is the bindings XPath set for the subject S and XP is the mappings XPath set for the property P*
13. **end if**
14. **else if** $O \in V$ *// If object is a variable*

15. **if** $P \epsilon V$ *// If predicate is a variable*
16. $XC.type \leftarrow$ Create a Let XQuery Clause
17. $XC.Var \leftarrow N_O$ *// Define an XQuery Variable with the name of SPARQL Variable O*
18. $XC.expr \leftarrow \$ N_P$ *// Set the expr equal to the Variable of predicate*
19. **if** Let XQuery Clause created for P
20. Create "Bindings Assurance Condition" for O [see "Biding Assurance Condition" Section]
21. **end if**
22. **else** *// predicate is not a variable-is a IRI*
23. **if** P ϵ KR
24. Create for/let clause
25. $XC.VarTemp \leftarrow$ concat("temp_" , N_O)
26. $XC.expr \leftarrow \$N_S$
27. **for each** keyref.field
28. $i \leftarrow$ *number of iteration*
29. create for/let
30. $VarProperty_i \leftarrow$ concat(N_O, "_" , N_S , "_" , i)
31. $ExprProperty_i \leftarrow \$N_S/p_n \ \forall \ p_n \epsilon (X_S \ominus X_{prKeyrefRi})$
32. **end for**
33. Create for/let clause
34. $XC.Var \leftarrow \$N_O$
35. $XC.expr \leftarrow \$XC.VarTemp$ [./y_1=$\$VarProperty_1$ and ... and ./y_n=$\$VarProperty_n$] $\forall \ y_i \epsilon (X_{VarTemp} \ominus x_i)$
36. **end if**
37. $XC.type \leftarrow$ For or Let XQuery Clause Selection (QF , RV , O) *//Create a For or Let XQuery Clause*
38. $XC.Var \leftarrow N_O$ *// Define an XQuery Variable with the name of SPARQL Variable O*
39. $XC.expr \leftarrow \$N_S/x_1$ union $\$N_S/x_2$ union ... union $\$N_S/x_n \ \forall \ x_i \epsilon (Xs \circledR X_P)$ LN
40. *// Set the expr equal to the variable corresponding to triple subject suffixed with some of*
41. *// Leaf Nodes of Predicate's XPath set. XS is the bindings XPath set for the subject S and*
42. *// XP is the mappings XPath set for the property P.*
43. **if** Let XQuery Clause created for O
44. Create "Bindings Assurance Condition" for O [*see "Biding Assurance Condition" Section*]
45. **end if**

46. **end if**
47. **end if**
48. **end for**
49. **return** XC

Figure 4.23 The Object Translation Algorithm

Filter Translation

The Filter Translation algorithm translates the SPARQL FILTERs that may be contained in a given BGP to XQuery expressions. The algorithm takes as input the Basic Graph Pattern (*BGP*) and the XML Schema *mappings*.

The Filter Translation algorithm has been modified since the previous version of SPARQL2XQuery in order to support queries that check the equality between complex types that include in their declaration information about identity constraints, i.e. key and unique. For every equality expression between two complex types that appears in a FILTER expression is exploited the information about their identity constraints that exists in the *mappings*. Thus, for every property being the primary key of a class an equality expression is created and joined together in order to create the appropriate expression that checks the equality between two complex types (line 5).

Due to the complexity that a FILTER expression may have, the algorithm translates all the filters into XQuery Where clauses, although some "simple" ones (e.g., conditions on literals) could be translated using Xpath predicates. Moreover, SPARQL operators (built-in functions) included in FILTER expressions are translated using the built-in XQuery functions and operators (line 8). However, for some "special" SPARQL operators (like sameTerm, lang, etc.) native XQuery functions have been developed to simulate them. These functions can be executed by any XQuery compliant engine. Afterwards, the algorithm generates a where clause (line 9) taking into account the translation of the filter expressions and the complex type equality expressions *XC.CTexpr*. Finally, the algorithm returns the generated where XQuery clause (line 11).

Filter Translation Algorithm

Input: Basic Graph Pattern *BGP,* XML schema *mappings*

Output: Xquery Where clause *XC*

1. **for each** *FILTER* in *BGP*
2. Let *CT_eqExpr* contain filter expressions of equality between complex types
3. **if** *CT_eqExpr* != ∅

4. **for each** *expr* in *CT_eqExpr*
5. *XC.CTexpr* ← $x/$k_1$=$y/k_1 and ... and $x/$k_n$=$y/k_n $\forall$ $k_n \epsilon$ ($X_x \ominus X_{keyR}$)
6. **end for**
7. **end if**
8. Translate SPARQL operators of filter's expressions *expr*, where *expr* $\notin$ *CT_eqExpr*
9. *XC* ← Create a Where clause condition including the *XC.CTexpr* conditions
10. **end for**
11. **return** *XC*

Figure 4.24 The Filter Translation Algorithm

Build Return Clause

Finally, the Build Return Clause algorithm builds the XQuery Return Clause. The algorithm takes as input the Basic Graph Pattern (BGP), the SPARQL query form (QF), the Return Variables (RV) and the determined variable types (varTypes).

For ASK queries (lines 1-2) the algorithm creates an XQuery Return clause which, for efficiency reasons includes only the string "yes" (line 2). For other query forms (lines 3-7), the algorithm creates an XQuery Return clause XC that includes all the Return Variables (RV) used in the BGP (line 5). The syntax of the return clause allows (using markup tags) the distinction of each solution in the solution sequence as well as the distinction of the corresponding values for each variable. This allows the further treatment of the returned results (e.g., applying SPARQL operators like AND or OPT on them). Finally, the algorithm, for each variable included in the return clause, and based on the variable types (varTypes), uses the appropriate function to format the result form of the variable (Section 7.2.1). Finally, the algorithm returns the generated return XQuery clause (line 9).

Build Return Clause Algorithm

Input: Basic Graph Pattern *BGP*, SPARQL query form *QF*, SPARQL Return Variables *RV*, Variable Types *varTypes*

Output: XQuery Return Clause *XC*

1. **if** *QF* is ASK
2. *XC* ← return("yes") *//Create an XQuery Return clause*

3. **else** *//Query form is not ASK*
4. *//Create an XQuery Return clause*
5. *XC* ← return(<Result> <var_1>..</var_1> ,<var_2>..<var_2>,...,<var_i>..</var_i></Result>) $\forall$ var_i ϵ (*RV* $\cap$ *BGP*)
6. *// Each Return Variable included in the given BGP, insert it to the XQuery return clause*
7. Use the *varTypes* to determine the form of result for each vari
8. **end if**
9. **return** *XC*

Figure 4.25 The Build Return Clause Algorithm

For or Let XQuery Clause Selection

A main issue in constructing the XQuery expression is the enforcement of the correct solution sequence based on the SPARQL semantics. To achieve this, a For or a Let clause is used according to the algorithm presented below. The algorithm takes as input the SPARQL query form (QF), the SPARQL Return Variables (RV) and a SPARQL variable (V) included in the given basic graph pattern.

For the query forms Select, Construct and Describe (lines 1-6) the algorithm generates for every variable V a For XQuery clause if V is included in the Return Variables (RV) or if any return variable is an extension (Definition 16) of variable V (line 3). Otherwise it generates a Let XQuery clause (line 5).

For Ask queries (lines 7-9) that do not return a solution sequence, and for efficiency reasons, the algorithm generates only Let XQuery clauses (line 8), in order to check if a BGP can be matched over XML data.

For or Let XQuery Clause Selection Algorithm

Input: SPARQL query form *QF*, Return Variables *RV*, SPARQL variable *V*

Output: For or Let XQuery Clause flag

1. **if** *QF* is not ASK
2. **if** (V ϵ RV) **OR** ($\exists$ K ϵ RV | K is extension of V) then

3. **return** Create a *For* XQuery Clause
4. **else**
5. **return** Create a *Let* XQuery Clause
6. **end if**
7. **else**
8. **return** Create a *Let* XQuery Clause
9. **end if**

Figure 4.26 For or Let XQuery Clause Selection Algorithm

4.4.4.3. Basic Graph Pattern to XQuery with Shared Variables

The Basic Graph Pattern to XQuery translation process that is supported by the BGP2XQuery algorithm is modified if there are triple patterns that have shared variables [Definition 2.20]. Moreover, the algorithm is modified according to the position of the shared variables within the triple pattern (subject/predicate/object). The possible combinations of the shared variables positions are listed below.

Consider the triple patterns: $(SPO)_1$, $(SPO)_2$, ... , $(SPO)_n$. Let *V* be the set of variables that appear in the basic graph pattern.

- **Case Subject-Subject:** $S_1 \equiv S_2 \equiv ... \equiv S_m$, where $m \leq n$ and $S_1, S_2, ..., S_m \in V$. In this case the shared variable appears in *m* triples at the subject position.

- **Case Predicate-Predicate:** $P_1 \equiv P_2 \equiv ... \equiv P_m$, where $m \leq n$ and $P_1, P_2, ..., P_m \in V$. In this case the shared variable appears in *m* triples at the predicate position.

- **Case Object-Object:** $O_1 \equiv O_2 \equiv ... \equiv O_m$, where $m \leq n$ and $O_1, O_2, ..., O_m \in V$. In this case the shared variable appears in *m* triples at the object position.

- **Case Subject-Object:** $S_1 \equiv S_2 \equiv ... \equiv S_m \equiv O_1 \equiv O_2 \equiv ... \equiv O_k$, where $m,k \leq n$ and $S_1, S_2, ..., S_m, O_1, O_2, ..., O_m \in V$. In this case the shared variable appears in *m* triples at the subject position and in *k* triples at the object position.

