University of Liege
Faculty of Applied Sciences

Using Knowledge Graph Technologies to Contextualize and Validate Declarations in the Social Security Domain

Master Thesis carried out to obtain the degree of Master of Science in Computer Science and Engineering

Author
CHIEM DAO Davan

Supervisor
DEBRUYNE Christophe

Academic year 2022-2023

Abstract

This master's thesis explores the feasibility and potential of knowledge graph technologies, with a specific emphasis on the Shapes Constraint Language (SHACL), for validating declaration forms in the Belgian Social Security domain. The study focused on validating DmfA declarations which are reports describing the work of employees done during a quarter sent by an employer to the government. The study first created a vocabulary for a knowledge graph based on DmfA declarations and the mapping of diverse data sources into Resource Description Framework (RDF) format. The heart of the work lies in the declaration of SHACL constraints for complex business rules, many of which were implemented via SPARQL, highlighting SHACL's scalability and versatility in addressing challenging validation tasks. Despite obstacles concerning data transformation while generating RDF representations of the declarations, the research demonstrates that SHACL provides a richer application profile and exceeds the expressiveness of XML Schema Definition (XSD) constraints in implementing complex validation rules. These findings illustrate the potential of knowledge graph technologies and SHACL for managing intricate validation tasks in social security systems and hint at the prospect of their application in future work, such as evolving document integration and validation report integration using named graphs and provenance information.

Acknowledgement

I would like to express my sincere gratitude to my supervisor, Professor Christophe Debruyne, for his invaluable guidance and encouragement throughout my thesis. His expertise and support have been crucial in this academic endeavor.

I also wish to extend heartfelt thanks to my family. Their persistent encouragement, unwavering support, and enduring faith in my abilities pushed me to strive for and achieve the highest goals.

Contents

1 Introduction

In an era where digital transformation becomes increasingly important, the demand for advanced data management techniques echoes across all sectors. This is particularly true in social security, a domain marked by intricate regulations and diverse data structures. Knowledge graph technologies emerge as a solution to these complexities.

To better evaluate these technologies, particularly the Shapes Constraint Language (SHACL), a Knowledge Graph was constructed with a focus on validating social security declarations. This evaluation forms the primary crux of this thesis, guiding our exploration of SHACL's potential and our creation of the knowledge graph itself.

The thesis starts by establishing a robust theoretical foundation, and familiarizing the reader with core concepts such as knowledge graphs, RDF, RML, SHACL, and other related technologies. This is followed by a deep dive into the Belgian social security, delving into the intricacies of e-governments and DmfA declarations. A section dedicated to related work grounds the research within the larger academic discourse.

At the core of the thesis is the Approach Section, which offers a detailed exploration of the methodologies adopted during the study. This includes steps such as generating a dedicated vocabulary, mapping DmfA declarations to RDF, developing SHACL rules, and setting up a thorough validation process.

Subsequently, the discussion section delves into critically evaluating of our approach and the knowledge graph technologies employed. We assess the advantages and disadvantages of our method in creating and maintaining the knowledge graph, underlining the strengths and challenges associated with SHACL and related technologies for validating data.

To conclude, the final chapter provides a succinct summary of the thesis, recounting the key steps and achievements of our work. The conclusion also looks forward, highlighting potential improvements and areas for future exploration that could continue to advance this field of study.

2 Theoretical Background

For the reader's convenience, the following chapter will present and describe all the theoretical terms linked to knowledge graph technologies that will be mentioned in this thesis. These terms will be broadly explained but they should be sufficient to understand the remainder of the thesis.

2.1 Knowledge Graph

In the literature, many have suggested definitions for knowledge graphs (KGs) that vary from general and inclusive definitions to technically detailed definitions. In [1], Hogan et al. reviewed existing definitions, and similarly to Noy et al. in [2], define KG as a "graph where nodes represent entities, and edges represent relationships between those entities". This is a general definition to which they add three criteria that are optional but that most KG should meet:

- An ontology (cf. 2.2) formally describes the relationships and the types of entities.
- Knowledge is integrated from various sources.
- New facts can be inferred from the graph thanks to a formal representation.

In this thesis, these three criteria should be considered compulsory for a KG.

2.2 Ontology

Etymologically the word ontology comes from the two Greek words *óntos* meaning being and *lógos* meaning study. Thus, its etymological meaning is the study of being in the world. Attempting to describe the world and its concepts echoes the computer science definition of an ontology proposed by Studer et al. in [3] as "a **formal**, **explicit** specification of a **shared conceptualization**". A conceptualization is an abstract representation of things or phenomena in the real world. **Formal** means that an ontology should be specified based on some mathematics or logic to be computer-readable and allow reasoning. **Explicit** refers to the fact that an ontology should be described by an external document to be shared and accessed by computer agents. To be useful, the symbols used in the formalism and the representation of the conceptualization in this formalism must be **shared** among a group rather than depicting specificities of individuals.

Depending on the formalism that is used to build ontologies, one can distinguish vocabularies from fully-fledged ontologies. While the term "ontologies" usually refers to highly-axiomatized ontologies, the term "vocabulary" is used to describe ontologies with only a few axions such as class hierarchies, domains, and ranges. Hence, vocabularies are a subset of ontologies.

All types of ontologies aim to achieve semantic interoperability by defining and describing concepts of a particular domain. This semantic interpretability is a necessary condition to achieve semantic interoperability which is the ability to exchange data between information systems in such a way that all systems interpret the data with the correct meaning.

2.3 RDF

The Resource Description Framework (RDF) is a W3C standard framework used to represent information on the Web [4]. The framework mainly provides an abstract graph-data model for data representation which is the base of many KG technologies such as RML and SHACL. This model defines two structures: RDF graphs and RDF datasets. Whereas RDF is an abstract model, there are multiple RDF serializations. Some serializations only support RDF graphs (e.g., RDF/XML, Turtle), others support named graphs, i.e., RDF datasets (e.g., TriG, N-Quads).

An RDF graph is a set of RDF triples consisting of a subject, a predicate, and an object. This graph can be viewed as a directed and labelled graph where the subject, predicate, and object are, respectively, the source vertex, arc from source to target vertices, and target vertex.

Two representations of a simple RDF graph are illustrated in Figure [1]. On the left, the graph is visually represented. While on the left, the equivalent graph is serialized in Turtle [5], a terse serialization format of RDF. Lines 3-4 describe a triple stating that the resource identified by `<ex:davanchiemdao>` is related to the literal "Davan" by the relationship `<ex:firstname>`. One can guess that the first name of the resource `<ex:davanchiemdao>` is Davan. However, this assumption cannot be confirmed solely from the graph as RDF is only a data model. We need to rely on ontology languages such as RDFS [6] and OWL [7]. These languages allow us to represent and thus provide ontological information on the semantics of the `<ex:firstname>` predicate. In other words, RDF allows us to represent and store data as a graph, and ontology languages add a meaningful "layer" allowing both humans and computers to interpret it. That layer is also represented and stored as a graph.

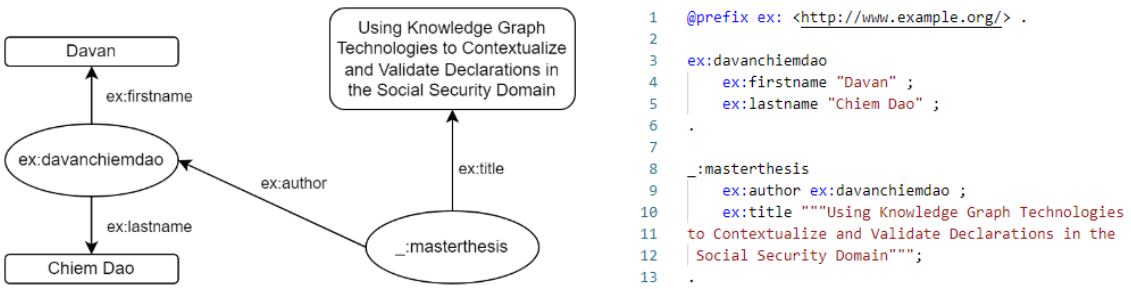


Figure 1: RDF example, graphical representation (left) and TURTLE serialization (right)

Some additional specifications can be mentioned about the RDF model. Different sets of terms can act as subjects, predicates, and objects (see Table 1). These sets are a combination of Internationalized Resource Identifier (IRI), blank node, and literal. IRIs, as its name suggest, identify resources on the Web meaning that resources sharing the same IRI are the same resource. Blank nodes, on the other hand, refer to anonymous resources; resources that cannot be identified or for which we do not know the identifier. Nevertheless, some concrete RDF syntaxes introduce blank node identifiers which are artificial identifiers that locally identify blank nodes. Thus, these identifiers are not portable across systems. They are not even portable across RDF graphs within an RDF dataset. Literals do not refer to resources but to constants. They are either plain strings with an optional language tag or strings with a type identified by an IRI.

Table 1: Allowed term for RDF triple component

	IRI	**Blank node**	**Literal**
Subject	Yes	Yes	No
Predicate	Yes	No	No
Object	Yes	Yes	Yes

The second data structure, the RDF dataset, is a set of RDF graphs. This set contains a single default graph which is an RDF graph and zero or more named graphs which are pairs of a graph name and an RDF graph. In other words, RDF datasets are a set of named RDF graphs. They are also viewed as a set of quadruples made of a triple (as in an RDF graph) and the optional name of the graph the triple belongs to. The N-Quads [8] format notably encodes RDF datasets using these quadruples.

2.4 Direct Mapping

The W3C Recommendation "A Direct Mapping of Relational Data to RDF" [9] defines a direct mapping as the representation of data in a relational database as an RDF graph. The purpose of this standard is to provide an automated way of sharing relational data on the Web of Data. It focuses on relational databases as they are the most common kind of databases and thus constitute an important source of data. However, this idea can be extended to the more general idea of representing the data in any specific format to an RDF graph.

The main disadvantage of this transformation method is that it reflects the structure of the source data and its particularities data such as the use of abbreviations, special encodings, or typos which are carried out to the RDF graph. This may hinder sharing capabilities, but it is sufficient if only an RDF graph is required.

2.5 R2RML

The Relational Database (RDB) to RDF Mapping Language (R2RML) is a W3C standardized language to express customized mappings of data from relational databases to an RDF dataset [10]. Compared to direct mapping, R2RML offers the ability to annotate the data with existing ontologies which favors interoperability.

The main concepts of R2RML will be illustrated with the example depicted in Figure [2]. It illustrates the source RDB, a custom mapping of this source in R2RML, and the resulting RDF graph. As the goal is to obtain an RDF graph, an R2RML mapping is a set of `rr:TriplesMap` instances which are resources describing how "things" of the RDB should be mapped into triples. Thus, the source data as well as mapping to the subject, predicate, and object must be described.

People

Id INTEGER PRIMARY KEY	firstName VARCHAR(100)	lastName VARCHAR(100)
1	Davan	Chiem Dao
2	John	Smith

Document

Id INTEGER PRIMARY KEY	title VARCHAR(100)	authorId INTEGER
1	Using Knowledge Graph Technologies to Contextualize and Validate Declarations in the Social Security Domain	1

```
@prefix rr: <http://www.w3.org/ns/r2rml#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix dbo: <http://dbpedia.org/ontology/> .
@base <http://www.example.org/> .

<#PersonTriplesMap>
    a rr:TriplesMap;
    rr:logicalTable [ rr:tableName "People" ] ;

    rr:subjectMap [
        rr:template "http://www.example.org/Person/{Id}" ;
        rr:class dbo:Person ;
    ] ;

    rr:predicateObjectMap [
        rr:predicate foaf:firstname ;
        rr:objectMap [ rr:column "firstName" ] ;

    rr:predicateObjectMap [
        rr:predicate foaf:surname ;
        rr:objectMap [ rr:column "lastName" ] ;
.
```

```
<#DocumentTriplesMap>
    a rr:TriplesMap;
    rr:logicalTable [ rr:tableName "Document" ] ;

    rr:subjectMap [
        rr:template "http://www.example.org/Document/{Id}" ;
        rr:class dbo:Document ;
    ] ;

    rr:predicateObjectMap [
        rr:predicate dbo:title ;
        rr:objectMap [ rr:column "title" ] ;

    rr:predicateObjectMap [
        rr:predicate dbo:author ;
        rr:objectMap [
            rr:parentTriplesMap <#PersonTriplesMap>
            rr:joinCondition[
                rr:child "authorId" ;
                rr:parent "Id" ;
            ] ;
        ] ;
    ] ;
.
```

```
<http://www.example.org/Person/1> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/ontology/Person> .
<http://www.example.org/Person/1> <http://xmlns.com/foaf/0.1/firstName> "Davan" .
<http://www.example.org/Person/1> <http://xmlns.com/foaf/0.1/surname> "Chiem Dao" .
<http://www.example.org/Person/2> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/ontology/Person> .
<http://www.example.org/Person/2> <http://xmlns.com/foaf/0.1/firstName> "Jhon" .
<http://www.example.org/Person/2> <http://xmlns.com/foaf/0.1/surname> "Smith" .
<http://www.example.org/Document/1> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/ontology/Document> .
<http://www.example.org/Document/1> <http://dbpedia.org/ontology/title> <http://dbpedia.org/ontology/document> "Using Knowledge Gr
<http://www.example.org/Document/1> <http://dbpedia.org/ontology/author> <http://www.example.org/Person/1> .
```

Figure 2: R2RML mapping example, RDB (top), R2RML mapping (middle), resulting RDF graph (bottom)

The `rr:logicalTable` predicate states which logical table serves as a sufficient subset of the RDB to produce the desired triples. These tables are the result of an SQL query to the RDB. Allowing an arbitrary SQL query result to be used as a logical table provides high flexibility in the mapping as some data transformations can be performed. In the example, both triples map considers the raw RDB tables as a logical table.

The `rr:subjectMap` predicate states how the subject of the triple should be generated. It mainly describes what the IRI or blank node is like. In the example, the resulting IRI is based on the template string where the substring `"Id"` is replaced by the value of the Id column in the logical table. Moreover, the type of the generated resources can be mentioned with the `rr:class` predicate.

The `rr:predicateObjectMap` predicate states how to generate predicates and objects. It characterizes the IRI of the predicate and uses data from the logical table to generate the object literal by mentioning the column name. When an object map references the result of the subject map of another triples map, this referenced triples map should be mentioned with the `rr:parentTriplesMap` predicate. Conceptually, the logical table of the triples map containing the referencing map is left joined with the logical table of the referenced triples map where only the lines respecting the optional `rr:joinCondition` are inserted.

2.6 RML

The RDF Mapping Language is an extension to R2RML which provides terms to express mapping from various sources of data [11]. Most RML processors support relational databases, relational data (e.g., CSV), and document models (e.g., XML, JSON). This is not an official specification standard but rather a community effort to extend mapping possibilities to RDF from different structured data, and not uniquely from RDBs as R2RML. A mapping from data model to RDF can be expressed with RML but RML processors may not support all kinds of formats.

Two components of the R2RML syntax are specific to RDBs. First, there is the way the data source is expressed by defining a logical table. Secondly, the data is accessed via the logical table's column names. RML generalizes these components with more generic concepts:

- `rml:logicalSource`: A predicate indicating a logical source which is a subset of a source of data is necessary to generate the desired triples.
- `rml:reference`: A predicate indicating which pieces of information are referred to.

Figure [3] exemplifies a mapping of an XML file to RDF with RML. The logical source depicted can be understood as a table of XML elements resulting from the XPath expression `/group/person` in the data.xml file. XPath is also the formulation of the references to pieces of information of these XML elements. For instance, consider lines 2-5 of the data source representing an XML element, the string `"Davan"` is referred to as the XPath `firstname` from this element. Line 22 of the mapping expresses this reference.

```xml
<group>
  <person id="1">
    <firstname>Davan</firstname>
    <lastname>Chiem Dao</lastname>
  </person>
  <person id="2">
    <firstname>John</firstname>
    <lastname>Smith</lastname>
  </person>
</group>
```

```
@prefix rr: <http://www.w3.org/ns/r2rml#>.
@prefix rml: <http://semweb.mmlab.be/ns/rml#>.
@prefix ql: <http://semweb.mmlab.be/ns/ql#>.
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@base <http://www.example.org/>.

<#PersonTriplesMap>
    a rr:TriplesMap;
    rml:logicalSource [
        rml:source "data.xml" ;
        rml:iterator "/group/person";
        rml:referenceFormulation ql:XPath;
    ];

    rr:subjectMap [
        rr:template "http://www.example.org/Person{@id}" ;
        rr:class foaf:Person ;
    ] ;

    rr:predicateObjectMap [
        rr:predicate foaf:firstname ;
        rr:objectMap [ rml:reference "firstname" ] ;
    ] ;

    rr:predicateObjectMap [
        rr:predicate foaf:surname ;
        rr:objectMap [ rml:reference "lastname" ] ;
    ] ;
.
```

```
@prefix ex: <http://www.example.org/>.
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

ex:Person1 rdf:type foaf:Person.
ex:Person1 foaf:firstname "Davan".
ex:Person1 foaf:surname "Chiem Dao".
ex:Person2 rdf:type foaf:Person.
ex:Person2 foaf:firstname "John".
ex:Person2 foaf:surname "Smith".
```

Figure 3: RML mapping example, XML data (left), RML mapping (middle), resulting RDF graph (right)

2.7 SPARQL

The SPARQL Protocol and RDF Query Language (SPARQL) is a W3C Recommendation used express queries to RDF data [12]. The queried data can be queried across multiple data sources either natively or virtually stored as RDF. The results of these queries are either data sets or RDF graphs. In the following, an overview of the SPARQL syntax, inspired by Feignbaum and Prud'hommeaux's tutorial [13], will be presented.