The rest of this subsection presents the analysis of the cases listed above and specifies the needed modifications to the BGP2XQuery in order to handle the cases of the shared variables. In the context of this thesis, it was re-examined each one of these cases in order to encapsulate and

exploit the semantics of the identity constraints that are supported by the SPARQL2XQuery 2.0 framework.

Case Subject-Subject

If the shared variable appears at the subject position in *m* triple patterns there is no need for any change in the translation process and the BGP2XQuery algorithm, since the variable binding with XPaths is soundly and completely covered by the Subject Translation algorithm (Figure 4.21).

Case Predicate-Predicate

If the shared variable appears at the predicate position in *m* triple patterns, the translation must be re-examined as the variables of the predicates and objects of a triple pattern are generated with extensions to the subject variable. However, the algorithms that handle the shared variables in this case are not affected by key-keyref definitions, as the identity constraints are parts of classes that appear at the positions of subjects, objects and not of predicates. As a consequence, no changes in the existing algorithms are required.

Case Object-Object

If the shared variable appears at the object position in *m* triple patterns, the existing algorithm was examined and changed as is shown in Figure 4.27.

Shared Object Object Translation Algorithm

Input: Basic Graph Pattern *BGP*, SPARQL query form *QF*, Return Variables *RV*, Ontology and XML schema *mappings*, Variable Bindings *bindings*, Variable Types *varTypes*

Output: Xquery Return Clause *XC*

1. **for each** *conjunctive triple* of case Object-Object in *BGP*
2. **if** *O* ∈ V and *O* is VLTP *// in case O is a variable of literal type*
3. *XC.type* ← For or Let XQuery Clause Selection (QF,RV,O)
4. *XC.var* ← N_O
5. *XC.expr* ← X_1[.= X_2[...[.= X_m]]...]
6. **if** P_i ∈ V *// in case the predicate of the i triple pattern is a variable*
7. X_i ← P_i
8. **else**

9. **for each** X_{k_i} in $(X_{Si} \oplus X_{Pi})^{LN}$
10. $X_i \leftarrow (\$S_i/X_{k\_1}$ union $\$S_i/X_{k\_2}$ union...union $\$S_i/X_{k\_n})$
11. **end for**
12. **end if**
13. **If** *XC.type* = Let clause
14. Create an Exist condition
15. **end if**
16. **else**
17. **if** *P* ϵ I and *P* represents a keyref
18. *XC.type* ← For or Let XQuery Clause Selection (QF,RV,O)
19. *XC.var* ← N_O
20. *XC.expr* ← $\$XC.VarTemp$ [./y_1=$\$VarProperty_{S1\_1}$ and ./y_1=$\$VarProperty_{S2\_1}$ and...and ./y_i=$\$VarProperty_{S1\_m}$ and ./y_1=$\$VarProperty_{S2\_m}$] where *m* is the number of conjunctive triples and y_i ϵ $(X_{VarTemp} \ominus x_i)$
21. **else if** *P* ϵ I
22. *XC.type* ← For or Let XQuery Clause Selection (QF,RV,O)
23. *XC.var* ← N_O
24. *XC.expr* ← BGP2XQuery
25. **else**
26. *XC.type* ← Let XQuery Clause
27. *XC.var* ← N_O
28. *XC.expr* ← $\$N_P$
29. *Enrich the Where Clause with equality check between the full path of object variable and the predicate variables.*
30. **return** *XC*
31. **end for**

Figure 4.27 The Shared Object-Object Translation Algorithm

Case Subject-Object

If the shared variable appears at the subject position in *m* triple patterns and at the object position in *k* triple patterns, the existing algorithm was examined and changed as is shown in Figure 4.28.

In this case the for/let declarations must follow another order than the one described in the BGP2XQuery algorithm. The triple patterns that include the shared variable at the object position must be translated before the translation of the triple patterns which include the shared variable at the subject position. Moreover, these subjects are ignored as the object declarations cover the requirements soundly and completely.

Furthermore, if the shared variable refers to a class (complex type) that has the identity constraint key, the algorithm (Figure 4.28) remains unchanged and the handling of the key - keyref pair takes place within the Object Translation algorithm presented in Figure 4.23.

Shared Subject Object Translation Algorithm

Input: Basic Graph Pattern *BGP*, SPARQL query form *QF*, Return Variables *RV*, Ontology and XML schema *mappings*, Variable Bindings *bindings*, Variable Types *varTypes*

Output: Xquery Expression *XE*

1. **for each** *conjunctive triple* of case Subject-Object in *BGP*
2. $XE \leftarrow$ BGP2XQuery() on the triple that contains the O_i , where $1 \leq i \leq k$
3. **return** *XE*
4. **end for**

Figure 4.28 The Shared Subject-Object Translation Algorithm

4.4.5. Query Form Translation

The SPARQL syntax recommendation describes four query forms (i.e. ASK, SELECT, DESCRIBE, CONSTRUCT) that can be followed in order to express queries. Each one query form results in a different form of the retrieved data. Moreover, the translation process which was presented in the previous sections remains the same among the different query forms and is completely independent of them. Consequently, the form of the results that comprise the answer to a specific query must be modified by another algorithm in order to satisfy the SPARQL requirements.

The *Query Form Translation* is the final step of the translation. In particular, after the translation of any solution sequence modifier, the generated XQuery is enhanced with appropriate XQuery expressions in order to achieve the desired structure of the results according to the query form.

The rest of this section presents the SELECT query translation (subsection 4.4.5.1), the ASK query translation (subsection 4.4.5.2), the CONSTRUCT query translation (subsection 4.4.5.3) and finally the DESCRIBE query translation (subsection 4.4.5.4).

4.4.5.1. SELECT Query Translation

The SELECT form queries return a sequence of results that correspond to the variables that are mentioned in the SELECT expression. On the other hand in the XQuery queries the return variables are specified within the return expression of the query. Thus, the variables of the XQuery return clause are those that are mentioned in the SELECT expression of the SPARQL query.

As it has been already mentioned, the XQuery evaluation returns a sequence of result elements, which comprise an XML file. However, these result elements will form an XML file with multiple top level elements as there is not any root element. Consequently, the XML file of the results will be invalid under these circumstances. This issue is treated by the *SELECT queries Translation* algorithm.

The algorithm aims to create a root element over the sequence of the result elements and host these elements. As it is shown in Figure 4.29, the XQuery that created by the translation process is assigned to a variable via a let expression (line 1-3) and the evaluation of the variable is taking place through the return expression (line 5) within the <Results> element tags. Thus the <Results> element encloses the results sequence and is the root element of the XML file.

SELECT query Translation Algorithm

Input: XQuery Expression *xquery*

Output: XQuery Expression *newXquery*

1. *XC.type* ← Let XQuery Clause
2. *XC.var* ← "Results"
3. *XC.expr* ← *xquery*
4. *newXquery.body* ← *XC*
5. *newXquery.return* ← concat("<Results>" , "$", *XC.var* , "</Results>")
6. **return** *newXquery*

Figure 4.29 The SELECT query Translation Algorithm

4.4.5.2. ASK Query Translation

The queries of ASK type do not return any sequence of solutions, but only a literal of value "yes" or "no", respectively, in case the query has some solution or not. Thus, the return clause of an ASK equivalent XQuery does not contain any variables.

The simplest solution to this issue would be the assignment of the “yes” literal to the return statement in order to return a “yes” whenever the XQuery finds a solution to the query into the XML repository. However, this approach would result into a sequence of “yes” strings if solution sequence was comprised of more than one results. This fact is not compatible with the semantics of SPARQL ASK query form, which returns only one “yes” or “no” literal. Consequently, the *ASK query Translation* algorithm has to handle this situation and adapt the XQuery in order to follow the semantics of the SPARQL ASK query.

As it is shown in Figure 4.30, the XQuery created by the translation process is assigned to a variable via a let expression (line 1-3). Afterwards, in the return expression (line 5), the algorithm assigns a conditional expression which will return “yes” if the Result set is not empty and in any other case it will return “no”.

ASK query Translation Algorithm

Input: XQuery Expression *xquery*

Output: XQuery Expression *newXquery*

1. *XC.type* ← Let XQuery Clause
2. *XC.var* ← “Results”
3. *XC.expr* ← *xquery*
4. *newXquery.body* ← *XC*
5. *newXquery.return* ← concat(“if(empty($” , *XC.var* , “)) then ‘no’ else ‘yes’ ”)
6. **return** *newXquery*

Figure 4.30 The ASK query Translation Algorithm

4.4.5.3. CONSTRUCT Query Translation

The queries of CONSTRUCT type return an RDF graph, which is specified by a graph template the structure of which is specified within the CONSTRUCT query. Furthermore, the new graph is formed by applying the values of the variables in the result sequence to the correspondent variables of the new graph template. In the special case that there is an unbound variable in a triple pattern, this triple pattern is excluded from the final graph pattern. Also, the template of the new graph pattern may contain blank nodes (Definition 2.1). In this case, for each solution of the result sequence, a different blank node is created as it describes below the representation of the *CONSTRUCT Query Translation* algorithm.

For enforcing the semantics of the Blank node naming conventions in the RDF graph it was defined an XQuery positional variable (line 7).

CONSTRUCT query Translation Algorithm

Input: XQuery Expression *xquery*

Output: XQuery Expression *newXquery*

1. Let *template* be the template graph pattern
2. *XC_let.type* ← Let XQuery Clause
3. *XC_let.var* ← "Results"
4. *XC_let.expr* ← *xquery*
5. *XC_for.type* ← For XQuery Clause
6. *XC_for.var* ← *"res"*
7. *XC_for.positionVar* ← "iter"
8. *XC_for.expr* ← XC_let.var
9. *newXquery.body*.**append**(*XC_let*)
10. *newXquery.body*.**append**(*XC_for*)
11. **for each** $triple_i$ in *template*
12. **if** $triple_i.S$ is blank node
13. *ConcatExpr* ← **concat** ($triple_i.S$, *XC_for.positionVar*)
14. **else**
15. *ConcatExpr* ← **string** (*XC_for.var* / $triple_i.S$)
16. *ConditionExpr* ← **exists** (*XC_for.var* / $triple_i.S$)
17. **end if**
18. **if** $triple_i.P$ is blank node
19. *ConcatExpr*.**append** (**concat** ($triple_i.P$, *XC_for.positionVar*))
20. **else**
21. *ConcatExpr*.**append** (**string** (*XC_for.var* / $triple_i.P$))
22. *ConditionExpr*.**and** (**exists** (*XC_for.var* / $triple_i.P$))
23. **end if**
24. **if** $triple_i.O$ is blank node
25. *ConcatExpr*.**append** (**concat** ($triple_i.O$, *XC_for.positionVar*))
26. **else**
27. *ConcatExpr*.**append** (**string** (*XC_for.var* / $triple_i.O$))
28. *ConditionExpr*.**and** (**exists** (*XC_for.var* / $triple_i.O$))
29. **end if**
30. *ConcatExpr*.**append** (".")
31. *XE.if.condition* ← *ConditionExpr*

32. *XE.if.true* ← *ConcatExpr*
33. *XE.if.false* ← () *// in case we have unbound variables*
34. *newXquery.return.***append** (*XE.if*)
35. **end for**
36. **return** *newXquery*

Figure 4.31 The CONSTRUCT query Translation Algorithm

4.4.5.4. DESCRIBE Query Translation

The Describe query form of SPARQL returns an RDF graph which provides a "description" of the matching resources. The "description" is not determined by the current SPARQL specification, but is determined by the SPARQL query processor (several SPARQL engines do not support Describe queries). For this reason, is provided ad hoc approximate support for this query form, by executing the Describe SPARQL query against the source ontology and then translating it as a Select form. The overall result is a combination of the RDF graph produced by the SPARQL Describe query and the result returned by the translated XQuery. This activity has not changed since the SPARQL2XQuery 1.0 framework.

4.5. Summary

This chapter described the translation of SPARQL queries to semantically correspondent XQuery queries via the SPARQL2XQuery 2.0 framework. The analysis of the framework is focused on the new features that the 2nd version of the SPARQL2XQuery framework introduces and the way the existing algorithms were modified in order to support the new functionality.

The architecture of the SPARQL2XQuery 2.0 framework as it was presented in this chapter, describes the exploitation of the mapping ontology in case of automatically generated mappings via the XS2OWL 2.0 framework. This is a crucial difference from its ancestor, which did not exploit the information of the mapping ontology. In the new version of XS2OWL, the mapping ontology holds important information about the identity constraints of the XML Schema. Thus, it is a good practice to include the mapping ontology in the Mappings generation process, which was presented in detail in section 4.3.

Moreover, the modifications of the algorithms, that were implemented in order to support the new features, were presented through the translation process via a comprehensive analysis and a broad number of examples.

The next chapter (Chapter 5) analyzes the implementation of the XS2OWL 2.0 and the SPARQL2XQuery 2.0 framework from a technical point of view.

5. Implementation

The XS2OWL 2.0 framework, responsible for the XML Schema transformation and the SPARQL2XQuery 2.0 framework, responsible for the translation of SPARQL queries in XQuery syntax, presented in this thesis, have been implemented as an integrated framework and at the same time as two self standing and independent software packages.

The XS2OWL 2.0 framework has been implemented using the XSLT 2.0 recommendation [W3C/XSLT] and has followed the same conventions and guidelines with XS2OWL 1.0. During the implementation of the framework a considerable number of difficulties that were due to the limitations of the XSLT language had to be overcome.

The SPARQL2XQuery 2.0 framework has been implemented using the Java 2SE software platform, XMLBeans for creating Java beans from the xml schema, the Jena Software framework for SPARQL query and Ontology parsing and the Oracle Berkeley DB XML as the Native XML Database which handles the underlying XML repository.

The rest of this section is structured as it follows: in subsection 5.1, is described the implementation of XS2OWL 2.0 framework and the by-products of the implementation process (subsection 5.1.1), in subsection 5.2, is described the implementation of SPARQL2XQuery 2.0 and the Java APIs that was used (XML Beans, Berkeley DB XML, Jena).

5.1. XS2OWL 2.0 Implementation

The XS2OWL 2.0 transformation model has been implemented as an extension of the XS2OWL framework, using the XSLT language presented below. In particular, the XS2OWL 2.0 framework has been implemented to uplift an XML Schema 1.1 to OWL 2.0 syntax. *Uplifting* is the process which captures the semantics of an xml schema that can be transformed and represented in OWL syntax while ignoring any structures that are not supported by the target technology. The main drawback of uplifting is the fact that there is no way to generate the source xml schema from the OWL ontology that was created through the uplifting process. The XS2OWL 2.0 framework, like the former XS2OWL 1.0 version, is implemented in order to make feasible to downlift from the OWL syntax to the correspondent xml schema by exploiting the products of the uplifiting process. *Downlifting* is the opposite process of uplifting. The XS2OWL framework supports downlifting through the mapping ontology, which holds any xml schema information that cannot be represented in OWL 2.0 syntax and would be required by another framework, e.g. OWL2XS (not implemented in this thesis), which would be responsible to generate a correspondent xml schema for a main ontology created by XS2OWL 2.0.

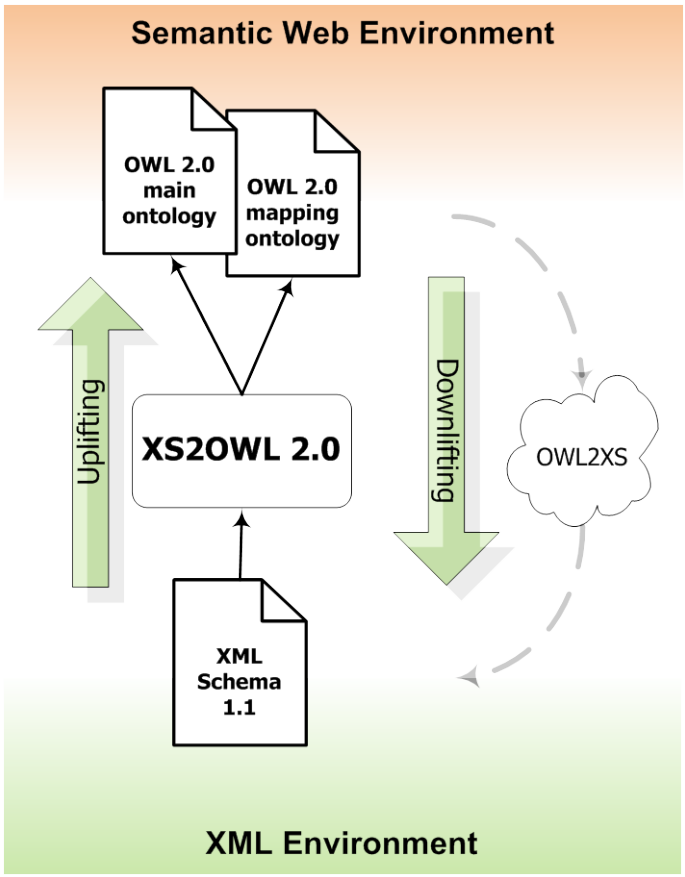

Figure 5.1 The XS2OWL 2.0 Implementation

XSLT

XSLT is a language for transforming XML documents into other XML or XML-like documents. In the core of XSLT transformation stands the XSLT processor, which is the software responsible for transforming source trees into result trees using an XSLT stylesheet. The term **result tree** is used to refer to any tree constructed from the instructions in the stylesheet. The term **source tree** means any tree provided as input to the transformation.

A transformation expressed in XSLT describes rules for transforming zero or more source trees into one or more result trees. The transformation is achieved by a set of **template** rules. A template rule associates a pattern, which matches the nodes in the source document, with a sequence constructor. In many cases, evaluating the **sequence constructor** will cause new nodes to be constructed, which can be used to produce a part of a result tree. The structure of the result trees can be completely different from the structure of the source trees. In constructing a result tree, nodes from the source trees can be filtered and reordered, and arbitrary structure can be added. This mechanism allows a stylesheet to be applicable to a wide class of documents that have similar source tree structures.

A **template** can serve either as a template rule, invoked by matching nodes against a pattern, or as a named template, invoked explicitly by name. It is also possible for the same template to serve in both cases. The **named templates** are very important in the XSLT development for creating recursions by calling a template rather than counting on pattern matching.

The **variables** allow the programmer to use a particular name to stand for a value within an expression. One can refer to a variable within an expression by prefixing its name with a dollar sign. When the expression is evaluated, the processor uses the value of the variable in place of the reference.

One of the most important trickier issues in XSLT is the way that the variables behave in a stylesheet. The **XSLT variables** have one quite peculiar feature: they **cannot vary**. Once a variable has been given a value, the value cannot be changed; it remains the same throughout its lifetime. XSLT does not provide an equivalent to the assignment operator available in many procedural programming languages. This is because an assignment operator would make it harder to create an implementation that processes a document in a non batch-like way, starting at the beginning and continuing to the end.