The general structure of a SPARQL query is shown by Listing [1]. It consists of five parts: prefix declarations, a result clause, a dataset definition, a query pattern, and query modifiers.

```
# prefix declarations (optional)
PREFIX foo: <...>
PREFIX bar: <...>
# result clause
SELECT ...
# dataset definition (optional)
FROM ...
FROM NAMED ...
# query pattern
WHERE {
...
}
# query modifiers (optional)
GROUP BY ...
HAVING ...
ORDER BY ...
LIMIT ...
OFFSET ...
BINDINGS ...
```

Listing 1: SPARQL query structure

- **Prefix declarations** are used to abbreviate IRIS resulting in more condensed queries for the user's convenience.
- The **result clause** determines what is returned and is one of the keywords `SELECT`, `ASK`, `DESCRIBE`, or `CONSTRUCT`:
  - `SELECT` queries return "selected" variables in a table format of data matching the query pattern.
  - `ASK` queries checks if there is at least a result matching the query pattern and returns a Boolean value.
  - `DESCRIBE` queries return an RDF graph that describes all the resources matching the query pattern.
  - `CONSTRUCT` queries return an RDF graph from a template specified in the result `CONSTRUCT` clause by using the data matching the query pattern.
- The **dataset definition** determines the RDF dataset that is queried.

- The **query pattern** determines the graph pattern that needs to be matched. In essence, a graph pattern is a set of triple patterns. Multiple graph patterns can be combined with some patterns being optional, others being united, negated, or filtered out and subqueries are also possible.
- **Query modifiers** determine how to rearrange the query result. For example, aggregation of values is achieved with the `GROUP BY` keyword or sorted in a particular manner with the `ORDER BY` keyword.

The previous description of the SPARQL syntax focused on querying data from RDF data sources. Another part of the syntax can be used to update an RDF store. Actions such as insertion, deletion of triples, and creation or deletion of named graphs are expressed with this syntax.

2.8 SHACL

The Shapes Constraint Language (SHACL) is a W3C standardized language to express conditions that RDF graphs should respect [14]. These conditions are also referred to as shapes because they impose that parts of an RDF graph must have a certain shape. This language is an RDF-based language thus the shapes form an RDF graph called a "shapes graph". On the other hand, the RDF graph that is validated against the shapes graphs is called a "data graph".

The validation process of a data graph against the shapes graph is illustrated in Figure [4]. For each shape, some nodes of the data graph are selected based on its target. These nodes are referred to as focus nodes. Then, a verification of these focus nodes against the shape's constraint is performed. When a node does not conform, a validation result will explain the cause of the error. If all nodes are valid, then the data graph is said to conform, and the validation report informs as such. Whereas if at least one node fails, the validation report states that the data graph does not conform and for each failing node the cause of the error is reported.

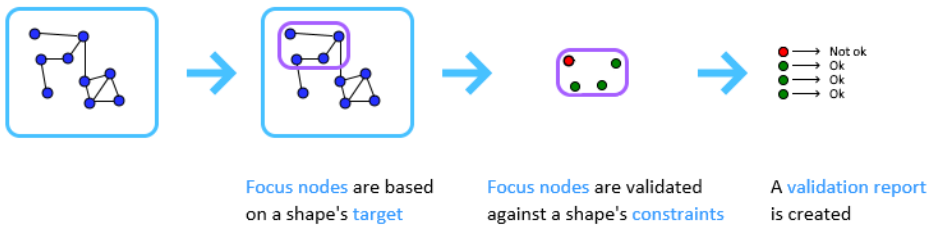


Figure 4: SHACL validation process steps

Figure [5] is an example of the validation with SHACL. The data graph contains information on two people, John, and me. The shapes graph states that every person must have a first name and a valid Social Security Number (SSN). In this case, the validation fails because John[1] does not have a first name and he has a wrong SSN.

[1]To be precise it is the resource describing John which does not have a first name

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix ex: <http://www.example.org/>.

ex:Davan a foaf:Person ;
    foaf:firstname "Davan" ;
    ex:SSN "0123.4567" ;
.

ex:John a foaf:Person ;
    ex:SSN "9876.4A" ;
.
```

```
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix ex: <http://www.example.org/>.

ex:PersonShape a sh:NodeShape ;
    # shape annotations
    sh:name "Person Shape" ;
    sh:description
        "Persons must have a first name and a SSN" ;

    # target definion
    sh:targetClass foaf:Person ;

    # constraints
    sh:property [
        sh:path foaf:firstname ;
        sh:minCount 1 ;
        sh:maxCount 1 ;
        sh:datatype xsd:string ;
        sh:nodeKind sh:Literal ;

    ] ;
    sh:property ex:SSNShape ;

    # validation annotations
    sh:message "Invalid person shape";
    sh:severity sh:Warning ;
.

ex:SSNShape a sh:PropertyShape ;

    # constraints
    sh:path ex:SSN;
    sh:pattern "^(\\d|\\.)*$" ;
    sh:datatype xsd:string ;
    sh:minLength 9 ;
    sh:maxLength 9 ;

    # validation annotations
    sh:message "Invalid social security number";
    sh:severity sh:Warning ;
.
```

```
@prefix ex:     <http://www.example.org/> .
@prefix foaf:   <http://xmlns.com/foaf/0.1/> .
@prefix rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix sh:     <http://www.w3.org/ns/shacl#> .

[ rdf:type     sh:ValidationReport ;
  sh:conforms  false ;
  sh:result    [
    rdf:type                    sh:ValidationResult ;
    sh:focusNode                ex:John ;
    sh:resultMessage            "Property needs to have at least 1 value" ;
    sh:resultPath               foaf:firstname ;
    sh:resultSeverity           sh:Violation ;
    sh:sourceConstraintComponent  sh:MinCountConstraintComponent ;
    sh:sourceShape              []
  ] ;
  sh:result    [
    rdf:type                    sh:ValidationResult ;
    sh:focusNode                ex:John ;
    sh:resultMessage            "Invalid social security number";
    sh:resultPath               ex:SSN ;
    sh:resultSeverity           sh:Warning ;
    sh:sourceConstraintComponent  sh:PatternConstraintComponent ;
    sh:sourceShape              ex:SSNShape ;
    sh:value                    "9876.4A"
  ] ;
  sh:result    [
    rdf:type                    sh:ValidationResult ;
    sh:focusNode                ex:John ;
    sh:resultMessage            "Invalid social security number";
    sh:resultPath               ex:SSN ;
    sh:resultSeverity           sh:Warning ;
    sh:sourceConstraintComponent  sh:MinLengthConstraintComponent ;
    sh:sourceShape              ex:SSNShape ;
    sh:value                    "9876.4A"
  ]
] .
```

Figure 5: SHACL validation example, data graph (left), shapes graph (middle), validation report (right)

These errors are reported in the validation report.

Moreover, the shapes graph in this figure illustrates how shapes are described with SHACL. The description of shapes can be divided into four parts: the shape annotations, the target definition, the constraints, and the validation annotations.

The shape annotations are optional properties that may be used as metadata of a shape. These properties are not part of the validating properties and are ignored by SHACL processors. Lines 7-10 of the shapes graphs show metadata naming and describing the shape. Other metadata, such as group and order, may be employed to print RDF files in a predictable manner.

The target definition determines the target of the shape, which are the focus nodes that must respect the shape's constraints. This target can be a specific node, nodes of a specific class (as in the example), subjects, or objects of a specific predicate.

The constraints are the rules that each focus node must respect. Many types of constraints can be expressed such as value type constraints, cardinality constraints, and value range constraints, among others. Lines 16-23 can be translated into natural language as "*focus nodes must have exactly value node reached through the* `foaf:firstname` *predicate. This value node must be a string literal*". This rule is a shape-based constraint which indicates that value nodes must respect the given shape. This illustrates how shapes can describe nodes linked to the focus node that themselves can also describe other nodes linked. As a result, a shape can describe a whole graph or sub-graph.

The validation annotations modify the reported result in case of an erroneous validation for a focus node. For instance, one can mention the severity of a shape or modify the message reported. In the example, the results of the errors in the SSN, both have a `sh:Warning` severity and the custom `sh:resultMessage`.

The previous explanation constituted the core part of SHACL also referred to as SHACL-CORE. A second part of the specification called SHACL-SPARQL provides mechanisms to create custom constraints or constraint components using SPARQL queries.

Listing [2] is an example of a SPARQL-based constraint which ensures that the values reached from the focus node following a particular path are all different. One can observe that a SPARQL-based constraint is defined by a SPARQL query following a specific syntax. Indeed, only a subset of SPARQL may be used in this context. For instance, some variables are pre-bound such as `$this` variable being bounded to a focus node. Furthermore, the `MINUS`, `SERVICE`, and `VALUES` keywords are prohibited, but more importantly only `ASK` (only for constraint components) and `SELECT` queries are allowed.