In the procedural programming languages, we can use iteration to perform calculations: we can initialize a variable and update it each time we loop. XSLT, on the other hand, is a **functional programming language**, which means that the same instruction in the same context will always produce the same thing, no matter how many times it is run. The **XSLT does not support while loops** and the for loops have different semantics from the procedural languages. This drawback leads to recursive solutions. When we are programming with XSLT, we need to use recursion to give the same effect as we would get from while loops in procedural languages. Recursion in XSLT can be implemented using named templates, as presented previously in this subsection, or using functions.

The stylesheet functions are functions that are defined within a stylesheet rather than being built into the XSLT processor. Both named templates and functions are blocks of code and instructions that can be called by their name, but whereas the named templates are called using an instruction (<xsl:call-template>) within a sequence constructor, the stylesheet functions are called using a standard function call from within an XPath expression or a pattern.

The XS2OWL 2.0 framework is implemented using XSLT 2.0, which is designed to be used in conjunction with XPath 2.0 [W3C/XPATH2].

Thesis by-products

The limitations of XSLT made it necessary to develop a set of functions and procedures in order to overcome them. Moreover, we created some functions that can be re-used in future XSLT projects independent of the XS2OWL 2.0 framework. These are the by-products of this thesis and are presented in the rest of this subsection.

XSLT Hashtable

We have already discussed, that the XSLT variables cannot vary. Thus, it is not feasible to implement a true hashtable; like the one that it is available in Java. The hashtable is implemented using a variable. The information about the hashtable is represented by the string which is assigned to that variable. This string comprises of *key-value* pairs separated with a neutral character according to the values that are stored within the hashtable-string, e.g. *Hashtable= "key:A,valueA,key:B,valueB,key:C,valueC"*.

The retrieval of the *value* that corresponds to a specific *key* is supported by a get function, which searches among the keys of the string and exploits the built-in functions "substring-after" and "substring-before" in order to extract the *value* string.

XPath Evaluator

Since the XPath expressions do not refer to the node hierarchy of the XML Schema but in the node structure of the XML data following it, the "XPath Evaluator" algorithm (see the activity diagram of Figure 3.29) has been developed for XPath expression evaluation. The XPath Evaluator algorithm supports the subset of XPaths that is used by the XML Schema identity constraints (unique, key, keyref).

In particular, the supported types of nodes, i.e. the types of nodes that the XPath Evaluator algorithm can traverse, are element, attribute, group, attributeGroup, references and complex types.

Furthermore, the XPath evaluator can be developed into a separate XSLT library useful for discovering the definition, in an XML Schema, of the XML element or attribute indicated by XPath expressions.

5.2. SPARQL2XQuery 2.0 Implementation

The SPARQL2XQuery 2.0 has been implemented as an extension of the SPARQL2XQuery 1.0 framework, using Java related technologies and the Oracle Berkeley DB XML database, as is described in the rest of this subsection.

XML Beans

The XML Beans framework provides the serialization of the XML schema, followed by underlying the XML DB repository, into Java beans in order to encapsulate the XML schema into the object-oriented philosophy of the SPARQL2XQuery 2.0 implementation. This process is called *XML data binding*.

XML data binding is the binding of XML documents to objects designed especially for the data in those documents. This allows applications (usually data-centric) to manipulate data that has been serialized as XML in a way that is more natural than using the DOM.

There is a respectable number of techniques that support XML data binding. However, it was decided to use the XML Beans due to a sequence of characteristics that fit to the SPARQL2XQuery 2.0 implementation:

- It provides a familiar Java object-based view of the XML data without losing access to the original, native XML structure.
- The integrity of an XML document is not lost with XMLBeans. XML-oriented APIs usually take the XML apart in order to bind to its parts. With XMLBeans, the entire XML instance document is handled as a whole. The XML data is stored in the main memory as XML. This means that the document order is preserved as well as the original element content including the whitespace.
- With the XML Schema types, access to XML instances is through JavaBean-like accessors, with get and set methods.
- It is designed with XML schema in mind from the beginning — XMLBeans supports all the XML schema definitions.
- The access to XML data is fast.

Oracle Berkeley DB XML

The underlying XML repository in this thesis is handled by an Open Source Native XML Database, the Berkeley DB XML of Oracle [BerkeleyDB]. Moreover, Berkeley DB XML is an embeddable XML database with XQuery-based access to documents stored in containers and indexed based on their content. The Oracle Berkeley DB XML is built on top of the Oracle Berkeley DB and inherits its rich features and attributes. Like the Oracle Berkeley DB, it runs in the same process with the application with no need for human administration. Oracle Berkeley DB XML adds a document parser, XML indexer and XQuery engine on top of the Oracle Berkeley DB to enable faster and more efficient data retrieval.

In Berkeley DB XML, all XML data is stored within files called containers. Those files are not restricted to follow the same XML Schema, they may follow different schemas. The containers essentially are a collection of XML documents and information about those documents.

Berkeley DB XML Benefits

The Berkeley DB XML is implemented to conform with the W3C standards for XML, XML Namespaces, and the latest available XQuery standards. The most important benefits of the XML DB are listed below:

- Containers: a single file that contains one or more XML documents, and their metadata and indexes.
- Indexes: quickly identify subsets of documents that match specific queries, thus allowing for improved query performance against the corresponding XML data set.
- Integrity: documents are stored (and retrieved) in their native format with all whitespace preserved.
- Metadata: each document stored in BDB XML can have "data about the data" associated with the document

Berkeley DB XML Architecture

The Oracle Berkeley DB XML is built on top of the Oracle Berkeley DB and inherits its rich features and attributes. The Oracle Berkeley DB XML adds a document parser, XML indexer and XQuery engine on top of the Oracle Berkeley DB to enable the fastest, most efficient retrieval of data.

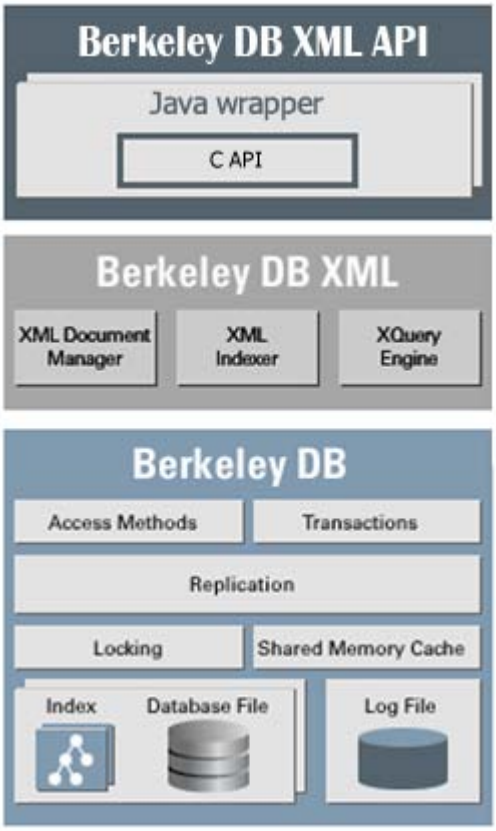

Figure 5.2 Oracle Berkeley DB XML Architecture

The Berkeley DB subsystems can be accessed through interfaces from multiple languages. Applications can use Berkeley DB via C, C++ or Java, as well as a variety of scripting languages such as Perl, Python, Ruby or Tcl. The Berkeley DB library is written entirely in ANSI C.

The Java classes provide a wrapper around the C API (see Figure 5.2). Berkeley DB constants and #defines are represented as "static final int" values. The error conditions are communicated as Java exceptions.

Moreover, each Java object has exactly one construct from the underlying C API associated with it. The Java structure is allocated with each constructor or open call, but is deallocated only by the Java garbage collector.

Jena

Jena is an open source Semantic Web framework developed in the HP labs. Jena is a Java API which can be used to create and manipulate RDF graphs. Jena has object classes to represent graphs, resources, properties and literals. The major advantage of Jena compared with other frameworks of similar functionality (e.g. Sesame) is the capability to support the OWL language.

Jena offers an enriched Model API for the manipulation of RDF graphs. The entire graph is represented by the "Model" class. Jena offers a parser for reading and writing RDF in RDF/XML, N3 and N-triples.

Except for the RDF API, Jena offers an OWL API, which uses a special model class, i.e. "OntModel" for the cooperation of Jena with OWL. Using this Jena can derive meaningful relationships that the Model class cannot express directly.

Finally, Jena offers a query language for RDF, the RDQL, which is very SQL like. Moreover, it offers in-memory and persistent storage.

5.3. Applications

In this section is demonstrated how the SPARQL2XQuery 2.0 framework can be used in real-world applications of the multimedia and the cultural heritage domains.

Multimedia Domain

As already mentioned, the dominant standards for content and service description (MPEG-7 [MPEG7] and MPEG-21 [MPEG21] respectively) in the multimedia domain have been expressed in XML Schema syntax. As a consequence, several groups have been working with these standards and a great number of MPEG-7 [MPEG7] and MPEG-21 [MPEG21] descriptions have been created. The development of the Semantic Web, though, has made many research groups to adopt the Semantic Web technologies, develop ontologies that capture (fully or partially) the semantics of the standards and work with OWL/RDF descriptions formed according to the ontologies. Since there exist standard-based XML descriptions as well as groups working with the XML Schema based syntax of the standards, the capability of transparently posing queries on both the RDF and the XML Schema repositories is necessary. This can be achieved using the SPARQL2XQuery 2.0 framework in two different usage scenarios:

(a) An existing ontology like [TsPoCh07], which captures the semantics of the standard(s), is used and mappings between the ontology and the XML Schemas are manually defined. Then, the end-users pose their SPARQL queries over the ontology and the SPARQL2XQuery 2.0 framework expresses the queries in XQuery syntax, evaluates them and returns the query results.

(b) The XS2OWL 2.0 framework is used to automatically express the semantics of the standards in OWL 2.0 syntax. Then, the SPARQL2XQuery 2.0 framework automatically specifies the mappings between the generated ontology and the XML Schema(s) and may support user queries expressed in SPARQL syntax over the generated ontology in the same way it supports the queries of scenario (a).

Cultural Heritage Domain

There are several standards (over 100) for content description in the cultural heritage domain expressed in XML Schema syntax, like the TEI [TEI], the EAD [EAD] and several others. On the other hand, the CIDOC/CRM [CIDOC/CRM] standard has been developed, which essentially is an ontology, expressed in OWL/RDF syntax, that subsumes the semantics of the above-referred standards. Since the cultural heritage institutions have invested a great amount of time in the specification of descriptions that are formed according to the XML-based standards and they may have even developed software that manages them, the SPARQL2XQuery 2.0 framework can be used in order to make their contents accessible to users aware of the CIDOC/CRM without having to change their working environment. In particular, mappings between the CIDOC/CRM ontology and the XML Schemas of the standards should be manually defined. Then, the end-users may pose their SPARQL queries over the CIDOC/CRM ontology and the SPARQL2XQuery 2.0 framework expresses the queries in XQuery syntax, evaluates them and returns the query results.

Chapter 5: Framework Implementation and Applications

6. Conclusion

This thesis aims to support interoperability between OWL 2.0 and XML environments. In this context have been developed the new versions of two pre-existing frameworks, the XS2OWL 2.0 and the SPARQL2XQuery 2.0. The former, XS2OWL 2.0, allows transforming the XML Schema 1.0/1.1 constructs in OWL 2.0, so that applications using the XML Schema based standards will be able to use the Semantic Web methodologies and tools. The SPARQL2XQuery 2.0 allows to apply SPARQL queries via XQuery interfaces on XML Databases. Both systems were implemented as extensions of the previous versions of the frameworks.

These two frameworks are treated as one integrated framework. This however does not prevent the reuse of each one individually.

6.1. Conclusion

The development of both of the XS2OWL 2.0 and the SPARQL2XQuery 2.0 frameworks in the context of this thesis assisted the exploitation of every provided feature of the XS2OWL 2.0 into the translation process of the SPARQL2XQuery 2.0.

In the new version of XS2OWL, it was redesigned the OWL representation of the XML constructs by exploiting the new features of OWL 2.0. Thus, the contribution of this thesis in the XS2OWL framework is:

- The representation of the XML Schema simple types directly in OWL 2.0 syntax and modification of the related algorithms in order to exploit the semantics of the XML Schema simple types within the generated ontology.

- The representation of the XML Schema identity constraints in OWL 2.0 syntax by capturing the semantics of them and supporting semantically correspondent functionality and reasoning. It is worthwhile to mention the new ability that is provided to the semantic web reasoners using the semantics of the identity constraints, which provide references and relations among individuals and constructs.

- The enrichment of the mapping ontology of the XS2OWL framework which stores any information of the XML Schema that cannot be represented directly in OWL 2.0 syntax. Moreover, the new mapping ontology was designed in order to play a crucial role in the activity of the mappings generation between an XML Schema and an ontology that takes place in the context of the SPARQL2XQuery framework.

- The representation and support of the new features of XML Schema 1.1 in order to allow the transformation of XML Schemas 1.1 to OWL 2.0 syntax.

- The ability of the framework to take as input and transform each one of the two available versions of the XML Schema, i.e. the 1.0 and the 1.1, in OWL 2.0 syntax.

- The implementation of a sequence of algorithms and methods that can be reused in the context of the XS2OWL transformation model or outside of it, in the future. Every algorithm is designed in order to implement a small autonomous part of the underlying logic, making it possible to be reused (e.g. the XPath Evaluator algorithm).

In conclusion, XS2OWL 2.0 exploits the OWL 2.0 semantics and supports the new XML constructs introduced by XML Schema 1.1.

The contribution of this thesis in the SPARQL2XQuery framework is:

- The SPARQL2XQuery 2.0 framework may work with both existing ontologies and with automatically produced ones, formed according to the XS2OWL 2.0 transformation model. In this case, the SPARQL2XQuery 2.0 exploits the new mapping ontology and generates enriched mappings between the XML Schema and the ontology in order to optimize the translation process that takes place in the SPARQL – to – XQuery translation.

- The new mapping XML document that supports the definition of mappings between the identity constraints of an XML Schema and the ontology constructs. Thus, the SPARQL2XQuery 2.0 framework allows the translation of even more sophisticated SPARQL queries to XQuery syntax.

- The OWL 2.0 semantics and supports the new XML constructs introduced by XML Schema 1.1. In particular, the XML Schema identity constraints can now be accurately represented in OWL 2.0 syntax (which was not possible in OWL 1.0 syntax), thus overcoming the most important limitation of the XS2OWL 1.0 framework [TsCh07][TsChODB07] and, consequently, of the SPARQL2XQuery 1.0 framework. Moreover, the XS2OWL 2.0 allows defining datatypes in OWL 2.0 syntax and consequently represent accurately the XML Schema simple types in OWL 2.0 syntax, thus overcoming the limitation in the query translation model of SPARQL2XQuery 1.0, which did not support simple types.

The contribution of this thesis in the additional queries supported by SPARQL2XQuery 2.0 is presented in the rest of this section. Thus, the additional queries supported by the SPARQL2XQuery 2.0 framework are:

- Queries that are based on specific datatypes and literals of a specific datatype.
- Queries that include onto-triples about the defined datatypes within the ontology.
- Queries that use the equality checking between complex types. This type of queries is based on the identity constraints of the unique and key constructs. Thus, the constraints are used to specify the primary key of a type and the equality conditions that apply to it.
- Sophisticated queries that exploit the references and relations among the individuals. These relations are described by the key and keyref pairs that are declared within the XML Schema.

6.2. Future work

The current research focuses on the semantic interoperability between the Semantic Web and the XML environments. In particular, this thesis presented the SPARQL2XQuery 2.0 framework, which allows expressing semantic queries on top of XML data through the translation of SPARQL queries in XQuery syntax and the XS2OWL 2.0 framework for uplifting XML Schemas to OWL 2 ontologies.

The future work of this thesis consists of:

- The optimization of the SPARQL-to-XQuery translation as far as the presented methods and algorithms are concerned by applying several optimization techniques during the SPARQL-to-XQuery translation activity.
- The exploitation and support of the SPARQL 1.1 Query language [W3C/SPARQL1.1] in the SPARQL2XQuery 2.0 framework as it introduces a set of new features (e.g. aggregates, subqueries, expressions in the SELECT clause etc.).
- The integration of the described frameworks within an ontology-based mediator framework [Ma&al09][Ma&al10] that is under development in the MUSIC laboratory and is

going to provide semantic interoperability and integration between distributed heterogeneous sources using the standard Semantic Web and XML technologies.

7. References

[AnIvMar07] Anicic N., Ivezic N., Marjanovic Z.: “Mapping XML Schema to OWL”, Enterprise Interoperability, Springer London 2007.

[BerkeleyDB] Oracle Berkeley DB XML. http://www.oracle.com/us/products/database/berkeley-db/xml/index.html

[Bik08] Bikakis N., Gioldasis N., Tsinaraki C., Christodoulakis S.: "The SPARQL2XQuery Framework ". Technical Report MUSIC/TUC Nov. 2008. http://www.music.tuc.gr/reports/SPARQL2XQUERY.PDF

[BiGiTsCho9] Bikakis N., Gioldasis N., Tsinaraki C., Christodoulakis S.: "Querying XML Data with SPARQL" In Proc. of DEXA 2009.

[BiGiTsChWSKS] Bikakis N., Gioldasis N., Tsinaraki C., Christodoulakis S.: "Semantic Based Access over XML Data" In Proc. of 2nd World Summit on Knowledge Society 2009 (WSKS 2009).

[BoAu05] Hannes Bohring, Sören Auer: “Mapping XML to OWL Ontologies”. Proceedings of 13. Leipziger Informatik-Tage (LIT 2005), Sep. 21-23, 2005.

[CarISWC03] Jeremy J. Carroll: “Pulling XML Events to Parse RDF”. ISWC 2003.

[CarWWW02] Jeremy J. Carroll: “Unparsing RDF/XML”. WWW2002, May 7-11, 2002, Honolulu, Hawaii, USA.

[ChSiPuMPEG7] Chang, S.F., Sikora, T., Puri, A.: Overview of the MPEG-7 standard. In IEEE Transactions on Circuits and Systems for Video Technology 11:688–695, 2001.

[CIDOC/CRM] ISO 21127:2006 Information and documentation – A reference ontology for the interchange of cultural heritage information (CIDOC/CRM)

[CrNiChr08] Christophe Cruz, Christophe Nicolle: “Ontology Enrichment and Automatic Population From XML Data”. VLDB ’08, August 24-30, 2008.

[CrumShOWA] Nick Drummond, Rob Shearer: “The Open World Assumption”. The University of Manchester.

[DmAlAtWg] Carlos Viegas Damásio, Anastasia Analyti, Grigoris Antoniou, Gerd Wagner: “Supporting Open and Closed World Reasoning on the Web”.