```
ex:valueIsUnique a sh:SPARQLConstraint;
    sh:prefixes <> ;
    sh:select """
        SELECT $this
        WHERE {
            {
                SELECT $this (COUNT(?v) as ?vOcc)
                WHERE {
                    $this $PATH ?v.
                }
                GROUP BY ?v $this
            }
        FILTER(?vOcc > 1)
    }""" ;
.
```

Listing 2: SPARQL-based constraint example

A SHACL engine uses the result of these queries to determine if the focus node is valid depending on whether the result set (which is tabular) returned by the `SELECT` query is empty or not. If the result set is not empty a validation result reports an error for each row.

Constraints using SPARQL-based constraint components work similarly to SPARQL based constraints. These components work as template SPARQL queries which allow the use of parameters. Note that in a case of an `ASK` query the shape is valid if the return value is true meaning that the query pattern was matched as opposed to a `SELECT` query being invalid if the query pattern matched.

3 Social Security Background

The previous chapter presented the technical background necessary to comprehend this thesis. In this chapter, we present the domain in which we conducted our study. This chapter will present the Belgian social security, its role, and some insight into how they operate. This chapter is mostly based on the brochure "Everything you have always wanted to know about social security" [15].

3.1 Social Security and its roles

The Belgian social security is the set of all provisions aiming at guaranteeing the financial autonomy of citizens whenever they are exposed to social risks. Depending on the risk to which one is exposed, the social security will financially support them accordingly. Specifically, the three functions of the social security are:

- Provide a replacement income in the event of loss of employment income such as unemployment, retirement, or work incapacity.
- Provide an income supplement for additional social burdens such as raising children or medical expenses.
- Provide a living wage to those deprived of professional income.

The social security is divided into seven branches to distribute social rights to achieve one or several of the functions above. These seven branches are:

- Old-age pension and survivors' pension
- Unemployment
- Insurance for accidents at work
- Insurance for occupational diseases
- Family allowance
- Illness and disability insurance
- Annual holiday

The funding of these social rights is done by collecting contributions from active citizens proportionally to their wages. Hence, the role of the social security is to redistribute collected through the different branches.

3.2 Organization

The organization of the Belgian social security is complex with many institutions working together. Figure [6] shows the whole network of institutions involved in the social security.

Figure 6: Social security institution network [15]

Most of these institutions correspond to one of the Belgium's social security branches. For instance, the Federal Pension Service (FPS) is responsible for pension, the National Employment Office is responsible for unemployment, and the National Office for Annual Leave (NOAL) for annual leave. Others are responsible for collecting contributions such as the National Office of Social Security Office (NOSS) for employee and employer contributions and the National Institute for Social Insurance of Self-Employed (NISSE) for self-employed people.

3.3 E-government

The electronic government (e-government) is the expansion of government services using information technology. Its goal in the social security domain is to reduce the number of filled-out forms, reduce the number of contacts between the social security and employers/employees, and reduce the time necessary to complete forms by limiting the number of fields.

To attain this goal, the social security developed three electronic declarations that employers send to the NOSS to communicate information about employees.

- The *Déclaration immédiate/onmiddelijke aangifte* (Dimona) is a declaration that contains information concerning the start and end of an employment relationship.
- The *Déclaration des Risques Sociaux* (DRS) declares that an employee encountered a social risk during the employment relationship. Social risks are events that change an employee's social position, such as being fired, being a victim of an accident at work, or suffering an illness for an extended period.
- The *Déclaration multifonctionnelle/ multifunctionele Aangifte* (DmfA) is a declaration that contains more general information about an employment relationship. That information is required by the multiple branches of the social security. We will describe this declaration in more detail in Section 3.4.

One could represent these declarations on a timeline on which the Dimona determines the edges, the DmfA describes the period between these edges, and the DRS describes some one-time events that may occur. (See Figure [7])

Figure 7: Timeline of social security declarations

3.4 DmfA

One of the goals of this thesis is to prove that KGs can facilitate and improve certain aspects of e-government. More specifically, this thesis focuses on data validation of declarations. The DmfA was selected as the subject of this study as it is the most complex electronic declaration. It is the one with the most fields with a variety of rules to respect. Thus, it could reveal to what extent KGs and their technologies can potentially improve processes supported by the existing information system. This section goes into the details of the DmfA and how it is currently validated.

First, the DmfA is, as its French-Dutch name suggests, a multifunctional declaration that employers send to the NOSS. It is multifunctional because all institutions use it within social security for their function, such as determining the amount of contribution a company owes and allocating social rights and indemnity payments. Thus, knowledge graphs are an interesting way of representing and storing DmfA information because the information from heterogenous sources (in this case, the different institutions) can be integrated into these graphs.

Concerning the kind of information transmitted, it includes salary data and the employee's working time. It can either be sent via the Web by manually filling in an online form or via file transfer, for which the information is contained in an XML file. The Web interface covers many data validation aspects, though filling those in manually can be tedious for companies employing multiple people. XML is used for communicating information about multiple employing in batch. The Belgian social security has made available the XSD schema and a simple Java application to do some "superficial" data validation, though we will explain how this is limited. This latter format for a DmfA will be the input source of the validation process. This thesis "assumes" no Web information system exists and starts from the XML file.

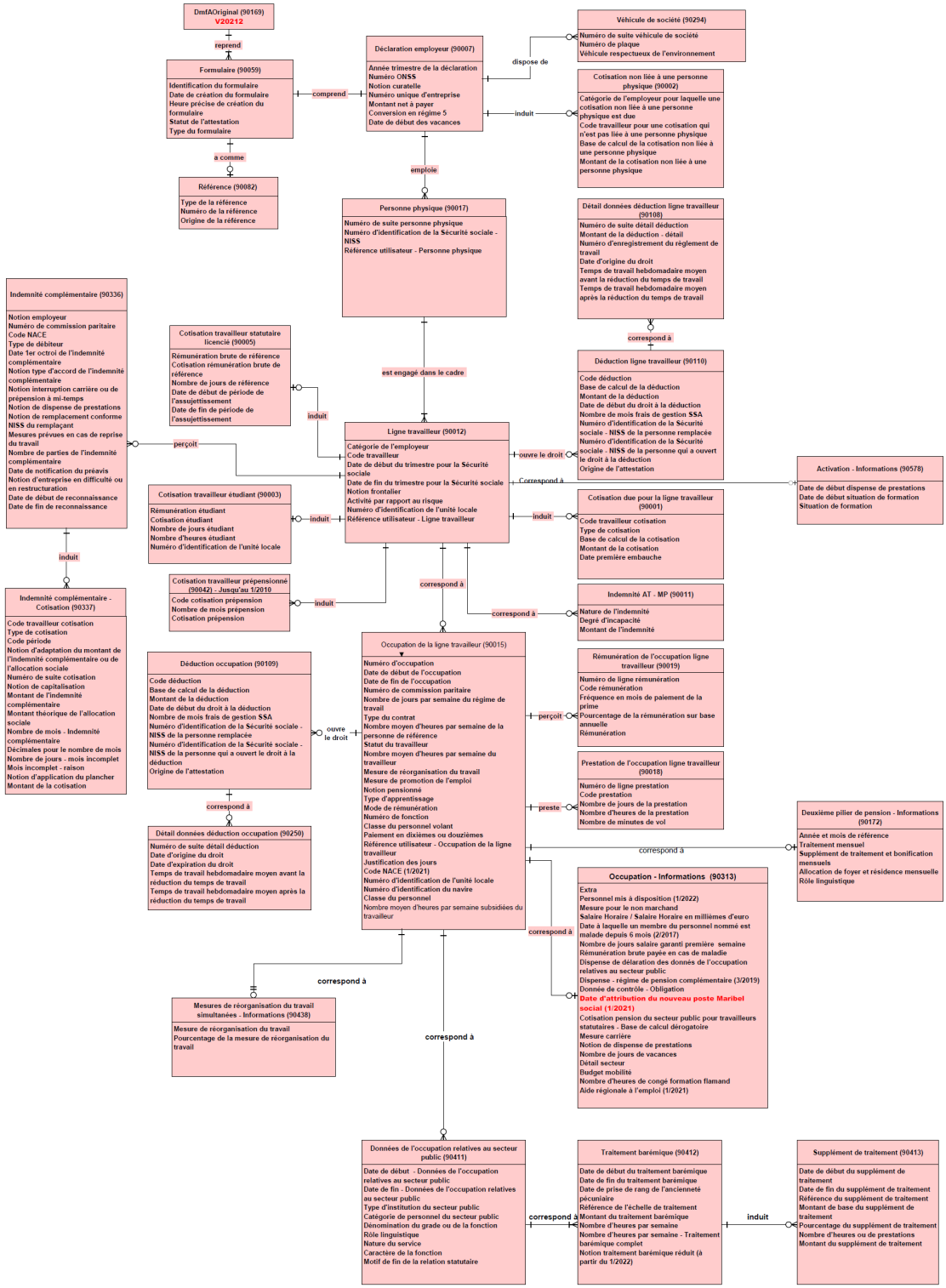


Figure 8: DmfA 2022/3 ERD [16]

A glossary provided by the NOSS details the content of the XML file [17]. It gives information on the various entities, relationships, and attributes of the DmfA.

The entity-relationship diagram (ERD) of the DmfA in Figure [8] illustrates these entities, relations, and attributes. In this ERD using Crow's Foot Notation, one can see 29 entities (also called functional blocks) and 28 relationships between entities. These relationships are always one to (optionally) one or (optionally) many. This indicates a hierarchy between the entities. Thus, this diagram is a tree that suits well the XML format. One can even argue that the ERD was designed with an XML in mind. Whereas all the entities are represented on the diagram even if they are currently forbidden, only the allowed attributes (also called zones) are represented. Nevertheless, a total of 215 attributes are described in the glossary.

The fact that some entities, relationships, or attributes can become prohibited highlights that the DmfA is changing over time. Indeed, as the regulations change, the DmfA must reflect these changes. Hence, there are different versions, each corresponding to a particular quarter. When an employer sends a DmfA for a particular quarter it must respect the rules of that quarter. This temporal aspect should be taken into consideration when designing a validation process.

To show how rules are described in the glossary, an example is provided in Figure [9]. It illustrates the glossary's HTML version, but PDF and XML versions also exist. One can observe that some rules are structured, such as the compulsory presence of an attribute or the maximum length of a value, which both can be expressed in XSD. Nevertheless, most rules are expressed in natural language, such as respecting a checksum. While some of these rules could also be expressed in XSD, such as a regular expression for 0 or 10 digits, many of them cannot and require to be validated by an application (e.g., computing the checksum).

Currently, there are two validation processes put in place by the NOSS. Both processes will be briefly presented in the following.

The simplest validation process consists of a lightweight Java program. This program is made available to employers to verify a DmfA before submitting it to the NOSS. This program verifies that the input DmfA file is a well-formed XML and its conformance against the XSD of the DmfA. Hence, only the rules expressed and present in the XSD are verified. Additionally, only two rules outside the expressivity of XSD are verified: the unicity of the Identification Number of the Social Security and the amount owned declared corresponds to the one computed. The details of these rules are not important at this point. However, it highlights that the validation of rules beyond XSD's expressivity requires software. Moreover, these complex rules are written in Java which, unlike XSD, is not interoperable.

The other validation process occurs upon the submission of a DmfA. The NOSS runs a non-disclosed program that validates all the rules a DmfA should respect. However, the validation result can take up to ten days to be sent. This can slow down the formation of a valid DmfA, as corrections could only be possible after this variable delay.

Figure 9: DmfA glossary example. This page describes the enterprise-number block. It not only contains a definition of that concept but also information about its permitted values — both structured and unstructured. Some rules about the permitted values cannot be captured in XSD. [17]

These explanations have highlighted some key issues with the current situation. Many rules fall outside the expressivity of XSD and are described in natural language and thus cannot be processed by a computer agent or are written in a non-interoperable format (i.e., Java code or non-disclosed). The current validation processes are either partially complete or not directly available for employers, increasing the required time to fill the declaration. Hopefully, knowledge graph technologies can overcome these problems by creating a set of interoperable data validation rules.

4 Related Work

In public administrations there have been some use-cases where SHACL was implemented to express constraints to which data must comply. In the European Union (EU), these constraints are used as part of an application profile. An application profile is the description of how to apply a standard to a specific application [18]. In other words, it is the specification of an application which is based on a standard. Thus, it may specify how the data should be, its constraints.

The European Data Portal is a key example of such an application. It provides 176 catalogs for a total of 1,601,899 datasets [19]. To facilitate the interchange of metadata describing these catalogues and datasets, the DCAT Application Profile for Data Portals in Europe (DCAT-AP) was developed as a shared data model [20]. This model is detailed in various formats including a natural language PDF, a UML diagram, and SHACL shapes [21]. The EU also provides an online validator for DCAT-AP, enabling the metadata, represented as an RDF graph, to be validated against these SHACL shapes.

At a local level, the Flemish Government in Belgium developed the Open Standaarden voor Linkende Organisaties (OSLO) specification [22]. It is a collection of specifications meant to exchange data between organizations. Among those specifications are some application profiles that specify constraints on data concerning, for example: addresses, persons, roads, buildings, public decision, etc. Most of these application profiles are expressed in SHACL among other formats. Similarly to the EU, an online SHACL validator is available for most of the application profiles.

The SHACL shapes in these application profiles describe the "simple" integrity constraints such as cardinality, datatype, and node type. There are also a few instances of 'or' operators used to combine shapes into a more complex shape. This shows that SHACL can be used to validate simple data models. In this work, we try to express more complex shapes by using more of the SHACL-core vocabulary and even more complex shapes with SHACL-SPARQL vocabulary.

5 Approach

In this chapter, we delve into the practical aspect of our research by presenting the approach taken to contextualize and validate DmfA declarations with graph technologies, specifically RML and SHACL. The various steps involved in the process can be seen in Figure [10]. We will outline the engineering process followed for each step of the validation process.

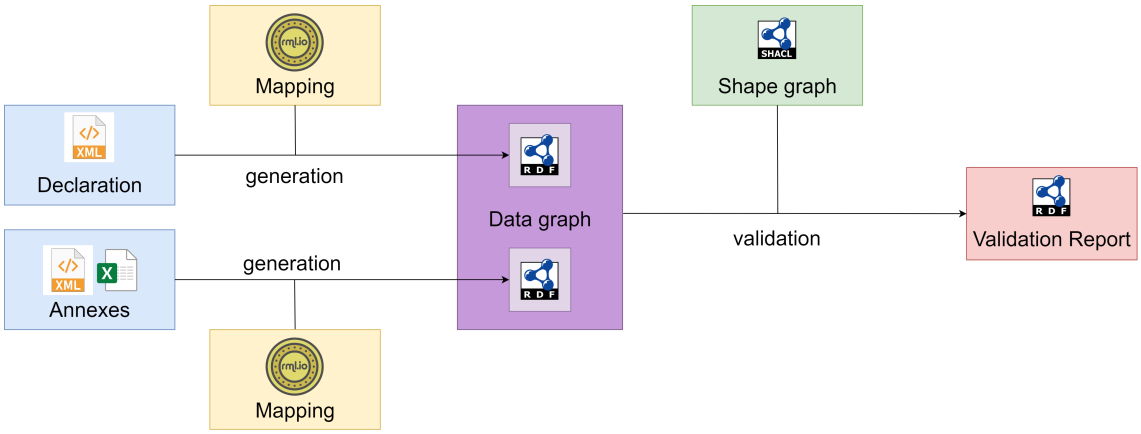


Figure 10: DmfA validation with KG technologies overview

This experiment along its sub-experiments attempts to answer the following questions.

- Is it possible to validate social security declarations with knowledge graph technologies?
- How difficult is it to develop a validation process with these technologies?
- What are the pros and cons of this method compared to existing validation process?
- How computationally intensive is this validation process?
- Is the validation process maintainable?

All the scripts and documents used for this experiment can be found at the following GitHub repository: `https://github.com/Ikeragnell/dmfa`.

5.1 DmfA Vocabulary

If the sole purpose of this experiment was to validate a DmfA, using a direct mapping of the DmfA XML to an RDF graph validated with SHACL would have been appropriate. However, as this experiment tries to go beyond this goal and reach interoperability between the social security institutions, a vocabulary was generated. In this section, the reason why and how the DmfA vocabulary was generated, as well as its current limitations will be presented.

5.1.1 Reasons to build a vocabulary

A vocabulary is a light-weight ontology, with only a few axioms such as class hierarchies, domains, and ranges. We decided to construct a DmfA vocabulary as we deemed it more suited for our needs compared to a fully-fledged ontology. This decision was driven by several key reasons.

Primarily, our project did not necessitate support for complex reasoning tasks that would typically call for a highly-axiomatized ontology. Instead, our primary need was the semantic annotation of data contained in a DmfA, and infer some additional information via the class and property hierarchies, for example. This would also bring semantic interpretability between different information systems at stake, being the several institutions of the NOSS and the companies making the declarations.

In addition, managing integrity constraints in an ontology with OWL presents complexity and potential difficulties that we wanted to avoid. OWL's open world assumption is an important challenge. For example, it necessitates declaring that entities are explicitly distinct.

Finally, evaluating SHACL capabilities is one of the main goals of this thesis. Thus, some reasoning tasks that could be achieved with an ontology such as cardinality constraints were omitted to not overlap with SHACL rules developed (cf. Section 5.4).

5.1.2 Generating the DmfA vocabulary

The actual generation of the vocabulary can be regarded as lifting the ERD to the vocabulary. The lifting process mainly consisted of mapping entities, relations, and attributes to their respective classes, object properties, and data properties. To do so, the XML documentation of the entities and the attributes issued from the glossary were processed by a Python script. Figures [11] and [12] show how a class and a data property were generated from these XML files.

As there was no computerized documentation of the relationships, a CSV was created with the domain, range, and their corresponding identification code from the XML documentation of entities. The file had to be additionally annotated with the French label of these relationships before generating them with a Python script (See

Figure 13). English translation of the labels could have been used for the IRI of the relationships (e.g., `#employs`, `#dispose_of`). However, many relationships shared the same label and would have resulted in relationships being synonymous while they are not. In addition, the development of rules would be more complex because the domain and range of these synonymous relationships would be a union of classes. Nevertheless, rules depend on a specific pair of subject and object classes and thus we would need to distinguish this specific pair from the union of classes. To make the distinction possible, the IRIs were built as a combination of the identification code of the domain and range. Note that another solution could have been to refine the vocabulary with domain experts by precisely defining and distinguishing the relationships.

```
From,To,Domain,Range,LabelFr
90007,90017,EmployerDeclaration,NaturalPerson,emploie
90007,90294,EmployerDeclaration,CompanyVehicle,dispose de
```

```
# ObjectProperties - Relation
# CSV file should have been generated with DmfA_Relation_Base.py and annotated
relationCSV = open("Dmfa_Relation.csv", encoding="utf-8")
relationTable = csv.reader(relationCSV, delimiter=",") # From,To,Domain,Range,LabelFr
next(relationTable, None)

for row in relationTable:

    relationName = "R_" + row[0] + "_" + row[1]
    labelFr = row[4]
    domain = row[2]
    range = row[3]

    outfile.write("""
    :{} a owl:ObjectProperty;
        rdfs:label "{}"@fr;
        rdfs:domain :{};
        rdfs:range :{};
    .
    """.format(relationName, labelFr, domain, range))

relationCSV.close()
```

```
:R_90007_90017 a owl:ObjectProperty;
    rdfs:label "emploie"@fr;
    rdfs:domain :EmployerDeclaration;
    rdfs:range :NaturalPerson;
.

:R_90007_90294 a owl:ObjectProperty;
    rdfs:label "dispose de"@fr;
    rdfs:domain :EmployerDeclaration;
    rdfs:range :CompanyVehicle;
.
```

Figure 13: Object properties generation, CSV of relationships (top), Python script (middle), generated object properties (bottom)

```xml
    <FunctionalBlock>
        <Code>90007</Code>
        <Version>2022/3</Version>
        <DatePublication>30/08/2022</DatePublication>
        <Nom>Déclaration employeur</Nom>
        <Zones>00013 - ANNÉE - TRIMESTRE</Zones>
        <Zones>00011 - NUMÉRO ONSS</Zones>
        <Zones>00012 - NOTION CURATELLE</Zones>
        <Zones>00014 - NUMÉRO D'ENTREPRISE</Zones>
        <Zones>00015 - MONTANT NET À PAYER</Zones>
        <Zones>00016 - CONVERSION EN RÉGIME 5</Zones>
        <Zones>00017 - DATE DE DÉBUT DES VACANCES</Zones>
        <XmlLabel>EmployerDeclaration</XmlLabel>
        <Description>Bloc fonctionnel permettant de déclarer les données de la déclaration.</Description>
        <CardinaliteMinimum>1</CardinaliteMinimum>
        <CardinaliteMaximum>1</CardinaliteMaximum>
        <NrBlocLie>90017</NrBlocLie>
        <NrBlocLie>90294</NrBlocLie>
        <NrBlocLie>90002</NrBlocLie>
        <BlocLie>90017 - Personne physique</BlocLie>
        <BlocLie>90294 - Véhicule de société</BlocLie>
        <BlocLie>90002 - Cotisation non liée à une personne physique</BlocLie>
        <Presence>Indispensable</Presence>
        <LimitationsSupplementaires></LimitationsSupplementaires>
        ...
    </FunctionalBlock>
```

```python
# Classes - Entities
rootBFxml = xmlRoot('VersCplt_BFdmfa20223FR.xml', encoding="iso-8859-1")
for bf in rootBFxml:

    className = bf.find("XmlLabel").text
    identifier = bf.find("Code").text
    labelEn = re.sub(r"([A-Z]+)([A-Z][a-z])|([a-z])([A-Z])", r"\1\3 \2\4", DPName)
    labelFr = bf.find("Nom").text
    commentFr = addTripleQuotes(bf.find("Description").text)
    presenceFr = addTripleQuotes(bf.find("Presence").text)

    strToWrite = ""
    strToWrite = appendIfNotNone(strToWrite, ':{} a owl:Class;', className)
    strToWrite = appendIfNotNone(strToWrite, 'dc:identifier "{}";', identifier)
    strToWrite = appendIfNotNone(strToWrite, 'rdfs:label "{}"@en;', labelEn)
    strToWrite = appendIfNotNone(strToWrite, 'rdfs:label "{}"@fr;', labelFr)
    strToWrite = appendIfNotNone(strToWrite, 'rdfs:comment {}@fr;', commentFr)
    strToWrite = appendIfNotNone(strToWrite, ':presence {}@fr;', presenceFr)

    outfile.write(strToWrite + "\n .")
```

```
:EmployerDeclaration a owl:Class;
    dc:identifier "90007";
    rdfs:label "Employer Declaration"@en;
    rdfs:label "Déclaration employeur"@fr;
    rdfs:comment """Bloc fonctionnel permettant de déclarer les données de la déclaration."""@fr;
    :presence """Indispensable"""@fr;
.
```

Figure 11: Class generation, XML glossary (top), Python script (middle), generated class (bottom)

```xml
11101     <glossary>
11102         <Code>00011</Code>
11103         <Version>2022/3</Version>
11104         <DatePublication>30/08/2022</DatePublication>
11105         <Title>NUMÉRO ONSS</Title>
11106         <LabelXml>NOSSRegistrationNbr</LabelXml>
11107         <BlocFonctionnel>Déclaration employeur</BlocFonctionnel>
11108         <Description>Chaque employeur, qu'il soit une personne physique, un groupement de
11109             personnes physiques ou une personne morale, qui occupe du personnel soumis à la loi
11110             du 27 juin 1969, doit être inscrit à l'ONSS. Il s'agit d'un numéro ONSS définitif.
11111         </Description>
11112         <DomaineDeDefinition>Nombre entier et élément de [100006;199999934] pour les numéros
11113             définitifs. Si le numéro unique d'entreprise est connu (zone 00014 différent de
11114             zéro), le numéro ONSS peut être mis à la valeur zéro. </DomaineDeDefinition>
11115         <Reference></Reference>
11116         <Type>Numérique</Type>
11117         <Longueur>9</Longueur>
11118         <Presence>Indispensable </Presence>
11119         <Format>0 ou NNNNNNNCC . NNNNNNN est le numéro · CC est le numéro de contrôle.</Format>
              ...
11170     </glossary>
```