[EAD] Official EAD Version 2002 Web Site - http://www.loc.gov/ead/

[FeZiTr04] Ferdinand M., Zirpins C., Trastour D.: “Lifting XML Schema to OWL”. In the proceedings of the International Conference on Web Engineering (ICWE) 2004: 354-358.

[FOAF] FOAF Vocabulary Specification 0.98 - Namespace Document 9 August 2010. (http://xmlns.com/foaf/spec/)

[GaCe05] Garcia R., Celma O.: “Semantic Integration and Retrieval of Multimedia Metadata”. In the proceedings of the Knowledge Markup and Semantic Annotation Workshop, Semannot’05. CEUR, 2005.

[GutSPARQL] ClaudioGutierrez: “Formal Introduction to SPARQL”. Department of Computer Science, Universidad de Chile.

[Jena] Jena: A Semantic Web Framework for Java. http://jena.sourceforge.net

[Ma&al09] Makris K., Bikakis N., Gioldasis N., Tsinaraki C., Christodoulakis S.: "Towards a Mediator based on OWL and SPARQL". In Proc. of WSKS 2009.

[Ma&al10] Makris K., Gioldasis N., Bikakis N., Christodoulakis S.: "Ontology Mapping and SPARQL Rewriting for Querying Federated RDF Data Sources". In Proc. of ODBASE 2010.

[METS] Metadata Encoding and Transmission Standard (METS) Official Website. (http://www.loc.gov/standards/mets/)

[MiSeRe] L. Miller, A. Seaborne, A. Reggiori: "Three implementations of SquishQL, a Simple RDF Query Language" (2002).

[MPEG7] Chang, S.F., Sikora, T., Puri, A.: Overview of the MPEG-7 standard. In IEEE Transactions on Circuits and Systems for Video Technology 11:688–695, 2001.

[MPEG21] ISO/IEC: 21000-7:2004 – Information Technology – Multimedia Framework (MPEG-21) – Part 7: Digital Item Adaptation, 2004.

[NoyMcG101] Natalya F. Noy and Deborah L. McGuinness , Ontology Development 101: A Guide to Creating Your First Ontology.

[PerArGutCom] J. Perez, M. Arenas, and C. Gutierrez. “Semantics and Complexity of SPARQL”. - ACM Transactions on Database Systems (TODS) Volume 34 Issue 3, August 2009 Article 16

[PerArGutSem] J.Perez, M.Arenas and C.Gutierrez: “Semantics of SPARQL” - Tech. rep., Universidad de Chile. Department of Computer Science, Universidad de Chile, TR/DCC-2006-17 http://ing.utalca.cl/~jperez/papers/sparql_semantics.pdf

[PerMPEG21] Pereira, F.: The MPEG-21 standard: Why an open multi-media framework?. In the Proc. of the 8th IDMS, 2001.

[RoRoCar06] Rodrigues T., Rosa P, Cardoso J.: “Mapping XML to Exiting OWL ontologies”, International Conference WWW/Internet 2006, Murcia, Spain, 5-8 October 2006.

[StTsBiGiCh] Stavrakantonakis I., Tsinaraki C., Bikakis N., Gioldasis N., Christodoulakis S.: "SPARQL2XQuery 2.0: Supporting Semantic-based Queries over XML Data". In the Proceedings of SMAP2010

[TEI] The TEI site - http://www.tei-c.org/

[ThLeLe] P.T.T. Thuy, Y.-K. Lee, S.Lee.: "DTD2OWL: Automatic Transforming XML Documents into OWL Ontology". In ICIS '09: Proceedings of the 2nd International Conference on Interaction Sciences, pages 125–131, New York, NY, USA, 2009. ACM.

[TimBus] Tim Berners-Lee: Talk titled "Cracks and Mortar" – 2007. (http://www.w3.org/2007/Talks/1107-tpac-tbl/)

[TimN3] Tim Berners-Lee: "Primer: Getting into RDF & Semantic Web using N3". (http://www.w3.org/2000/10/swap/Primer)

[TimWW] Tim Berners-Lee, Mark Fischetti: "Weaving the Web". 1999

[TsCh07] Tsinaraki C., Christodoulakis S.: "Support for Interoperability between OWL based and XML Schema based Applications". 2nd DELOS Conference On Digital Libraries, Tirrenia, Italy, December 2007.

[TsChODB07] Tsinaraki C., Christodoulakis S. "Interoperability of XML Schema Applications with OWL Domain Knowledge and Semantic Web Tools", In the proceedings of the 6th International Conference on Ontologies, DataBases, and Applications of Semantics (ODBASE 2007), OTM Conferences (1) 2007: 850-869, Vilamoura, Algarve, Portugal, November 27-29 2007.

[TsChXS2OWL] Tsinaraki C., Christodoulakis S. "XS2OWL: A Formal Model and a System for enabling XML Schema Applications to interoperate with OWL-DL Domain Knowledge and Semantic Web Tools" , In the proceedings of the 1st DELOS Conference, pp. 124-136, Tirrenia, Italy, February 2007.

[TsPhD08] Tsinaraki C . "A Semantic Based Framework for Multimedia Management and Interoperability" , PhD Thesis Technical University of Crete, ECE Department, Chania 2008.

[TsPoCh07] Tsinaraki C., Polydoros P., Christodoulakis S.. Interoperability support between MPEG-7/21 and OWL in DS-MIRF. In Transactions on Knowledge and Data Engineering (TKDE), Special Issue on the Semantic Web Era, 19(2):219–232, 2007.

[W3C/FiXSD] Web Consortium (W3C): "W3C XML Schema Definition Language (XSD) 1.1 Part1", 3.11.6.3 sub-section: field subset of XPath. (http://www.w3.org/TR/xmlschema11-1/#c-fields-xpaths).

[W3C/OWL1] Web Consortium (W3C): “OWL Web Ontology Language Overview”. Recommendation 10 February 2004. (http://www.w3.org/TR/owl-features/)

[W3C/OWL1R] Web Consortium (W3C): “OWL Web Ontology Language Reference”. (http://www.w3.org/TR/owl-ref/)

[W3C/OWL2] Web Consortium (W3C): “OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax”. (http://www.w3.org/TR/owl2-syntax/)

[W3C/OWLDS] Web Consortium (W3C): “OWL 2 Web Ontology Language Direct Semantics”. (http://www.w3.org/TR/owl2-direct-semantics/)

[W3C/OWLMP] Web Consortium (W3C): “OWL 2 Web Ontology Language Mapping to RDF Graphs”. (http://www.w3.org/TR/owl2-mapping-to-rdf/)

[W3C/OWLNF] Web Consortium (W3C): “OWL 2 Web Ontology Language New Features and Rationale”. (http://www.w3.org/TR/owl2-new-features/)

[W3C/OWLOV] Web Consortium (W3C): “OWL 2 Web Ontology Language Document Overview”. (http://www.w3.org/TR/owl2-overview/)

[W3C/OWLPF] Web Consortium (W3C): “OWL 2 Web Ontology Language Profiles”. (http://www.w3.org/TR/owl2-profiles/)

[W3C/OWLPR] Web Consortium (W3C): “OWL 2 Web Ontology Language Primer”. (http://www.w3.org/TR/owl2-primer/)

[W3C/OWLPR] Web Consortium (W3C): “OWL 2 Web Ontology Language Primer”. (http://www.w3.org/TR/owl2-primer/)

[W3C/OWLQR] Web Consortium (W3C): “OWL 2 Web Ontology Language Quick Reference Guide”. (http://www.w3.org/TR/owl2-quick-reference/)

[W3C/OWLRD] Web Consortium (W3C): “OWL 2 Web Ontology Language RDF-Based Semantics”. (http://www.w3.org/TR/owl2-rdf-based-semantics/)

[W3C/RDF] RDF Primer - W3C Recommendation 10 February 2004 (http://www.w3.org/TR/2004/REC-rdf-primer-20040210/).

[W3C/RDF97] Web Consortium (W3C): “Resource Description Framework (RDF) Model and Syntax”. (http://www.w3.org/TR/WD-rdf-syntax-971002/).

[W3C/RDFCo] Resource Description Framework (RDF): Concepts and Abstract Syntax - W3C Recommendation 10 February 2004. (http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/).

[W3C/RDFS] Web Consortium (W3C): "RDF Vocabulary Description Language 1.0: RDF Schema". (http://www.w3.org/TR/rdf-schema/).

[W3C/SelXSD] Web Consortium (W3C): "W3C XML Schema Definition Language (XSD) 1.1 Part1", 3.11.6.2 sub-section: selector subset of XPath. (http://www.w3.org/TR/xmlschema11-1/#c-selector-xpath).

[W3C/SPARQL] Web Consortium (W3C): "SPARQL Query Language for RDF", (http://www.w3.org/TR/rdf-sparql-query/)

[W3C/SPARQL1.1] Web Consortium (W3C): "SPARQL 1.1 Query Language", (http://www.w3.org/TR/sparql11-query/)

[W3C/SW] Web Consortium (W3C) Semantic Web activity. http://www.w3.org/2001/sw

[W3C/VCard] Representing vCard Objects in RDF - W3C Member Submission 20 January 2010. (http://www.w3.org/TR/vcard-rdf/)

[W3C/XML] Extensible Markup Language (XML) 1.0 (Fifth Edition) - W3C Recommendation 26 November 2008. (http://www.w3.org/TR/2008/REC-xml-20081126/).

[W3C/XPATH] Web Consortium (W3C): "XML Path Language (XPath) Version 1.0" - 16 November 1999. (http://www.w3.org/TR/xpath/).

[W3C/XPATH2] Web Consortium (W3C): "XML Path Language (XPath) 2.0". (http://www.w3.org/TR/xpath20/).

[W3C/XPointer] XML Pointer Language (XPointer) - W3C Working Draft 16 August 2002. (http://www.w3.org/TR/xptr/).

[W3C/XQUERY] XQuery 1.0: An XML Query Language - W3C Recommendation 23 January 2007. (http://www.w3.org/TR/xquery/).

[W3C/XQXPfn] XQuery 1.0 and XPath 2.0 Functions and Operators - W3C Recommendation 23 January 2007. (http://www.w3.org/TR/xquery-operators/).

[W3C/XSD1.1a] Web Consortium (W3C): "W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures". (http://www.w3.org/TR/xmlschema11-1/).

[W3C/XSD1.1b] Web Consortium (W3C): "W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes". (http://www.w3.org/TR/xmlschema11-2/).

[W3C/XSD1] XML Schema Part 0: Primer Second Edition W3C Recommendation 28 October 2004 (http://www.w3.org/TR/xmlschema-0/).

[W3C/XSD1dat] XML Schema Part 2: Datatypes Second Edition - W3C Recommendation 28 October 2004. (http://www.w3.org/TR/xmlschema-2/).

[W3C/XSD1str] XML Schema Part 1: Structures Second Edition - W3C Recommendation 28 October 2004. (http://www.w3.org/TR/xmlschema-1/).

[W3C/XSLT] Web Consortium (W3C): "XSL Transformations (XSLT) Version 2.0". (http://www.w3.org/TR/xslt20/).

[XMLBeans] Apache Foundation: "XML Beans – XML binding framework". http://xmlbeans.apache.org

[XpathOverXS] XPath over XML Schema for Java. http://xpath-on-schema.sourceforge.net/ARQ

[XqFunctX] FunctX XQuery functions. http://www.xqueryfunctions.com/xq

[ZiDaCeCoEl] Zilli A., Damiani E., Ceravolo P., Corallo A., Elia G.(eds.): "Semantic Knowledge Management: An Ontology-based Framework". Information Science Reference 2008

Appendix A. Mappings Representation

This appendix provides the XML Schema syntax which describes the XML Schema followed by the SPARQL2XQuery 2.0 mappings and specifies the XML structures in which the mappings are stored during the mapping process of SPARQL2XQUERY according to the type of each construct.

The first child of the "**MappingFile**" root element is the element "**MappingInformation**", holding the URIs of the Ontology and the XML Schema being mapped. The siblings of the "MappingInformation" element specify the mappings between the OWL 2.0 constructs and the XML Schema constructs of the Ontology and the XML Schema, respectively. In particular, the "**Class**" element includes an attribute with the IRI of the class and also includes a sequence of "XPath" elements that hold the possible paths of the equivalent XML Schema ComplexType instances. In case of Datatype and Object properties, the mapping elements "**DataTypeProperty**" and "**ObjectProperty**", respectively, contain the XPaths defined within the elements DomainXpath and RangeXpath referring to the domain and range, respectively, of the specific properties.

Finally, the SPARQL2XQuery extensions of this thesis require a mapping construct for the datatypes of the Ontology and the Simple Types of the XML Schema. This is accomplished by the element "**Datatype**" of the mapping Schema. The syntax of this element is similar to that of the "Class" element. In the rest of this appendix is provided the XML Schema of the mapping document.

Appendix A: Mappings Representation

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns="http://www.music.tuc.gr/xs2owl/mappingfile"
targetNamespace="http://www.music.tuc.gr/xs2owl/mappingfile" elementFormDefault="qualified">

   <xs:complexType name="xpaths_type">
     <xs:sequence>
          <xs:element name="Domain_Xpath" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
          <xs:element name="Range_Xpath" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
     </xs:sequence>
  </xs:complexType>


    <xs:complexType name="PropertyPair_type">
      <xs:sequence>
           <xs:element type="xs:string" name="keyrefMember" minOccurs="1" maxOccurs="1"/>
           <xs:element type="xs:string" name="keyMember" minOccurs="1" maxOccurs="1"/>
      </xs:sequence>
  </xs:complexType>

  <xs:complexType name="Keys_type">
    <xs:sequence>
     <xs:element name="PropertyPair" type="PropertyPair_type" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="onClass" type="xs:string"/>
  </xs:complexType>

  <xs:complexType name="Property_Type">
          <xs:sequence>
                  <xs:element name="Xpaths" type="xpaths_type"/>
                  <xs:element name="IC" type="Keys_type" minOccurs="0" maxOccurs="1"/>
          </xs:sequence>
          <xs:attribute name="name" type="xs:string"/>
  </xs:complexType>

  <xs:complexType name="Class_Type">
          <xs:sequence>
                  <xs:element name="Xpath" type="xs:string" maxOccurs="unbounded"/>
          </xs:sequence>
          <xs:attribute name="name" type="xs:string"/>
  </xs:complexType>

  <xs:complexType name="Datatype_Type">
          <xs:sequence>
                  <xs:element name="Xpath" type="xs:string" maxOccurs="unbounded"/>
          </xs:sequence>
          <xs:attribute name="name" type="xs:string"/>
  </xs:complexType>
```

```
<xs:complexType name="mapping_info_type">
        <xs:sequence>
                <xs:element name="OntologyURI" type="xs:anyURI"/>
                <xs:element name="XMLSchemaURI" type="xs:anyURI"/>
        </xs:sequence>
</xs:complexType>

<xs:element name="MappingFile">
   <xs:complexType>
      <xs:sequence>
        <xs:element name="MappingInformation" type="mapping_info_type"/>
        <xs:element name="Class" type="Class_Type" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="Datatype" type="Datatype_Type" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="ObjectProperty" type="Property_Type" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="DataTypeProperty" type="Property_Type" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
</xs:element>
</xs:schema>
```

Appendix B. Mappings Representation Example

This appendix provides the representation of the mappings for the XML Schema presented in Figure 2.12 and the corresponding ontology. For the mapping representation, follow the XML Schema discussed in the Appendix A.

```
<?xml version="1.0" encoding="UTF-8"?>
<map:MappingFile xmlns:map="http://www.music.tuc.gr/xs2owl/mappingfile">
   <map:MappingInformation>

<map:OntologyURI>http://www.music.tuc.gr/xs2owl/querying/mappingfile#</map:OntologyURI>

<map:XMLSchemaURI>C:\CodingBox\person_SimpleTypesSmall.xsd</map:XMLSchemaURI>
   </map:MappingInformation>
   <map:Class name="EmployeeType">
      <map:Xpath>/Persons/Employee</map:Xpath>
   </map:Class>
   <map:Class name="PersonsType">
      <map:Xpath>/Persons</map:Xpath>
   </map:Class>
   <map:Class name="AddressType">
      <map:Xpath>/Persons/Person/Address</map:Xpath>
      <map:Xpath>/Persons/Employee/Address</map:Xpath>
   </map:Class>
```

```
<map:Class name="PersonType">
   <map:Xpath>/Persons/Person</map:Xpath>
   <map:Xpath>/Persons/Employee</map:Xpath>
</map:Class>
<map:Class name="NameType">
   <map:Xpath>/Persons/Person/Name</map:Xpath>
   <map:Xpath>/Persons/Employee/Name</map:Xpath>
</map:Class>
<map:Class name="IDType">
   <map:Xpath>/Persons/Person/ID</map:Xpath>
   <map:Xpath>/Persons/Person/Knows</map:Xpath>
   <map:Xpath>/Persons/Employee/ID</map:Xpath>
   <map:Xpath>/Persons/Employee/Knows</map:Xpath>
</map:Class>
<map:Datatype name="validAgeType">
   <map:Xpath>/Persons/Person/Age</map:Xpath>
   <map:Xpath>/Persons/Employee/Age</map:Xpath>
</map:Datatype>
<map:ObjectProperty name="Address__AddressType">
   <map:Xpaths>
      <map:Domain_Xpath>/Persons/Person</map:Domain_Xpath>
      <map:Domain_Xpath>/Persons/Employee</map:Domain_Xpath>
      <map:Range_Xpath>/Persons/Person/null</map:Range_Xpath>
      <map:Range_Xpath>/Persons/Employee/null</map:Range_Xpath>
   </map:Xpaths>
</map:ObjectProperty>
<map:ObjectProperty name="ID__IDType">
   <map:Xpaths>
      <map:Domain_Xpath>/Persons/Person</map:Domain_Xpath>
      <map:Domain_Xpath>/Persons/Employee</map:Domain_Xpath>
      <map:Range_Xpath>/Persons/Person/null</map:Range_Xpath>
      <map:Range_Xpath>/Persons/Employee/null</map:Range_Xpath>
   </map:Xpaths>
</map:ObjectProperty>
<map:ObjectProperty name="Persons__PersonsType">
   <map:Xpaths>
      <map:Range_Xpath>/Persons</map:Range_Xpath>
   </map:Xpaths>
</map:ObjectProperty>
<map:ObjectProperty name="Employee__EmployeeType">
   <map:Xpaths>
      <map:Domain_Xpath>/Persons</map:Domain_Xpath>
      <map:Range_Xpath>/Persons/null</map:Range_Xpath>
   </map:Xpaths>
</map:ObjectProperty>
```

Appendix B: Mappings Representation Example