```python
81  # DatatypeProperties - Attributes
82  rootZones = xmlRoot('VersCpltdmfa20223FR.xml', "iso-8859-1")
83  for zone in rootZones:
84
85      DPName = zone.find("LabelXml").text
86      identifier = zone.find("Code").text
87      labelEn = re.sub(r"([A-Z]+)([A-Z][a-z])|([a-z])([A-Z])", r"\1\3 \2\4", DPName)
88      labelFr = zone.find("Title").text
89      commentFr = addTripleQuotes(zone.find("Description").text)
90      presenceFr = addTripleQuotes(zone.find("Presence").text)
91      isDefinedByFr = addTripleQuotes(zone.find("DomaineDeDefinition").text)
92      domain = """
93          [ a owl:Class ;
94              owl:unionOf ({})
95          ]
96      """.format(" ".join(domainDict[identifier]))
97      length = zone.find("Longueur").text
98      typeFr = addTripleQuotes(zone.find("Type").text)
99
100     strToWrite = ""
101     strToWrite = appendIfNotNone(
102         strToWrite, ':{} a owl:DatatypeProperty;', DPName)
103     strToWrite = appendIfNotNone(strToWrite, 'dc:identifier "{}";', identifier)
104     strToWrite = appendIfNotNone(strToWrite, 'rdfs:label "{}"@en;', labelEn)
105     strToWrite = appendIfNotNone(strToWrite, 'rdfs:label "{}"@fr;', labelFr)
106     strToWrite = appendIfNotNone(strToWrite, 'rdfs:comment {}@fr;', commentFr)
107     strToWrite = appendIfNotNone(strToWrite, ':presence {}@fr;', presenceFr)
108     strToWrite = appendIfNotNone(strToWrite, 'rdfs:domain {};', domain)
109     strToWrite = appendIfNotNone(
110         strToWrite, 'rdfs:isDefinedBy {}@fr;', isDefinedByFr)
111     strToWrite = appendIfNotNone(strToWrite, ':length {};', length)
112     strToWrite = appendIfNotNone(strToWrite, ':type {}@fr;', typeFr)
113
114     outfile.write(strToWrite + "\n.\n")
```

```turtle
3542  :NOSSRegistrationNbr a owl:DatatypeProperty;
3543      dc:identifier "00011";
3544      rdfs:label "NOSS Registration Nbr"@en;
3545      rdfs:label "NUMÉRO ONSS"@fr;
3546      rdfs:comment """Chaque employeur, qu'il soit une personne physique, un groupement de personnes ...
3547      """@fr;
3548      :presence """Indispensable """@fr;
3549      rdfs:domain
3550          [ a owl:Class ;
3551              owl:unionOf (:EmployerDeclaration)
3552          ]
3553      ;
3554      rdfs:isDefinedBy """Nombre entier et élément de [100006;199999934] pour les numéros définitifs.
3555      Si le numéro unique d'entreprise est connu (zone 00014 différent de zéro), le numéro ONSS peut ...
3556      """@fr;
3557      :length 9;
3558      :type """Numérique"""@fr;
3559  .
```

Figure 12: Data property generation, XML glossary (top), Python script (middle), generated data property (bottom)

5.2 Mapping DmfA XML to RDF

In this section, we discuss how DmfAs as XML can be transformed to RDF through RML mappings. We explain how the mappings were generated, how we developed an IRI strategy and investigate storage strategies.

5.2.1 Generating mappings

5.2.1.1 Generating mappings for a particular DmfA XML

In Section 2.6, we presented RML which is a language to define mappings from any semi-structure data. To generate the mappings, we consider the example of a DmfA XML provided by the NOSS in their technical library as the input source. Although this file respects the DmfA XSD, it is not a valid declaration which will be shown in Section 5.5.1.

As the vocabulary is on the ERD of the XML's structure, the generation of mappings can be bootstrapped. A triples map instance was created for each class of the vocabulary by a Python script. Figure [14] shows an example of the mapping generated for a class. Each triples map defines a logical source based on the XML file to transform and the XPath of the element corresponding to the class. The subject map is also defined and determines how the subject is generated. In this case, the IRI is a Uniform Resource Name (URN) with the class name in it. More details will be explained in Section 5.2.2.

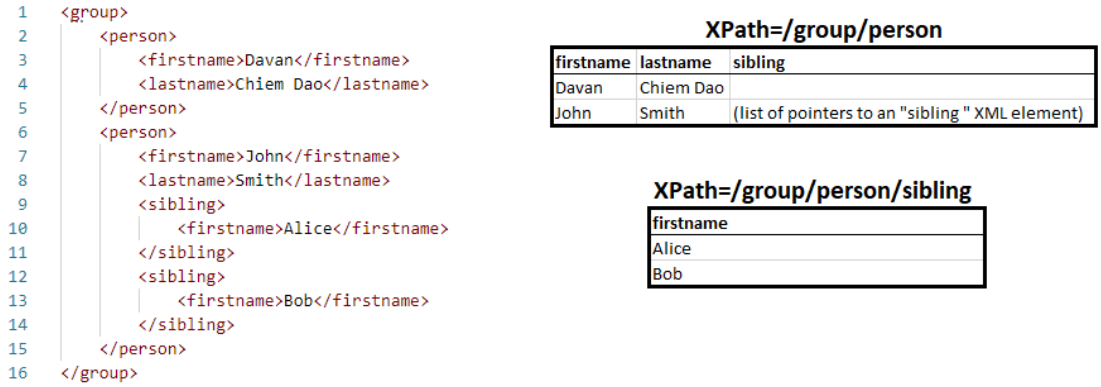
The mappings of data properties were straightforward to generate with the script because these properties have a corresponding element in the XML. In addition, the datatypes are determined using the ones in the XSD.

While generating triples maps and mapping rules for subjects and data properties were straightforward, the mapping rules of the object properties were more complex. In fact, it is currently impossible to correctly generate them because there is a loss of information with the way the data is represented[2]. As a reminder, the logical source from an XML can be understood as a table of XML elements resulting from the XPath expression. These logical tables do not contain information on the actual structure of the XML. As an example, Figure [15] shows and XML file and two logical tables based on XPath. One can observe that there are no columns matching which would indicate the relationship between John and his siblings. In short, these tables lack primary and foreign keys.

To solve this issue, some preprocessing was done to add these keys. An id and parent attribute, which corresponds to a primary and a foreign key, was added to each XML attribute. The mappings of the object properties (Figure [15]) could then be generated automatically. The object map references the result of the subject map

[2]The RML community is aware of this issue and are currently investigating potential solutions.

Figure 14: DmfA XML to RDF mapping example



firstname	lastname	sibling
Davan	Chiem Dao	
John	Smith	(list of pointers to an "sibling " XML element)

firstname
Alice
Bob

Figure 15: XML data (left) and table view of XPaths (right)

of another triples map and the join condition determines the resources that follow this predicate (the object property).

Having the mappings of the DmfA XML example fully generated, its RDF graph can be produced with any RML processor. For this thesis, the RMLMapper [23] program was chosen as processor. It is the reference implementation of RML written in Java but is only suitable for small datasets. Hence, the RDF generation may not scale properly.

5.2.1.2 Generating mappings for any DmfA XML

In the previous section, the mappings for a particular DmfA were presented. However, in practice, both companies and the social security manipulate multiple DmfAs. Thus, the specific mappings must be generalized.

There are two straightforward solutions, both with their downsides. First, the DmfA

XML files that need to be transformed could be renamed into the file name of the previous mappings. However, this would make the distinction between the files cumbersome. The other solution is to modify the Python script such that it takes as input the DmfA file name. Not only would the whole mappings be generated for each declaration, but it is also not interoperable as it requires specific technologies (in this case, Python).

While more complex, our proposed solution is declarative and relies uniquely on KG technologies. The previously generated mappings are stored in a quad store in their own named graphs. To generate the mappings for a specific file, a `CONSTRUCT` query is sent to the quad store (See Listing [3]).

```
PREFIX rml: <http://semweb.mmlab.be/ns/rml#>
CONSTRUCT {
  ?s ?p ?o.
  ?LogicalSource rml:source "filename"
}
FROM <http://kg.socialsecurity.be/mappings/dmfaxml/>
WHERE {
  {
    ?s ?p ?o.
    FILTER (?p != rml:source)
  }
  UNION
  {
    ?LogicalSource rml:source ?source.
  }
}
```

Listing 3: CONTRUCT query for DmfA to RDF mappings

In this query, a SPARQL endpoint is instructed to look for triples of DmfA mappings which are stored in `<http://kg.socialsecurity.be/mappings/dmfa/>`, a named graph. This query contains two parts. In the `WHERE clause`, the first part fetches all the triples that do not have `rml:source` as the predicate. In the `CONSTRUCT` clause, the second part creates for the subjects of `rml:source` triples with the same predicate but the desired file name. The filename is filled in by the script.

5.2.1.3 Prototype Deployment

As a proof of concept, a Jupyter Notebook was developed as prototype of user interface for the generation of RDF from DmfAs. The program takes as input the DmfA file name, RDF graph file name and the URL of the SPARQL endpoint where the reference mappings are stored. The SPARQL server storing the mappings is an Apache Jena Fuseki [24] server.

The program sends the query in Listing [3] with the input file name to the SPARQL endpoint. Then, it launches RMLMapper with the response and stores the result with the desired file name.

5.2.2 IRI strategy

Initially, blank nodes identifiers were considered for resources in the DmfA RDF. The reason being that there was no obvious way to identify a resource from its corresponding XML element apart from using the whole subtree. The combination of the class and the occurrence number could not be used to identify resources either, as two XMLs with permutations of their elements would be transformed into two RDF graphs with resources having inconsistent names. Thus, using blank nodes seemed more appropriate as resources cannot be identified.

Nevertheless, blank nodes identifiers had a more impactful downside because they represent different resources across systems and even RDF graphs. In fact, in RDF, the same blank node identifier in different name graphs of the same triples store does not refer to the same entity. This is by design when RDF was standardized; a blank node merely declares the existence of something. Hence, when validating a DmfA RDF against shapes with a SHACL processor, erroneous resources in the validation report would not have the same blank node identifiers. This would make the correction process more tedious because the source of the error cannot be identified.

To address this issue, we adopted a naming strategy inspired by Uniform Resource Names (URNs) resources as they follow patterns to identify resources. The structure of a URN is `urn:{nid}:{nss}` where `{nid}` is a namespace identifier registered by the Internet Assigned Numbers Authority (IANA) and `{nss}` is a namespace-specific string. This format can be used internally to the social security without registering the namespace to the IANA. In this case, the `{nid}` is `ss` which stands for social security.

The proposed IRI strategy and resulting IRIs are named by `urn:ss:{ref}-{class}{id}` (e.g., `urn:ss:012abcDEF-Form0`) where ref is the reference of the declaration class, `{class}` is the class of the resource and `{id}` is an identifier. These three components form the `{nss}` component of the URN and allow us to differentiate the data within a declaration (`{class}` and `{id}`) and between declarations (`{ref}`).

The reference is a 20-character long alphanumerical string that uniquely identifies a declaration. From the perspective of the social security, the reference should be known before creating the RDF graph and used in the subject map template to make the distinction between stored declarations possible. A variation of the query in Listing [3] could be used to retrieve the mappings to create such a graph (See Annex A). From the perspective of companies who are testing the validity of a declaration without knowing its reference should use the special reference `NOREF`.

Note that this naming scheme uses the occurrence number and thus problems with inconsistent naming in case of XML permutations still exist. This should be clearly documented and taken into consideration when querying the knowledge base.

5.2.3 Storing strategy

When strategizing on how to store the RDF declarations, one should bear in mind how the graph is going to be queried. For instance, some use-cases are computing the contributions due for the NOSS, computing the number of paid leave for the NOAL, and computing pension rights for the FPS. These usages have different requirements and thus different storing strategies must be considered.

To compute the contributions due for a quarter the NOSS needs the whole declaration. Storing each declaration in a named graph makes their retrieval faster than storing all of them in the default graph. To confirm this claim, a small benchmark was conducted, and the results are presented in Figure [16]. The details of this benchmark can be found in Annex B.2. One can observe that querying a storage with the default graph strategy is nearly always slower than querying one with the named graph strategy. The scale of this discrepancy is minor since response times are in the order of milliseconds, but we hypothesize that it may be more significant in larger datasets.

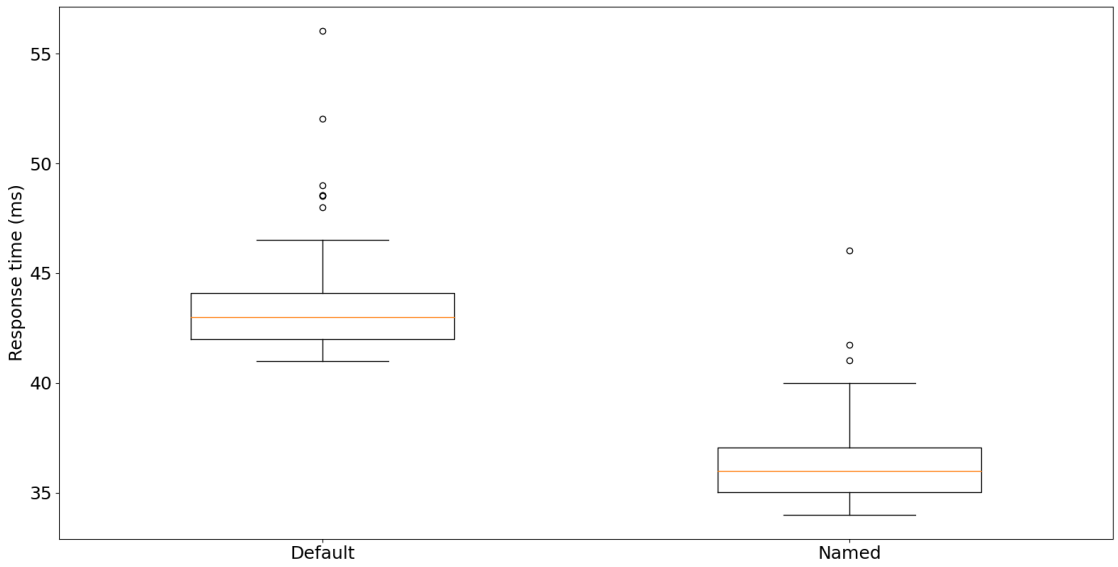


Figure 16: Storing strategy benchmark

Other institutions do not require the whole declaration. Hypothetically, the NOAL may only require information on the employees' working days and the FPS may only require salary data. Developing multiple triple-stores which only store relevant information for each institution may speed up data access, as it allows the institutions to retrieve only the data that they need, rather than having to access and filter through the entire dataset. This approach can be seen as a form of caching, as it stores the relevant data in a more easily accessible location in order to speed up future access to the same data.

5.3 Integrating DmfA Annexes

Some rules concerning the DmfA are based on information contained in annexes. Hence, these annexes must be integrated to be used in the validation process. This integration is similar to the transformation of DmfA XML to RDF. In this case, annexes as XML are transformed into RDF graphs. In this section, we will provide a general overview of the integration of the annexes and then delve into the specifics of certain aspects.

5.3.1 Integration of Annexes Overview

The integration of the annexes followed a process similar to transforming DmfA XML to RDF. A small vocabulary was built for each annex, mappings were developed, and an RML processor generated the graph. As most annexes share a similar structure, we will exemplify the integration process with an annex titled *Annexe numéro 7: Codification des rémunérations*.

All annexes are available in five formats: PDF, DOCX, XLS, CSV, and XML. Whereas RML mappings can be written for XML and CSV files, the XML format was chosen because it is the same format as DmfA. Each of the annexes contains data on an entity and its attributes. Annex 7 describes remuneration codes with attributes such as a value, description, and validity period (See Figure [17]).

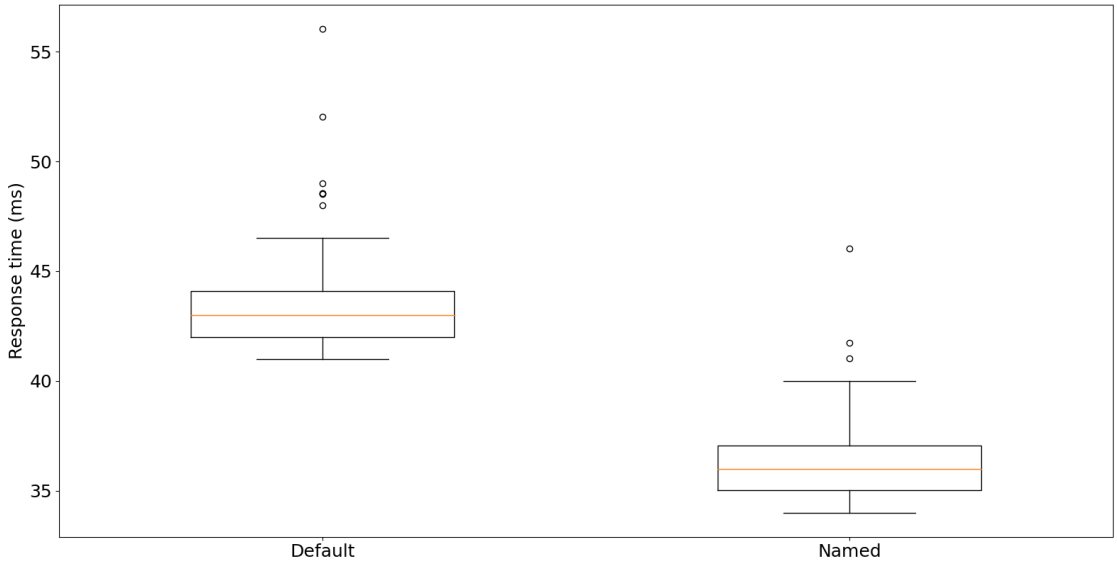


Figure 17: Annex 7: Codification of salaries XML [17]

The created vocabulary consists of a class and some data properties. It was generated manually to obtain IRIs for class and data properties that are readable and convey some of its semantics rather than using the XML label. Annexes describe

the validity period with dates, quarters, or both. Data properties for valid quarters were added if they were not initially present to obtain a consistent temporal representation. The whole vocabulary of Annex 7 can be seen in Figure [18]. For instance, it shows that a data property has the IRI `:validCodeDMFA`[3] instead of simply DMFA as in the XML.

```
:RemunCode a owl:Class ;
.

:Code a owl:DatatypeProperty ;
    rdfs:range :RemunCode;
.

:validCodeDMFA a owl:DatatypeProperty ;
    rdfs:range :RemunCode;
    rdfs:domain xsd:boolean;
.

:validCodeDRS a owl:DatatypeProperty ;
    rdfs:range :RemunCode;
    rdfs:domain xsd:boolean;
.

:validFrom a owl:DatatypeProperty ;
    rdfs:range :RemunCode;
.

:validTo a owl:DatatypeProperty ;
    rdfs:range :RemunCode;
.

:validFromQuarter a owl:DatatypeProperty ;
    rdfs:range :RemunCode;
.

:validToQuarter a owl:DatatypeProperty ;
    rdfs:range :RemunCode;
.
```

Figure 18: Annex 7 Vocabulary[3]

The mappings were simple to generate as most properties have a corresponding element in the XML. Nevertheless, some properties required some data transformations. This will be discussed in more detail in Section 5.3.2. As for DmfA XML files, the mappings were passed to RMLMapper and the Annex 7 graph was generated.

This integration process is valid for any annex, but not all the annexes were integrated as part of this thesis. The excluded annexes are either the ones that are no longer required to validate the current version of the DmfA or those that require domain experts to clarify aspects. Indeed, one annex seemed unclear and complex, and we preferred deferring the integration over the integration of potentially nonsensical data. These annexes should still be integrated for retro-compatibility and completeness, but that is for future work. It is believed that the exercise will be similar to the other annexes once a domain expert can clarify its structure and meaning, and excluding them will not impact the conclusions we can draw from this study. The status of the annexes' integration is summarized in Table [2]. Of the 24 annexes, only five were excluded; four are currently irrelevant, and one unclear.

[3]Base IRI: `http://kg.socialsecurity.be/ressources/annex7#`

Table 2: Integration of Annexes status

Name	**Integrated**	**Reason (If no)**	**Addition of valid quarters**
1 - Codes communes - Code INS	No	Not needed	/
2 - Liste des codes travailleurs pour lesquels des cotisations sont dues	Yes	/	No
3 - Valeurs autorisées pour le "type de cotisation" en fonction des codes travailleurs cotisations	No	Too complex	/
4 - Liste des codes déductions	Yes	/	No
5 - Liste des codes pays	No	Not needed	/
6 - Activité par rapport au risque	Yes	/	Yes
7 - Codification des rémunérations	Yes	/	Yes
8 - Codification des données de temps de travail	Yes	/	No
9 - Numéros de fonction du personnel déclaré au forfait suivant l'indice de catégorie de l'employeur	Yes	/	Yes
10 - Codification des indemnités pour les catégories employeurs 027 et 028	Yes	/	Yes
11 - Identification du formulaire	Yes	/	Yes
21 - Liste des valeurs autorisées pour le statut du travailleur	Yes	/	No
23 - Liste des codes postaux et communes en 30 positions	No	Not needed	/
24 - Table de conversion - ASCII - codepage 850	No	Not needed	/
27 - Catégories d'employeurs	Yes	/	Yes
28 - Code travailleur APL	Yes	/	Yes
31 - Liste des codes NACE	Yes	/	Yes
35 - Mesure de promotion de l'emploi	Yes	/	No
42 - Nomenclature des types d'institutions du secteur public	Yes	/	No
43 - Nomenclature des catégories de personnel du secteur public	Yes	/	No
44 - Mesures de réorganisation du travail	Yes	/	No
45 - Nomenclature des Classes du personnel	Yes	/	Yes
46 - Détail secteur	Yes	/	Yes
dmfas02 - Détail des données pour les déductions	Yes	/	No

5.3.2 Data transformation

When analyzing the annexes, one can observe that some attributes are formatted in a "human-readable" way rather than in an XML datatype. These attributes were transformed such that a computer agent could perform operations on them. For instance, the annexes use Yes/No values instead of boolean values and the `dd/mm/yyyy` format instead of `yyyy-mm-dd`.

As a reminder, `rr:logicalTable` represents the SQL query that an engine will send to an RDB and then transform into RDF. The advantage of SQL is that the Data Query Language (DQL) allows us to declare data transformations in the SQL query. Examples of such transformations include data formatting, string formatting,

and concatenation. This cannot be done with RML when the data source is not relational. For XML, for instance, the logical source of a mapping uses an XPath expression to iterate over the XML resulting XML sub-documents. Hopefully, the Function Ontology (FnO) can be used to describe the data transformations in an interoperable way. In short, this ontology is a way to semantically describe functions and related concepts such as parameters and executions [25]. Note that an RML processor must support FnO to take the transformations into account.

In Figure [19], we present an example of an RML mapping that uses FnO. The mapping illustrates how to generate the predicate and object of a triple. The predicate is `ont:validFrom`, and the object is generated by applying the `grel:array_join` function (1) applied on the `grel:p_string_sep` (2) and `grel:p_array_a` (3) parameters. The latter is itself the returned value of another function. All in all, the object is the result of the pseudocode in Listing [4]. The `begin_geldigheid` (start validity date) is split into an array based on the "/" separator. This array is reversed and joined with the "-" separator. This changes the format of the date from `dd/mm/yyyy` to `yyyy-mm-dd`.

```
data = array_join(array_reverse(string_split(begin_geldigheid,"/")),"-")
```

Listing 4: FnO function pseudocode

This shows that the generation of an RDF graph with some data transformation is possible with RML and FnO. While it is possible to define these transformations in an interoperable way, there are some limitations to this approach. In particular, the set of functions supported by a specific RML-FnO processor may be limited, which can make it challenging to express certain transformations, even if they are relatively simple. As an example, the transformation of the date format using just three functions already takes up a lot of space. To reduce its size, one could define a single function transforming dates in the `dd/mm/yyyy` format to `yyyy-mm-dd` format. Nevertheless, this function should not have an implementation in any RML-FnO processor. Thus, one would also require providing an implementation that the processor can utilize to generate the graph.

Due to these limitations, only one annex was integrated with FnO data transformations as a proof of concept. The other annexes were processed using Python scripts to handle the necessary transformations.

```
47      rr:predicateObjectMap [
48          rr:predicate ont:validFrom ;
49          rr:objectMap [
50              rr:datatype xs:date ;
51              fnml:functionValue [
52                  rr:predicateObjectMap [                    1
53                      rr:predicate fno:executes ;
54                      rr:objectMap [
55                          rr:constant grel:array_join ;
56                          rr:termType rr:IRI ;
57                      ] ;
58                  ] ;
59
60                  rr:predicateObjectMap [                    2
61                      rr:predicate grel:p_string_sep ;
62                      rr:objectMap [
63                          rr:constant "-" ;
64                          rr:termType rr:Literal ;
65                      ] ;
66                  ] ;
67
68                  rr:predicateObjectMap [                                                    3
69                      rr:predicate grel:p_array_a ;
70                      rr:objectMap [
71                          fnml:functionValue [
72                              rr:predicateObjectMap [
73                                  rr:predicate fno:executes ;
74                                  rr:objectMap [
75                                      rr:constant grel:array_reverse ;
76                                      rr:termType rr:IRI ;
77                                  ] ;
78                              ] ;
79
80                              rr:predicateObjectMap [
81                                  rr:predicate grel:p_array_a ;
82                                  rr:objectMap [
83                                      fnml:functionValue [
84                                          rr:predicateObjectMap [
85                                              rr:predicate fno:executes ;
86                                              rr:objectMap [
87                                                  rr:constant grel:string_split ;
88                                                  rr:termType rr:IRI ;
89                                              ] ;
90                                          ] ;
91
92                                          rr:predicateObjectMap [
93                                              rr:predicate grel:p_string_sep ;
94                                              rr:objectMap [
95                                                  rr:constant "/" ;
96                                                  rr:termType rr:Literal ;
97                                              ] ;
98                                          ] ;
99
100                                         rr:predicateObjectMap [
101                                             rr:predicate grel:valueParameter ;
102                                             rr:objectMap [
103                                                 rml:reference "begin_geldigheid" ;
104                                             ] ;
105                                         ] ;
106                                     ] ;
107                                 ] ;
108                             ] ;
109                         ] ;
110                     ] ;
111                 ] ;
112             ] ;
113         ] ;
114     ] ;
```

Figure 19: RML-FnO mapping example

5.3.3 IRI strategy

Resources of the annexes are named by `http://kg.socialsecurity.be/resource/{annexId}/{ref}` where `{annexId}` is the annex identifier and `{ref}` is an identifier of the resource. `{ref}` follows the scheme `{class}{id}-{quarter}` where `{class}` is the vocabulary class, `{id}` is a code assigned by the NOSS to the resource, and quarter is the start of the validity of the resource (i.e., from when it can be used in declarations)

Several considerations were taken into account when creating this IRI strategy, inspired by Tim Berners-Lee's guidelines on building IRIs [26]. One important factor was that resources from the same annex should be grouped together. To achieve this, `{annexId}` was used to create subdirectories for each annex. This also allows users to easily navigate between annexes by modifying the annex identifier, which is designed to be somewhat human-readable (e.g., `annex7`).

Even though the annexes that have been integrated only consist of a single class, other annexes or future annexes may have multiple classes. The `{class}` component is included in the IRI to avoid conflicting resource naming between these potential classes.

The annexes have an attribute that is a code used to identify a resource (i.e., `{id}` component), but this code is only unique at a particular point in time. To ensure that each resource can be uniquely identified, the code must be combined with a temporal aspect. The quarter from which the resource is valid (i.e., `{quarter}` component) was chosen as the temporal aspect. This choice of attribute allows the IRI to be permanent as the starting quarter is not subject to change. If a resource becomes invalid, the only change that will occur is the end of validity quarter being updated to the expiration quarter instead of being set to "end of times".

5.4 SHACL rules development

This section covers the process of creating SHACL rules for the DmfA. It covers the generation of the rules, common patterns that emerged, design considerations, and the extension of these rules to other declarations.

5.4.1 Rules generation

The base shapes for a class and data property were created by processing the XSD. Figure [20] shows these basic shapes. The starting point for a class' shape is a rule stating that each instance of the class should have valid data properties. Other shapes describe criteria that these data properties must meet, such as specified datatypes, value and length constraints, and patterns. These criteria are the SHACL equivalent of XSD constraints.

```
kg:EmployerDeclarationShape a sh:NodeShape ;
    rdfs:comment "Property Shape for EmployerDeclaration (90007)" ;
    sh:targetClass ont:EmployerDeclaration ;

    sh:property kg:QuarterShape;
    sh:property kg:NOSSRegistrationNbrShape;
    sh:property kg:TrusteeshipShape;
    sh:property kg:CompanyIDShape;
    sh:property kg:NetOwedAmountShape;
    sh:property kg:System5Shape;
    sh:property kg:HolidayStartingDateShape;
.
```

```
kg:QuarterShape a sh:PropertyShape;
    rdfs:comment "Property Shape for Quarter (00013)" ;
    sh:path ont:Quarter;

    sh:datatype xs:integer;
    sh:minInclusive 20031;
    sh:maxLength 5;
    sh:pattern "\\d{4}(1|2|3|4)";
.
```

Figure 20: Basic DmfA rules

These rules do not cover all the constraints that the declaration should respect. Thus, they were manually verified and refined. The glossary of each class, data property and object property were consulted to determine missing rules. Missing rules concerning classes and data properties were added to the base shapes, whereas object properties' rules were added to the shape of their domain. Figure [21] shows how the rules concerning the Quarter field were refined. Some of the missing rules can be expressed with the SHACL-core component, others had to be expressed with a SHACL-SPARQL constraint. Both cases represent rules that SHACL was able to support, and this already demonstrates that SHACL is more expressive than XSD.

This refinement process was tedious as the number of classes, data properties and object properties were vast. For each shape, some test cases were designed to verify the correctness of the rules. Nevertheless, as the number of implemented shapes increased, some patterns emerged and will be discussed in the next section.

NUMERO DE ZONE: 00013	VERSION: 2022/3	DATE DE PUBLICATION: 30/08/2022

ANNÉE - TRIMESTRE
(Label XML : Quarter)

BLOC FONCTIONNEL: Déclaration employeur
Code(s): 90007
Label(s) xml: EmployerDeclaration

DESCRIPTION: Détermination de l'année et du trimestre.

Détermination de l'année et du trimestre de la déclaration.

DOMAINE DE DEFINITION: Il doit être compris entre le premier trimestre 2003 (20031) et le dernier trimestre civil échu (AAAAT en cours -1).

REFERENCE LEGALE:

TYPE: Numérique
LONGUEUR: 5

PRESENCE: Indispensable

FORMAT: AAAAT
. AAAA est l'année
. T est le trimestre

CODE ANOMALIE SUR ACCUSE DE RECEPTION:

Intitulé anomalie	Code anomalie	Gravité
Non présent	00013-001	B
Non numérique	00013-002	B
Invalide	00013-003	B
Pas dans le domaine de définition	00013-008	B
Longueur incorrecte	00013-093	B
Non admis	00013-146	B
Erreur de cardinalité	00013-090	B
Erreur de séquence	00013-091	B
Attention ! Trimestre en danger de prescription ou prescrit.	00013-313	B

```
kg:QuarterShape a sh:PropertyShape;
    rdfs:comment "Property Shape for Quarter (00013)" ;
    sh:path ont:Quarter;
    sh:datatype xs:integer;
    sh:minCount 1 ;
    sh:maxCount 1 ;
    sh:minLength 5 ;
    sh:maxLength 5 ;
    sh:minInclusive 20031 ;
    sh:pattern "^\\d{4}[1-4]$" ;
    sh:sparql [
        sh:message "Quarter beyond [20031, last quarter in progress - 1]." ;
        sh:prefixes <> ;
        sh:select """
            SELECT $this ?value
            WHERE {
                $this $PATH ?value .
                BIND ( MONTH ( NOW() ) as ?currentMonth)
                BIND (
                    COALESCE (
                        IF(?currentMonth <= 3, 1, ""),
                        IF(?currentMonth <= 6, 2, ""),
                        IF(?currentMonth <= 9, 3, ""),
                        IF(?currentMonth <= 12, 4, ""),
                        ""
                    ) as ?quarter
                )
                BIND ( (YEAR ( NOW() ) * 10 + ?quarter) as ?currentquarter)
                BIND ( IF(?currentquarter = 1, ?currentquarter - 7, ?currentquarter - 1) as ?maxAllowedQuarter)
                FILTER ( ?value < 20031 || ?maxAllowedQuarter < ?value)
            }"""
    ]
.
```

Figure 21: Glossary of Quarter (left) and refined shape of Quarter (right)

5.4.2 Patterns

In this section, we will present and explain the patterns that were identified during the development of the SHACL rules. Our aim is to provide rules that can be applied in contexts beyond the social security domain, or to serve as a model for creating other rules, by presenting the structure of the developed rules.

5.4.2.1 Checksums

Checksums can be used to verify the correctness of identifiers such as a company ID, identification number of the social security (INSS), or NOSS registration number. Checksums can be distinguished by the algorithm used for the verification. Each of the checksums can be expressed with a rule implementing the algorithm. Listing [5] illustrates a checksum rule where the 97 minus the last two digits must be equal to the modulo 97 of the number before the last two digits. (e.g., `123427: (1234 mod 97) ?= 97 - 27`). Despite expressing a specific checksum, this constraint can be reused as is for numbers of any length.