```
<map:ObjectProperty name="Knows__IDType">
  <map:Xpaths>
    <map:Domain_Xpath>/Persons/Person</map:Domain_Xpath>
    <map:Domain_Xpath>/Persons/Employee</map:Domain_Xpath>
    <map:Range_Xpath>/Persons/Person/null</map:Range_Xpath>
    <map:Range_Xpath>/Persons/Employee/null</map:Range_Xpath>
  </map:Xpaths>
</map:ObjectProperty>
<map:ObjectProperty name="Name__NameType">
  <map:Xpaths>
    <map:Domain_Xpath>/Persons/Person</map:Domain_Xpath>
    <map:Domain_Xpath>/Persons/Employee</map:Domain_Xpath>
    <map:Range_Xpath>/Persons/Person/null</map:Range_Xpath>
    <map:Range_Xpath>/Persons/Employee/null</map:Range_Xpath>
  </map:Xpaths>
</map:ObjectProperty>
<map:ObjectProperty name="Person__PersonType">
  <map:Xpaths>
    <map:Domain_Xpath>/Persons</map:Domain_Xpath>
    <map:Range_Xpath>/Persons/null</map:Range_Xpath>
  </map:Xpaths>
</map:ObjectProperty>
<map:ObjectProperty name="Age__validAgeType">
  <map:Xpaths>
    <map:Domain_Xpath>/Persons/Person</map:Domain_Xpath>
    <map:Domain_Xpath>/Persons/Employee</map:Domain_Xpath>
    <map:Range_Xpath>/Persons/Person/null</map:Range_Xpath>
    <map:Range_Xpath>/Persons/Employee/null</map:Range_Xpath>
  </map:Xpaths>
</map:ObjectProperty>
<map:DataTypeProperty name="city__xs_string">
  <map:Xpaths>
    <map:Domain_Xpath>/Persons/Person/Address</map:Domain_Xpath>
    <map:Domain_Xpath>/Persons/Employee/Address</map:Domain_Xpath>
    <map:Range_Xpath>/Persons/Person/Address/@city</map:Range_Xpath>
    <map:Range_Xpath>/Persons/Employee/Address/@city</map:Range_Xpath>
  </map:Xpaths>
</map:DataTypeProperty>
<map:DataTypeProperty name="country__xs_string">
  <map:Xpaths>
    <map:Domain_Xpath>/Persons/Person/Address</map:Domain_Xpath>
    <map:Domain_Xpath>/Persons/Employee/Address</map:Domain_Xpath>
    <map:Range_Xpath>/Persons/Person/Address/@country</map:Range_Xpath>
    <map:Range_Xpath>/Persons/Employee/Address/@country</map:Range_Xpath>
  </map:Xpaths>
```

```
</map:DataTypeProperty>
<map:DataTypeProperty name="Number__xs_nonNegativeInteger">
  <map:Xpaths>
    <map:Domain_Xpath>/Persons/Person/Address</map:Domain_Xpath>
    <map:Domain_Xpath>/Persons/Employee/Address</map:Domain_Xpath>
    <map:Range_Xpath>/Persons/Person/Address/Number</map:Range_Xpath>
    <map:Range_Xpath>/Persons/Employee/Address/Number</map:Range_Xpath>
  </map:Xpaths>
</map:DataTypeProperty>
<map:DataTypeProperty name="Person__PersonType">
  <map:Xpaths>
    <map:Range_Xpath>/Person</map:Range_Xpath>
  </map:Xpaths>
</map:DataTypeProperty>
<map:DataTypeProperty name="Colleague__xs_IDREF">
  <map:Xpaths>
    <map:Domain_Xpath>/Persons/Person</map:Domain_Xpath>
    <map:Domain_Xpath>/Persons/Employee</map:Domain_Xpath>
    <map:Range_Xpath>/Persons/Person/@Colleague</map:Range_Xpath>
    <map:Range_Xpath>/Persons/Employee/@Colleague</map:Range_Xpath>
  </map:Xpaths>
</map:DataTypeProperty>
<map:DataTypeProperty name="Address__AddressType">
  <map:Xpaths>
    <map:Range_Xpath>/Address</map:Range_Xpath>
  </map:Xpaths>
</map:DataTypeProperty>
<map:DataTypeProperty name="PassportID__xs_nonNegativeInteger">
  <map:Xpaths>
    <map:Domain_Xpath>/Persons/Person/ID</map:Domain_Xpath>
    <map:Domain_Xpath>/Persons/Person/Knows</map:Domain_Xpath>
    <map:Domain_Xpath>/Persons/Employee/ID</map:Domain_Xpath>
    <map:Domain_Xpath>/Persons/Employee/Knows</map:Domain_Xpath>
    <map:Range_Xpath>/Persons/Person/ID/PassportID</map:Range_Xpath>
    <map:Range_Xpath>/Persons/Person/Knows/PassportID</map:Range_Xpath>
    <map:Range_Xpath>/Persons/Employee/ID/PassportID</map:Range_Xpath>
    <map:Range_Xpath>/Persons/Employee/Knows/PassportID</map:Range_Xpath>
  </map:Xpaths>
</map:DataTypeProperty>
<map:DataTypeProperty name="Employee__EmployeeType">
  <map:Xpaths>
    <map:Range_Xpath>/Employee</map:Range_Xpath>
  </map:Xpaths>
</map:DataTypeProperty>
<map:DataTypeProperty name="IdentityCardID__xs_string">
```

```
    <map:Xpaths>
      <map:Domain_Xpath>/Persons/Person/ID</map:Domain_Xpath>
      <map:Domain_Xpath>/Persons/Person/Knows</map:Domain_Xpath>
      <map:Domain_Xpath>/Persons/Employee/ID</map:Domain_Xpath>
      <map:Domain_Xpath>/Persons/Employee/Knows</map:Domain_Xpath>
      <map:Range_Xpath>/Persons/Person/ID/IdentityCardID</map:Range_Xpath>
      <map:Range_Xpath>/Persons/Person/Knows/IdentityCardID</map:Range_Xpath>
      <map:Range_Xpath>/Persons/Employee/ID/IdentityCardID</map:Range_Xpath>
      <map:Range_Xpath>/Persons/Employee/Knows/IdentityCardID</map:Range_Xpath>
    </map:Xpaths>
  </map:DataTypeProperty>
  <map:DataTypeProperty name="FirstName__xs_string">
    <map:Xpaths>
      <map:Range_Xpath>/FirstName</map:Range_Xpath>
    </map:Xpaths>
  </map:DataTypeProperty>
  <map:DataTypeProperty name="prefix__xs_string">
    <map:Xpaths>
      <map:Domain_Xpath>/Persons/Person/Name</map:Domain_Xpath>
      <map:Domain_Xpath>/Persons/Employee/Name</map:Domain_Xpath>
      <map:Range_Xpath>/Persons/Person/Name/@prefix</map:Range_Xpath>
      <map:Range_Xpath>/Persons/Employee/Name/@prefix</map:Range_Xpath>
    </map:Xpaths>
  </map:DataTypeProperty>
  <map:DataTypeProperty name="ID__IDType">
    <map:Xpaths>
      <map:Range_Xpath>/ID</map:Range_Xpath>
    </map:Xpaths>
  </map:DataTypeProperty>
  <map:DataTypeProperty name="Surname__xs_string">
    <map:Xpaths>
      <map:Range_Xpath>/Surname</map:Range_Xpath>
    </map:Xpaths>
  </map:DataTypeProperty>
  <map:DataTypeProperty name="Age__validAgeType">
    <map:Xpaths>
      <map:Range_Xpath>/Age</map:Range_Xpath>
    </map:Xpaths>
  </map:DataTypeProperty>
  <map:DataTypeProperty name="salary__xs_integer">
    <map:Xpaths>
      <map:Domain_Xpath>/Persons/Employee</map:Domain_Xpath>
      <map:Range_Xpath>/Persons/Employee/salary</map:Range_Xpath>
    </map:Xpaths>
  </map:DataTypeProperty>
```

```
  <map:DataTypeProperty name="Street__xs_string">
    <map:Xpaths>
      <map:Domain_Xpath>/Persons/Person/Address</map:Domain_Xpath>
      <map:Domain_Xpath>/Persons/Employee/Address</map:Domain_Xpath>
      <map:Range_Xpath>/Persons/Person/Address/Street</map:Range_Xpath>
      <map:Range_Xpath>/Persons/Employee/Address/Street</map:Range_Xpath>
    </map:Xpaths>
  </map:DataTypeProperty>
  <map:DataTypeProperty name="Name__NameType">
    <map:Xpaths>
      <map:Range_Xpath>/Name</map:Range_Xpath>
    </map:Xpaths>
  </map:DataTypeProperty>
  <map:DataTypeProperty name="RegID__xs_ID">
    <map:Xpaths>
      <map:Domain_Xpath>/Persons/Person</map:Domain_Xpath>
      <map:Domain_Xpath>/Persons/Employee</map:Domain_Xpath>
      <map:Range_Xpath>/Persons/Person/@RegID</map:Range_Xpath>
      <map:Range_Xpath>/Persons/Employee/@RegID</map:Range_Xpath>
    </map:Xpaths>
  </map:DataTypeProperty>
  <map:DataTypeProperty name="Knows__IDType">
    <map:Xpaths>
      <map:Range_Xpath>/Knows</map:Range_Xpath>
    </map:Xpaths>
  </map:DataTypeProperty>
  <map:DataTypeProperty name="postCode__xs_integer">
    <map:Xpaths>
      <map:Domain_Xpath>/Persons/Person/Address</map:Domain_Xpath>
      <map:Domain_Xpath>/Persons/Employee/Address</map:Domain_Xpath>
      <map:Range_Xpath>/Persons/Person/Address/@postCode</map:Range_Xpath>
      <map:Range_Xpath>/Persons/Employee/Address/@postCode</map:Range_Xpath>
    </map:Xpaths>
  </map:DataTypeProperty>
</map:MappingFile>
```