```
[
    sh:message "Checksum is wrong." ;
    sh:prefixes <> ;
    sh:select """
        SELECT $this ?value
        WHERE {
            $this $PATH ?value .
            BIND( FLOOR(?value / 100) AS ?number )
            BIND( ?value - (100 * FLOOR(?value / 100)) AS ?check )
            BIND( ?number - (97 * FLOOR(?number / 97)) AS ?rest )
            BIND( 97 - ?rest AS ?check2 )
            FILTER ( ?check != ?check2 )
        }""" ;
] .
```

Listing 5: SHACL-SPARQL checksum constraint example

5.4.2.2 Existing code w.r.t annexes

Some annexes define values that specific fields can accept. Listing [6] illustrates the validity checking for the `ont:PositionCode` field with a set of permitted values defined by Annex 9. This rule checks if at least one resource of type `an9:PositionCode` with a code equal to the field's value. The main advantage of this rule is that it is up to date with the current state of the annex.

```
[
    sh:message "Invalid code for a position, code does not exist" ;
    sh:prefixes <> ;
    sh:select """
        SELECT $this ?value
        WHERE {
            $this ont:PositionCode ?value.
            OPTIONAL{
                ?pc a an9:PositionCode ; an9:Code ?value.
            }
            FILTER(!BOUND(?pc))
        }""" ;
] .
```

Listing 6: SHACL-SPARQL existing code constraint example

5.4.2.3 Code within valid period w.r.t annexes

The annexes defining values for certain fields also define their validity period. As explained in Section 5.3.1, data properties concerning the starting and ending quarter of the validity of a code were added to annexes specifying them with dates. This addition made the rules checking the temporal validity of a code follow the same pattern. It also eases the checking as the quarter of declaration must not be transformed into a date before being compared to a validity period. An example of this type of rule is presented in Listing [7]. The quarter of the declaration is reached through the inverse path from a resource and compared to the starting and ending quarter of the code matching the value of the `ont:PositionCode` data property. Like the previous rule, being up to date with the current state of the annex is the main advantage of this rule.

```
[
    sh:message "Invalid ont:PositionCode, code is out of valid quarter range." ;
    sh:prefixes <> ;
    sh:select """
        SELECT $this ?value
        WHERE {
            $this ont:PositionCode ?value.
            OPTIONAL {
                ?pc a an9:PositionCode;
                    an9:Code ?value;
                    an9:validFromQuarter ?startQuarter;
                    an9:validToQuarter ?endQuarter;
                .
                $this ^ont:R_90012_90015/
                      ^ont:R_90017_90012/
                      ^ont:R_90007_90017/
                      ont:Quarter ?quarter.
                FILTER( ?startQuarter < ?quarter && ?quarter < ?endQuarter)
            }
            FILTER(!BOUND(?pc))
        }""" ;
] .
```

Listing 7: SHACL-SPARQL code within valid period constraint example

5.4.2.4 Field presence w.r.t annexes

Two annexes specified the presence of fields based on the value of a particular field. Listing [8] illustrates a rule verifying the correct presence of the `ont:ReplacedINSS` data property. A subquery counts the data property's number of occurrences. This number is compared to the expected count which depends on the value of `ont:DeductionCode`.

```
[
    sh:message "Wrong cardinality for ont:ReplacedINSS." ;
    sh:prefixes <> ;
    sh:select """
        SELECT $this
        WHERE {
            {
                SELECT $this (SUM (?occ) as ?nbrOcc)
                WHERE {
                    $this ?p ?o.
                    BIND( IF(?p = ont:ReplacedINSS, 1, 0) AS ?occ)
                }
                GROUP BY $this
            }
            $this ont:DeductionCode ?code.
            ?dc a an4:DeductionCode;
                an4:Code ?code;
                an4:ReplacedINSSSPresence ?presence;
            .
            BIND(   IF(?presence = "Obligatoire"@fr, 1,
                    IF(?presence = "Interdit"@fr, 0,
                    IF(?presence = "Optionnel"@fr, ?nbrOcc, -1))) AS ?expectdNbrOcc)
            FILTER(?expectdNbrOcc != ?nbrOcc)
        }""" ;
] .
```

Listing 8: SHACL-SPARQL field presence constraint example

5.4.2.5 Unicity

Some rules described in the glossary state that some values of an entity must be unique. Listing [9] illustrates a rule expressing the unicity of natural person sequence numbers in an employer declaration. This rule counts the number of occurrences for a sequence number. The sequence number is not unique if this number is greater than one.

This pattern can be applied to a combination of fields, as illustrated in Listing [10]. Elements are cast as strings and concatenated to represent a combination. Then, the unicity checking is done by counting the combinations' number of occurrences. Note that two distinct combinations may appear as the same concatenated strings. For instance, the concatenation of the pairs (`123`, `45`) and (`1`, `2345`) are both equal to `12346`. Nevertheless, this exception will not occur if elements are of a fixed length, which is the case for the developed rules. If the elements are not of fixed length, separators could be used but at the cost of doubling the number of string concatenations.

```
[
    sh:message "Each ont:NaturalPersonSequenceNbr must be unique for a ont:EmployerDeclaration." ;
    sh:prefixes <> ;
    sh:select """
        SELECT $this
        WHERE {
            {
                SELECT $this (COUNT(?seqNbr) as ?seqNbrOcc)
                WHERE {
                    $this ont:R_90007_90017/ont:NaturalPersonSequenceNbr ?seqNbr .
                }
                GROUP BY ?seqNbr $this
            }
            FILTER(?seqNbrOcc > 1)
        }""" ;
] .
```

Listing 9: SHACL-SPARQL unicity constraint example

```
[
    sh:message "A combination of ont:RemunCode, ont:PercentagePaid and ont:BonusPaymentFrequency
    ↪  must not appear several time for the same ont:Occupation." ;
    sh:prefixes <> ;
    sh:select """
        SELECT $this
        WHERE {
            {
                SELECT $this ?combination (COUNT(?combination) as ?combinationOcc)
                WHERE {
                    $this ont:R_90015_90019 ?r .
                    ?r ont:RemunCode ?rc;
                        ont:PercentagePaid ?pp;
                        ont:BonusPaymentFrequency ?bpf;
                    .

                    BIND(
                        CONCAT( STR(?rc),
                        CONCAT( STR(?pp),
                                STR(?bpf)))
                        as ?combination)
                }
                GROUP BY $this ?combination
            }
            FILTER(?combinationOcc >1 )
        }""" ;
] .
```

Listing 10: SHACL-SPARQL combination unicity constraint example

5.4.3 Design considerations

During the implementation of SHACL rules, various design choices were made. In this section, we will present these choices and the considerations that led to their implementation. We will also compare the trade-offs and potential impacts of each choice on the effectiveness and efficiency of the SHACL rules. These considerations should be taken into account as they can have significant impacts on the usability and maintenance of the SHACL rules.

The performance of the validation can vary depending on the SHACL processor used.

For this experiment, we only used the TopBraid SHACL processor [27]. However, we expect that other processors should give similar results. The mean time required for one hundred validations was used as the metric to compare the implementation of the rules.

5.4.3.1 SPARQL-based constraint optimization

When developing SPARQL-based constraints, some optimization was made on the SPARQL query they are based on. In the section, we will give two examples of queries that were optimized.

Checksum A first implementation of a checksum constraint splits the number using type casting and the substring function. However, string manipulation is often slower than operating on integers. Thus, this query was improved by performing a modulo operation to split the number. A reduction of 200 milliseconds was achieved on 5000 numbers to verify. We refer to Appendix B.3.1 for the explicit SPARQL query and benchmark results.

Code existence The two implementations for a code existence constraint involve either counting the number of resources with a matching code matching the value or checking if a resource with a matching code is bound. The former involves aggregation, which can create an overhead. Empirical results show a reduction of 330 milliseconds when verifying 5000 codes. We refer to Appendix B.3.2 for the explicit SPARQL query and benchmark results.

5.4.3.2 Selecting the target of a rule

We explain in Section 5.4.1 that the rules described by the glossary concerning a class are regrouped into a single shape. These shapes express all the constraints that instances of this class must respect. However, strict adherence to this approach may significantly slow down the validation process. In fact, it is possible to use SPARQL queries to navigate through the graph and express constraints from any resource because the declaration's graph does not have isolated vertices. Thus, adapting a rule to target another class may be more efficient.

An example of a data graph respecting unique `ont:NaturalPersonSequenceNbr` value for a `ont:EmployerDeclaration` is illustrated in Figure [22]. The unicity constraint can either be placed at the `ont:EmployerDeclaration` level or the `ont:NaturalPerson` level. The former is more efficient as only a single query verifies that no children have the same sequence number. The number of queries involved in the latter increases linearly with the number of natural persons, which negatively impacts the validation time.

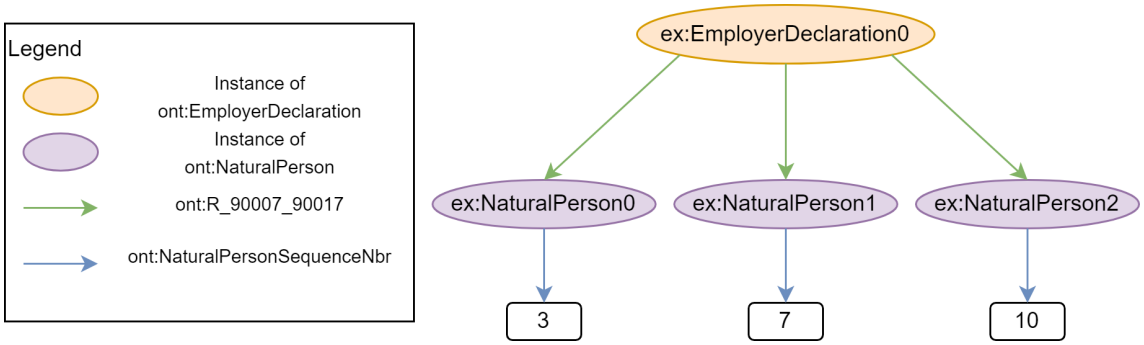


Figure 22: Valid data graph with unique sequence number

Figure [23] presents empirical evidence supporting these claims. One might expect the validation time for a constraint at the child level to increase quadratically since each child node compares its sequence number to all its siblings. However, the results show a linear increase of the validation time. This can be explained by the fact that the increase in complexity of the query (i.e., comparing to more natural persons) does not significantly affect the query time. The increase is mostly due to the increase of queries' amount. Similarly, for a constraint at the parent level, the validation time remains nearly constant rather than increasing linearly. We refer to Appendix B.4 for the explicit SPARQL query and benchmark results.

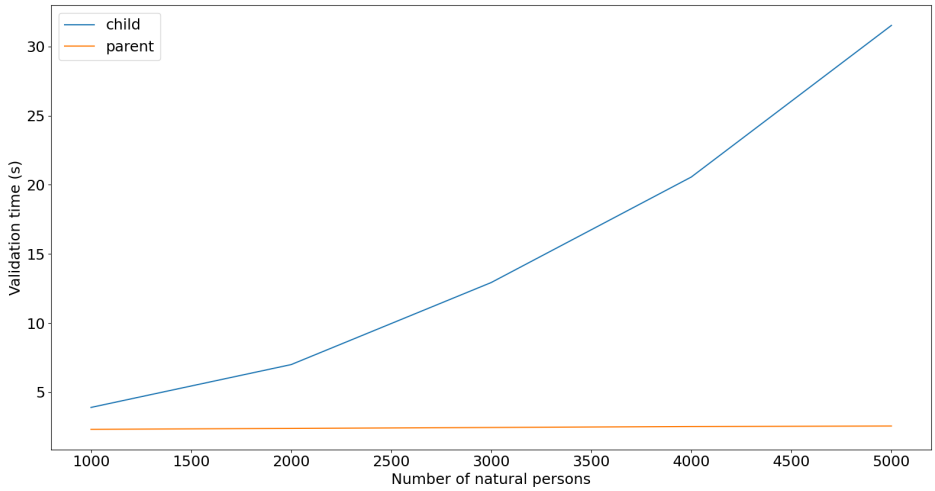


Figure 23: Validation time for a uniqueness constraint as a function of the number of natural persons

5.4.3.3 Redundancy

A shape's constraints may have overlapping checks. For example, consider the shape of an ont:Quarter shown in Figure [24]. If the length constraints are not met, the

pattern constraint also fails. Similarly, if the value is less than 20031, both the minimum value constraint and the SPARQL-based constraint will fail.

While the length and minimum value constraints are redundant, they offer a more specific reason for failure. An error indicating a violation of the minimum length directly reflects a missing character, whereas a mismatching pattern error does not. For this reason, redundant but more precise constraints were favored. Nevertheless, this choice comes with a decrease in efficiency as more checks must be performed.

```
kg:QuarterShape a sh:PropertyShape;
    rdfs:comment "Property Shape for Quarter (00013)" ;
    sh:path ont:Quarter;
    sh:datatype xs:integer;
    sh:minCount 1 ;
    sh:maxCount 1 ;
    sh:minLength 5 ;
    sh:maxLength 5 ;
    sh:minInclusive 20031 ;
    sh:pattern "^\\d{4}[1-4]$" ;
    sh:sparql [
        sh:message "Quarter beyond [20031, last quarter in progress - 1]." ;
        sh:prefixes <> ;
        sh:select """
            SELECT $this ?value
            WHERE {
                $this $PATH ?value .
                BIND ( MONTH ( NOW() ) as ?currentMonth)
                BIND (
                    COALESCE (
                        IF(?currentMonth <= 3, 1, ""),
                        IF(?currentMonth <= 6, 2, ""),
                        IF(?currentMonth <= 9, 3, ""),
                        IF(?currentMonth <= 12, 4, ""),
                        ""
                    ) as ?quarter
                )
                BIND ( (YEAR ( NOW() ) * 10 + ?quarter) as ?currentquarter)
                BIND ( IF(?currentquarter = 1, ?currentquarter - 7, ?currentquarter - 1) as ?maxAllowedQuarter)
                FILTER ( ?value < 20031 || ?maxAllowedQuarter < ?value)
            }"""
    ]
.
```

Figure 24: Quarter Shape

5.4.3.4 Clarity vs. efficiency

When developing the rules, expressing rules in a clear and easily understandable manner was favored other making them as efficient as possible. Clear rules are often the result of a combination of simple constraints that make them easier to maintain. As they are fragmented, parts of these rules can be reused for other purposes. On the other hand, efficient rules are more likely to be complex and harder to maintain.

To illustrate this point, Figure [25] presents a clear and efficient version of a shape for a `ont:NOSSRegistrationNbr`. The constraints on the left are easy to understand: the registration number must either be in the range `[100006; 199999934]` and respect a checksum, or it must be equal to `0` when the company ID is known.

Whereas the left version expresses rules with two SPARQL queries, the right version achieves the same results more efficiently with a single SPARQL query but is harder to understand. However, if the number can no longer be equal to `0`, the left version can be easily adapted by removing the second condition of the `sh:or`. On the other hand, the right version would require modifying several parts of the SPARQL query in a non-obvious manner to achieve the same change.

```
kg:NOSSRegistrationNbrShape a sh:NodeShape;
    rdfs:comment "Property Shape for NOSSRegistrationNbr (00011)" ;
    sh:or (
        [
            sh:path ont:NOSSRegistrationNbr;
            sh:minInclusive 100006 ;
            sh:maxInclusive 199999934 ;
            sh:sparql [
                sh:message "Invalid NOSS registration number: Checksum is wrong" ;
                sh:prefixes <> ;
                sh:select """
                    SELECT $this ?value
                    WHERE {
                        $this $PATH ?value .
                        BIND( FLOOR(?value / 100) as ?number )
                        BIND( ?value - (?number * 100) AS ?check )
                        BIND( ?number - (97 * FLOOR(?number / 97)) AS ?rest )
                        BIND( 96 - ?rest AS ?check2 )
                        FILTER ( ?check != ?check2 )
                    }
                    }"""
            ] ;
        ]
        [
            sh:path ont:NOSSRegistrationNbr;
            sh:in (0) ;
            sh:sparql [
                sh:message "CompanyID (00014) is unknown : value is 0" ;
                sh:prefixes <> ;
                sh:select """
                    SELECT $this ?value
                    WHERE {
                        $this ont:CompanyID ?value .
                        FILTER ( ?value = 0 )
                    }"""
            ] ;
        ]
    ) ;
.
```

```
kg:NOSSRegistrationNbrShape a sh:PropertyShape;
    rdfs:comment "Property Shape for NOSSRegistrationNbr (00011)" ;
    sh:sparql [
        sh:message """Invalid NOSS registration number: Checksum is wrong or
                      value is 0 and ont:CcompanyId is 0""" ;
        sh:prefixes <> ;
        sh:select """
            SELECT $this  ?value
            WHERE {
                $this ont:CompanyID ?companyId .
                $this $PATH ?value .
                BIND( FLOOR(?value / 100) as ?number )
                BIND( ?value - (?number * 100) AS ?check )
                BIND( ?number - (97 * FLOOR(?number / 97)) AS ?rest )
                BIND( 96 - ?rest AS ?check2 )
                FILTER (
                    (?value = 0 && ?companyId = 0) ||
                    (0 < ?value && ?value < 100006) ||
                    (100006 <= ?value && ?value <= 199999934 && ?check != ?check2) ||
                    (199999934 < ?value)
                )
            }"""
    ] ;
.
```

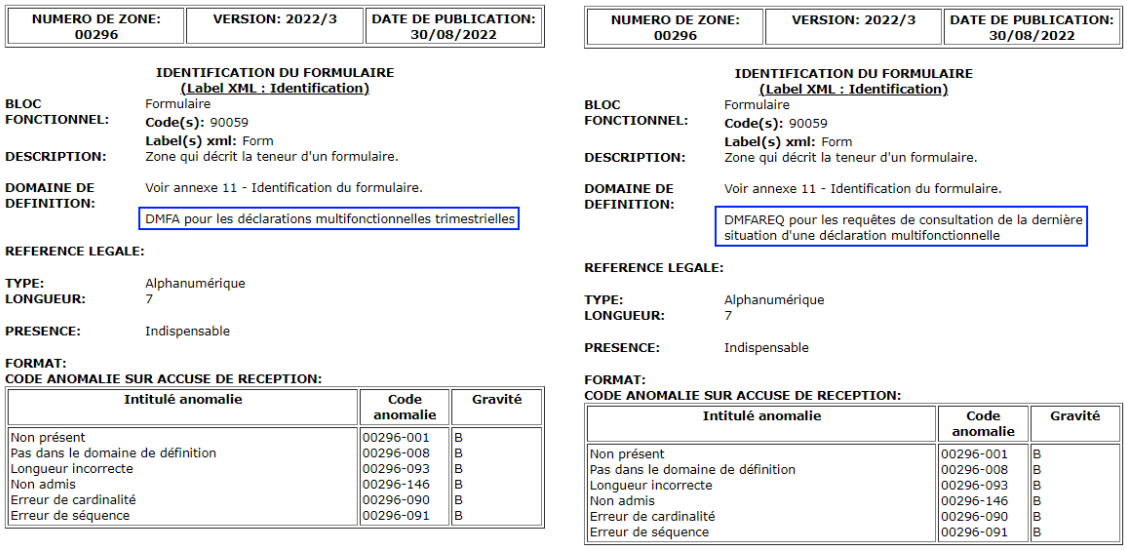
Figure 25: kg:NOSSRegistrationNbrShape, clear version (left) and efficient version (right)

5.4.4 Extending rules to other quarters and declarations

Up to this point only the rules of a DmfA for the 2022/3 quarter were developed. However, there are many more declarations, each with their own set of rules. Apart from the three main declarations (DmfA, Dimona, and DRS), there are declarations linked to the main ones such as modification and consultation request declarations. Moreover, each of these declarations have multiple versions, one for each quarter. In this section, we present how the rules for the main DmfA declaration were extended to validate other declarations and their different versions.

By comparing the several declarations, many similarities were found. They share classes, object properties and, data properties which must respect nearly identical constraints. For example, Figure [26] highlights the only difference for the `Identification` property between the glossary of an original DmfA (left) and a DmfA consultation request (right).

To avoid expressing nearly equivalent shapes repeatedly across declarations, constraints common to all declarations were regrouped into a common shapes graph.



NUMERO DE ZONE: 00296	VERSION: 2022/3	DATE DE PUBLICATION: 30/08/2022

IDENTIFICATION DU FORMULAIRE
(Label XML : Identification)

BLOC FONCTIONNEL: Formulaire
Code(s): 90059
Label(s) xml: Form
DESCRIPTION: Zone qui décrit la teneur d'un formulaire.

DOMAINE DE DEFINITION: Voir annexe 11 - Identification du formulaire.

DMFA pour les déclarations multifonctionnelles trimestrielles

REFERENCE LEGALE:

TYPE: Alphanumérique
LONGUEUR: 7

PRESENCE: Indispensable

FORMAT:
CODE ANOMALIE SUR ACCUSE DE RECEPTION:

Intitulé anomalie	Code anomalie	Gravité
Non présent	00296-001	B
Pas dans le domaine de définition	00296-008	B
Longueur incorrecte	00296-093	B
Non admis	00296-146	B
Erreur de cardinalité	00296-090	B
Erreur de séquence	00296-091	B

NUMERO DE ZONE: 00296	VERSION: 2022/3	DATE DE PUBLICATION: 30/08/2022

IDENTIFICATION DU FORMULAIRE
(Label XML : Identification)

BLOC FONCTIONNEL: Formulaire
Code(s): 90059
Label(s) xml: Form
DESCRIPTION: Zone qui décrit la teneur d'un formulaire.

DOMAINE DE DEFINITION: Voir annexe 11 - Identification du formulaire.

DMFAREQ pour les requêtes de consultation de la dernière situation d'une déclaration multifonctionnelle

REFERENCE LEGALE:

TYPE: Alphanumérique
LONGUEUR: 7

PRESENCE: Indispensable

FORMAT:
CODE ANOMALIE SUR ACCUSE DE RECEPTION:

Intitulé anomalie	Code anomalie	Gravité
Non présent	00296-001	B
Pas dans le domaine de définition	00296-008	B
Longueur incorrecte	00296-093	B
Non admis	00296-146	B
Erreur de cardinalité	00296-090	B
Erreur de séquence	00296-091	B

Figure 26: Glossary of Identification for an original DmfA (left) and a DmfA consultation request (right)

In addition, constraints specific to a declaration were stored into a specific shapes graph. Thus, combining the common and specific shapes graph results in a shapes graph expressing all the rules for a declaration. In fact, this combination is only valid because specific rules are more restrictive and thus being a valid declaration implies respecting the common rules. The resulting separation of constraints for the `Identification` property can be seen in Figure [27].

Extending to other quarters was more complex as previous and future constraints can be more inclusive or restrictive. Therefore, the previous method of separation is not suitable in this case. It might have been possible to identify the shapes related to a specific quarter by adding some additional annotations to the shapes, but this would require additional computation to retrieve the shapes and it was not clear how to best organize them. The proposed solution is to include all the shapes related to a quarter in a single graph, resulting in each quarter having its own shapes graph. The simplicity of this solution makes it easier to maintain at the cost of a slight increase in memory usage.

In short, if `Q` is the number of quarter and `D` the number of declarations, the number of different shapes graph would be `(Q * (D + 1))`. For each of the `Q` quarter, there are `D` specific shapes graph and 1 common shapes graph.

Common

```
kg:IdentificationShape a sh:PropertyShape;
    rdfs:comment "Property Shape for Identification (00296)" ;
    sh:path ont:Identification;
    sh:datatype xs:string;
    sh:minCount 1 ;
    sh:maxCount 1 ;
    sh:maxLength 7 ;
    sh:sparql [
        sh:message "Invalid ont:Identification, code is out of valid quarter range." ;
        sh:prefixes <> ;
        sh:select """
            SELECT $this ?value
            WHERE {
                $this $PATH ?value.
                OPTIONAL{
                    ?it a an11:IdentificationType;
                        rdfs:label ?value;
                        an11:validFromQuarter ?startQuarter;
                        an11:validToQuarter ?endQuarter;
                    .
                    $this ont:R_90059_90007/ont:Quarter ?quarter.
                    FILTER(?startQuarter < ?quarter && ?quarter < ?endQuarter)
                }
                FILTER(!BOUND(?it))
            }""" ;
    ] ;

    sh:sparql [
        sh:message "Invalid Document type, does not exist" ;
        sh:prefixes <> ;
        sh:select """
            SELECT $this ?value
            WHERE {
                $this $PATH ?value.
                OPTIONAL{
                    ?it a an11:IdentificationType ; rdfs:label ?value.
                }
                FILTER(!BOUND(?it))
            }""" ;
    ] ;
.
```

DmfA

```
kg:DmfAIdentificationShape a sh:PropertyShape;
    rdfs:comment "Property Shape for Identification (00296)" ;
    sh:path ont:Identification;
    sh:in ("DMFA") ;
.
```

DmfaREQ

```
kg:DmfAREQIdentificationShape a sh:PropertyShape;
    rdfs:comment "Property Shape for Identification (00296)" ;
    sh:path ont:Identification;
    sh:in ("DMFAREQ") ;
.
```

Figure 27: Example of constraints separation into common shape, DmfA specific shape and DmfAREQ specific shape

5.5 Validation process

As a proof of concept, a Jupyter Notebook was developed as prototype of user interface for the validation of declarations. Currently, this program can validate two types of declaration, DmfA and DmfAREQ, for the quarters between 2022/1 and 2022/3, both included. This notebook communicates with a SPARQL server storing the annexes and the 9^4 shapes graphs required for the validation. Its capabilities can be extended to other declarations and quarters by storing their shapes graphs on the server.

The program starts by taking user input for the file to validate and the declaration to validate against. Then, it sends the SPARQL query in Listing [11] to the endpoint. This query fetches all the RDF graphs of all the annexes from their respective named graph. The result is appended to a copy of the input file and forms the data graph. A second query with the selected declaration type and quarter, shown in Listing [12], is sent to the endpoint to retrieve the shapes graph. This shapes graph is the combination of a common and a specific shapes graph. The last step is launching the TopBraid SHACL processor with the data and shapes graphs as parameters and saving resulting validation graph into a file.

```
CONSTRUCT{
    ?s ?p ?o.
}
FROM <http://kg.socialsecurity.be/resource/annex2/>
FROM <http://kg.socialsecurity.be/resource/annex4/>
FROM <http://kg.socialsecurity.be/resource/annex6/>
FROM <http://kg.socialsecurity.be/resource/annex7/>
FROM <http://kg.socialsecurity.be/resource/annex8/>
FROM <http://kg.socialsecurity.be/resource/annex9/>
FROM <http://kg.socialsecurity.be/resource/annex10/>
FROM <http://kg.socialsecurity.be/resource/annex11/>
FROM <http://kg.socialsecurity.be/resource/annex21/>
FROM <http://kg.socialsecurity.be/resource/annex27/>
FROM <http://kg.socialsecurity.be/resource/annex28/>
FROM <http://kg.socialsecurity.be/resource/annex31/>
FROM <http://kg.socialsecurity.be/resource/annex35/>
FROM <http://kg.socialsecurity.be/resource/annex42/>
FROM <http://kg.socialsecurity.be/resource/annex43/>
FROM <http://kg.socialsecurity.be/resource/annex44/>
FROM <http://kg.socialsecurity.be/resource/annex45/>
FROM <http://kg.socialsecurity.be/resource/annex46/>
FROM <http://kg.socialsecurity.be/resource/dmfas02/>
WHERE{
    ?s ?p ?o.
}
```

Listing 11: CONSTRUCT query to fetch all annexes

[4]By using the formula `(Q * (D + 1))` from Section 5.4.4, `9 = (3 * (2 + 1))`

```
CONSTRUCT{
    ?s ?p ?o.
}
FROM <http://kg.socialsecurity.be/resource/shapes/common/quarter/>
FROM <http://kg.socialsecurity.be/resource/shapes/declarationType/quarter/>
WHERE{
    ?s ?p ?o.
}
```

Listing 12: CONSTRUCT query to fetch shapes graph for a specific declaration and quarter

5.5.1 Examples

We provide two examples of DmfA RDF declarations that can be used to tryout the validation process. Both examples are validated against the shapes graph of a DmfA for the 2022/3 quarter.

The first example was generated from the transformation of the DmfA XML example provided by the NOSS in their technical library. It is an invalid declaration and stored into the file named `invalid_dmfa.ttl`. Parts of the validation report can be viewed in Figure [28]. Apart from an error caused by an incorrect totalization of the contribution amount, all errors are due to invalid codes, either non-existent or outdated. Although this declaration respects the DmfA XSD, it does not respect the SHACL rules. This provides empirical evidence that SHACL is more expressive than XSD.

The second example is a corrected version of the first one for which each error of the validation report was removed. This file is named `valid_dmfa.ttl` As a result, the validation report for this example states that the declaration conforms to the shapes graph, as shown in Figure [29].

```
1  @prefix dash:    <http://datashapes.org/dash#> .
2  @prefix graphql: <http://datashapes.org/graphql#> .
3  @prefix kg:      <http://kg.socialsecurity.be/resource/shapes/> .
4  @prefix ont:     <http://kg.socialsecurity.be/ont/dmfa#> .
5  @prefix owl:     <http://www.w3.org/2002/07/owl#> .
6  @prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
7  @prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
8  @prefix sh:      <http://www.w3.org/ns/shacl#> .
9  @prefix swa:     <http://topbraid.org/swa#> .
10 @prefix tosh:    <http://topbraid.org/tosh#> .
11 @prefix xsd:     <http://www.w3.org/2001/XMLSchema#> .
12
13 [ rdf:type     sh:ValidationReport ;
14   sh:conforms  true
15 ] .
```

Figure 29: `valid_dmfa.ttl` validation report

```
@prefix dash:    <http://datashapes.org/dash#> .
@prefix graphql: <http://datashapes.org/graphql#> .
@prefix kg:      <http://kg.socialsecurity.be/resource/shapes/> .
@prefix ont:     <http://kg.socialsecurity.be/ont/dmfa#> .
@prefix owl:     <http://www.w3.org/2002/07/owl#> .
@prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix sh:      <http://www.w3.org/ns/shacl#> .
@prefix swa:     <http://topbraid.org/swa#> .
@prefix tosh:    <http://topbraid.org/tosh#> .
@prefix xsd:     <http://www.w3.org/2001/XMLSchema#> .

[ rdf:type     sh:ValidationReport ;
  sh:conforms  false ;
  sh:result    [ rdf:type                      sh:ValidationResult ;
                 sh:focusNode                  <urn:ss:NOREF-Remun3> ;
                 sh:resultMessage              "Invalid code for ont:RemunCode, does not exist" ;
                 sh:resultPath                 ont:RemunCode ;
                 sh:resultSeverity             sh:Violation ;
                 sh:sourceConstraint           _:b0 ;
                 sh:sourceConstraintComponent  sh:SPARQLConstraintComponent ;
                 sh:sourceShape                kg:RemunCodeShape ;
                 sh:value                      999
               ] ;
  sh:result    [ rdf:type                      sh:ValidationResult ;
                 sh:focusNode                  <urn:ss:NOREF-OccupationPublicServiceData0> ;
                 sh:resultMessage              "Invalid ont:PublicSectorPersonnelCategory code, code is out of valid quarter range." ;
                 sh:resultPath                 ont:PublicSectorInstitutionType ;
                 sh:resultSeverity             sh:Violation ;
                 sh:sourceConstraint           _:b1 ;
                 sh:sourceConstraintComponent  sh:SPARQLConstraintComponent ;
                 sh:sourceShape                kg:PublicSectorInstitutionTypeShape ;
                 sh:value                      99
               ] ;
  sh:result    [ rdf:type                      sh:ValidationResult ;
                 sh:focusNode                  <urn:ss:NOREF-Occupation3> ;
                 sh:resultMessage              "Invalid Employment Promotion, code does not exist" ;
                 sh:resultPath                 ont:EmploymentPromotion ;
                 sh:resultSeverity             sh:Violation ;
                 sh:sourceConstraint           _:b2 ;
                 sh:sourceConstraintComponent  sh:SPARQLConstraintComponent ;
                 sh:sourceShape                kg:EmploymentPromotionShape ;
                 sh:value                      0
               ] ;
...
  sh:result    [ rdf:type                      sh:ValidationResult ;
                 sh:focusNode                  <urn:ss:NOREF-OccupationPublicServiceData2> ;
                 sh:resultMessage              "Invalid ont:PublicSectorInstitutionType code, code does not exist" ;
                 sh:resultPath                 ont:PublicSectorInstitutionType ;
                 sh:resultSeverity             sh:Violation ;
                 sh:sourceConstraint           _:b3 ;
                 sh:sourceConstraintComponent  sh:SPARQLConstraintComponent ;
                 sh:sourceShape                kg:PublicSectorInstitutionTypeShape ;
                 sh:value                      99
               ] ;
  sh:result    [ rdf:type                      sh:ValidationResult ;
                 sh:focusNode                  <urn:ss:NOREF-EmployerDeclaration0> ;
                 sh:resultMessage              "Amount declared different from amount calculated" ;
                 sh:resultPath                 ont:NetOwedAmount ;
                 sh:resultSeverity             sh:Violation ;
                 sh:sourceConstraint           [] ;
                 sh:sourceConstraintComponent  sh:SPARQLConstraintComponent ;
                 sh:sourceShape                kg:NetOwedAmountShape ;
                 sh:value                      00000000009999
               ] ;
```

Figure 28: `invalid_dmfa.ttl` validation report

6 Discussion

The previous chapter outlined the steps taken to validate social security declarations using knowledge graph technologies, including the development of a vocabulary, mapping of data, and the creation of SHACL rules. In this chapter, we will reflect on the strengths and limitations of our approach.

6.1 Glossary

The glossaries of the declarations provided by NOSS served as the primary source of documentation to develop the validation process. They provided the rules that declarations must respect and the annexes describing additional information. However, some issues were encountered during the development.

First, not all the rules were well-documented and thus some additional research was necessary to find missing information. For example, the glossary states that some numbers must respect a checksum but does not state how the checksum is computed. The specifications of some checksums were found on the social security website, but one had to be found on a website unrelated to social security. Finding how to compute the total amount of contribution a company owes to the NOSS was even harder as no documentation specifying the computation was found. We knew that the Java validator (cf. Section 3.4) checks its validity and resorted to analyze the source code the validator to understand how to express this rule. While the SHACL shapes we developed formalize these rules, we could foresee a knowledge management approach to complement those with glosses and even pointers to the various definitions (e.g., using the `rdfs:isDefinedBy` predicate).

Moreover, some rules were unclear. As we are not domain experts, these rules expressed with domain specific knowledge could not be developed. For example, Annex 3 was not comprehensive as it used a nomenclature which was not specified. Thus, the rules based on this annex could not be expressed in SHACL. It even seems that there are some inconsistencies between the rules. For instance, a `ont:EmployerDeclaration` may have an unlimited number of `ont:CompanyVehicle`[5] but these instances must all have different `ont:CompanyVehicleSequenceNbr` in the `[1;99999]` interval which restricts the amount of allowed `ont:CompanyVehicle`.

Modifying the glossaries could solve these problems but our approach already provides some solutions. The SHACL rules that were developed provide a more complete expression of the rules that a declaration must respect as they centralized information from multiple sources. SHACL also provides a formalism that expresses the rules in a clear and precise manner that can capture the domain specific knowledge to be understood to anyone with some training in SHACL and SPARQL.

[5]To be precise, each ont:EmployerDeclaration can be linked to an unlimited number of ont:CompanyVechicle instances through the `ont:R_90007_90294` predicate.

6.2 Vocabulary

The creation of a vocabulary from the glossaries served as the foundation for both mapping the data and developing SHACL shapes. Despite the fact that this vocabulary was sufficiently to semantically annotation of data in a DmfA, there were some limitations that could hinder the semantic interpretability between the different information systems.

The first limitation concerns the language used for the various annotation properties. For most of these properties, only a French string was included in the vocabulary. However, Belgium has three official languages (French, Dutch, and German) thus the people managing the information systems involved in the DmfA most likely do not understand all of them. Although a French and Dutch version of the document used to generate the vocabulary exists, only the French one was considered as we are not sufficiently proficient in Dutch to assess the quality of the translations. Nevertheless, the scripts used to generate the vocabulary can easily be extended to incorporate the Dutch version as it follows the same structure but requires to be validated by a domain expert. The same process applies for any future German version. Note that this limitation pertains to human understanding of the vocabulary. For computer agents, those human-readable labels are not important as long as the domain is formally captured in an adequate way.

Secondly, there are a couple of "semantic incoherences" in the vocabulary. An ontology is supposed to approximate a domain. We have generated a vocabulary from an XML schema which annotates data. The vocabulary can thus be considered more "data-centric" rather than "true to the domain". These incoherences pertain to the object properties (i.e., relationships). For example, one could say from the information in the vocabulary: "An employer declaration has a company vehicle." or "An occupation receives a remuneration.". These statements would be more semantically correct if they said, "An employer declares an employee having a company vehicle." and "A natural person receives a remuneration for an occupation." However, to achieve the latter statements, one would not only require modifying the relationships but also the entities that relationship links. I.e., the vocabulary had to be overhauled. If we were to do this, we could potentially include additional semantic errors as we had no access to domain experts. Since trying to fix the semantic incoherences was too complex, they were left out and should be understood as something linking two entities. This, however, had no impact on the study. As we wanted to validate the data contained in employer declarations, it did not really matter whether there was a direct link between a company vehicle and an employee or indirectly via employer declaration.

Another limitation concerns the naming we chose for these object properties. As many relationships were shared between entities (e.g., `has`), we have chosen to give opaque IRIs for those relationships (e.g., `ont:R_90007_90294`) to make them distinct and avoid giving them names The vocabulary could be refined by renaming these predicates with the help of domain experts. This bore little impact as the core of the

thesis is data validation and only certain predicates in the mappings and SHACL rules would need to be replaced.

6.3 Mappings

Transforming data from social security declarations and annexes into RDF necessitated the creation of mappings. The annexes were available in both CSV and XML formats. The CSV data could have been processed as relational data by an R2RML processor. However, the declarations were only available in XML, requiring a more expressive language such as RML. To maintain consistency across the entire process, we chose to use RML for mapping all data sources. We chose RMLMapper as the RML processor as it was the reference implementation and the most accurate processor according to the RML implementation report [28].

Nevertheless, the expressivity of RML presented some limitations. One noticeable limitation was the loss of information during the mapping of XML[6] files (cf. Section 5.2.1.1). The problem arises from the way RML uses XPath to represent XML elements as logical tables; these tables lack primary and foreign keys that would link an element to its sub-elements. This representation results in a loss of the hierarchical information found in the tree structure of XML.

Another challenge we faced was data transformations. They could not be expressed with RML apart from transformation with an SQL query for RDB data. Thankfully, RMLMapper supported the Function Ontology (FnO), which provides the capability for expressing data transformations (cf. Section 5.3.2). However, these transformations can be verbose, making them hard to develop and maintain. This verbosity issue might be mitigated by developing better tools for writing and managing FnO transformations, improving the accessibility and usability of RML.

Alternatively, data transformation could be performed using SPARQL, particularly with the help of extensions like SPARQL-Generate [29]. SPARQL-Generate allows for data transformations to be expressed using SPARQL functions and operators. This approach provides a more straightforward and less verbose way of handling data transformation. The downside of SPARQL-Generate, however, is that the mappings are not encoded as RDF and can, therefore, not be part of the knowledge graph. Another advantage of RML over SPARQL-Generate is that it supports data governance tasks such as data lineage; one can inquire where data comes from.

Comparatively, an R2RML processor like R2RML-F [30] seems to offer capabilities that could have mitigated some of these limitations. It provides an iteration counter for CSV files, which could have helped with primary keys. Additionally, R2RML-F encapsulates ECMAScript functions within the mapping process, offering a less verbose way to perform data transformations compared to FnO.

Reflecting upon this, it seems that the RMLMapper implementation of RML may not be fully mature. However, the challenges we encountered were simple to over-

[6] Although it was not shown in this thesis, a similar observation can be made for JSON files.

come with some Python scripts. Moreover, the ability to map data from a wider range of sources, such as XML, CSV, and JSON, is a significant upside. This capability is particularly important in the context of the Belgian social security, where multiple companies and NOSS institutions are involved.

6.4 SHACL

The main goal of this thesis was to assess the capabilities of SHACL in comparison to XSD for data validation tasks. We aimed to determine the extent to which SHACL is more expressive than XSD, its scalability for handling validation tasks, the difficulty of expressing shapes, and their maintainability.

We have shown in our study that SHACL is more expressive than XSD. All the XSD constraints that we encountered had an equivalent representation in SHACL-core. Moreover, SHACL enables the creation of more complex rules through the use of logical operators (part of SHACL-core) or SHACL-SPARQL constraints. Despite being able to express all the most complex rules for a DmfA with SHACL-SPARQL constraints, this achievement prompts us to question the expressivity of SPARQL itself. While the shapes can be used to validate individual declarations, we have yet to explore the validation of declarations against previously submitted ones and/or external information such as information for other social security institutions, which would require access to a knowledge graph incorporating this extra information. However, the declaration's validation does incorporate information from annexes, demonstrating the feasibility of validating declarations using external data sources.

Interoperability is a notable advantage of SHACL. The SHACL shapes we have developed are part of the knowledge graph, expressed in a machine-readable format rather than natural language or bespoke code. SHACL processors exist for different software ecosystems, such as Python, Java, .NET, Scala, and others. However, it is important to note that the maturity of these processors may vary. During our investigation, we encountered certain bugs in the Python processor, pyshacl [31], which resulted in invalid reports for valid declarations. Although we identified patterns within the shapes, they were expressed in a domain-specific manner. Future work could explore the use of SPARQL-based constraint components to achieve higher levels of abstraction and develop more generalized and widely applicable shapes.

Building effective SHACL rules was a complex task as it involved considering three important factors: performance, maintainability, and reusability. In some cases, it was possible to improve one factor without compromising the others. For example, using integer operations instead of string operations for checksum calculations can enhance performance. However, there was often a trade-off among these factors. For instance, rewriting a rule from the perspective of a related entity rather than strictly adhering to glossary specifications, prioritizes performance over maintainability. Similarly, favoring logical operators over complex SPARQL rules enhances maintainability and reusability at the expense of potential performance gains. This

shows that developing effective SHACL rules requires profound knowledge in RDF, SHACL, SPARQL, and computer science principles.

Another challenge was the organization of the rules in order to validate the multiple declarations and their multiple versions.

To deal with the multiplicity of the declarations, our solution was separate shapes into common shapes that apply to any declaration and specific shapes that apply to a specific declaration. While this separation was relatively simple as we only created shapes for two declarations, we hypothesize dealing with more declarations may increase the complexity of the separation, but it may be mitigated by domain expertise.

Moreover, we created near copies of the shapes graph for a declaration to manage the time multiplicity. This approach created a lot of repetitions which can make corrections harder as multiple shapes must be changed. However, it made the management of the shapes graph easier as there is a version for each quarter. We also explored another solution with its own sets of problems. The idea was to use deltas to encompass the differences between versions such as Git, but it would create a computation overhead to retrieve shapes graph and modifications in the "middle" of the history may be harder.

By organizing the shapes into named graphs based on the declaration and the corresponding quarter, we achieved a simpler retrieval of a particular shapes graph. The query to retrieve a particular shapes graph retrieved all the triples from the common and specific named shapes graph for a quarter. Comparatively, storing all the shapes into a default graph required additional annotations to differentiate their domain of applicability, and we found no significant benefits to store the shapes in this manner.

In summary, this discussion emphasizes the advantages of using SHACL as a more expressive and interoperable language for data validation. However, it also highlights that building effective rules with SHACL is a complex engineering task. It requires in-depth knowledge, proper tools, and a meticulous approach to successfully design and implement SHACL rules.

7 Conclusions

7.1 Summary and Achievements

Throughout this thesis, we demonstrated the feasibility and potential of using knowledge graph technologies, specifically SHACL, for validating social security declarations. We embarked on creating a prototype knowledge graph and examining how SHACL performs in validating complex constraints.

In the early stages, we crafted a vocabulary for our knowledge graph, informed by glossaries of the declarations provided by NOSS. The initial challenge of incomplete or unclear rules in the glossaries was addressed by using SHACL in later stages. Incorporating the SHACL rules into our knowledge graph provided a complete, precise, and centralized expression of the validation rules. This achievement underscores the capability of our approach to represent complex rules expressed in natural language (and jargon). We even formalized rules for which the documentation lacked clarity.

Moving forward, we embarked on the development of a mapping process that transformed DmfA declarations (stored as XML documents) and annexes (stored as CSV documents) into RDF. Despite encountering challenges with RML due to the loss of structural information of data sources and verbose data transformation, we successfully mapped data from diverse sources using RML and simple Python scripts. Being able to map from various sources is particularly significant considering the Belgian social security context, which involves multiple companies and institutions, each potentially providing data in different formats.

At the core of our work, we successfully developed SHACL shapes and a process for validating DmfA XML files. We have identified some patterns to build rules that may be used in a context beyond social security. Moreover, we showed that building effective SHACL rules was a complex engineering task that requires a profound understanding of RDF, SHACL and SPARQL in order to balance performance, maintainability, and reusability.

These developed rules highlight SHACL's expressiveness, showing that it is more expressive than XSD constraints. All the encountered XSD constraints in our study had an equivalent SHACL constraint and one can even express more complex rules through logical operators or SHACL-SPARQL constraints. Thus, a more complete DmfA data validation can be achieved by sharing them with employers.

Furthermore, the SHACL shapes are also more interoperable because they are expressed in a machine-readable format rather than natural language or bespoke code. They are also integrated within the knowledge graph and SHACL processors exist for different software ecosystems.

Reflecting on the work, our achievements showcase the applicability of knowledge graph technologies for data validation in the social security domain. Through care-

ful evaluation, we have identified SHACL as a capable tool for addressing complex validation tasks, paving the way for future work in this area. On a personal note, I am immensely proud that this work has gained recognition in the academic community. It was accepted as a presentation based on a peer-reviewed non-published abstract at the ENDORSE2023 conference and subsequently invited for a paper for the ENDORSE2023 post-proceedings.

7.2 Further Improvements

During the development of this project, we identified several ideas and challenges which can improve the developed system. We discuss a few of them in this section.

Despite the existence of some proprietary software integrating various knowledge graph tools, we opted for open-source tools only to make our project reproducible. Developing our solution with these non-integrated tools required substantial manual intervention, indicating the need to create and improve open-source knowledge graph tools in an integrated manner. For example, integrating an editor with a SHACL processor and a triple-store into a single software could simplfy shapes development. One could load the data and shapes graph in the triple-store, the editor could allow testing a single shape at a time and show the validation report. It could also provide some syntactic sugar for constraints such as sh:exactCount for sh:minCount and sh:maxCount.

Similarly to Debruyne and McGlinn in [32], the patterns that were identified in this project could be published according to Linked Data [33] principles in order to share and make them reusable. As these patterns were domain-specific, they needed to be rewritten with a higher level of abstraction. This could probably be achieved with SHACL-SPARQL constraint components.

On another note, extending our approach to validate all types of declarations over time presents an interesting challenge. We have shown that we could provide a validation report for a declaration in isolation, but there may be some different rules that span across different quarters and declarations. Assessing whether our knowledge organization strategy can adequately or efficiently support such validation processes would require further study and more substantial experiments.

Finally, a limitation was the scarcity of declaration examples that were available online. The declarations found were simple, did not exemplify all possible fields, and were, unfortunately, invalid (despite conforming to the XSD). Thus, testing with authentic data may yield more realistic observations. However, given the sensitive nature of the data, this would necessitate collaborating with the ONSS.

References

[1] A. Hogan, E. Blomqvist, M. Cochez, C. d'Amato, G. de Melo, C. Gutierrez, J. E. L. Gayo, S. Kirrane, S. Neumaier, A. Polleres, R. Navigli, A. N. Ngomo, S. M. Rashid, A. Rula, L. Schmelzeisen, J. F. Sequeda, S. Staab, and A. Zimmermann, "Knowledge graphs," CoRR, vol. abs/2003.02320, 2020. [Online]. Available: https://arxiv.org/abs/2003.02320

[2] N. Noy, Y. Gao, A. Jain, A. Narayanan, A. Patterson, and J. Taylor, "Industry-scale knowledge graphs: Lessons and challenges," Communications of the ACM, vol. 62 (8), pp. 36–43, 2019. [Online]. Available: https://cacm.acm.org/magazines/2019/8/238342-industry-scale-knowledge-graphs/fulltext

[3] R. Studer, V. Benjamins, and D. Fensel, "Knowledge engineering: Principles and methods," Data & knowledge engineering, vol. 25, no. 1, pp. 161–197, 1998.

[4] D. Wood, M. Lanthaler, and R. Cyganiak. (2014, Feb.) RDF 1.1 concepts and abstract syntax. [Online]. Available: https://www.w3.org/TR/rdf11-concepts/

[5] G. Carothers and E. Prud'hommeaux. (2014, Feb.) RDF 1.1 turtle. [Online]. Available: https://www.w3.org/TR/turtle/

[6] R. Guha and D. Brickley. (2014, Feb.) RDF schema 1.1. [Online]. Available: https://www.w3.org/TR/rdf-schema/

[7] B. Motik, P. F. Patel-Schneider, and B. C. Grau. (2012, Dec.) Owl 2 web ontology language: Direct semantics (second edition). [Online]. Available: http://www.w3.org/TR/owl2-direct-semantics/

[8] G. Carothers. (2014, Feb.) RDF 1.1 N-Quads. [Online]. Available: https://www.w3.org/TR/n-quads/

[9] M. Arenas, A. Bertails, E. Prud'hommeaux, and J. Sequeda. (2012, Sep.) A Direct Mapping of Relational Data to RDF. [Online]. Available: https://www.w3.org/TR/rdb-direct-mapping/

[10] S. Das, S. Sundara, and R. Cyganiak. (2010, October) R2RML: RDB to RDF Mapping Language. [Online]. Available: http://www.w3.org/TR/r2rml/

[11] A. Dimou, M. Vander Sande, B. De Meester, P. Heyvaert, and T. Delva. (2022, Nov.) RDF Mapping Language (RML) Unofficial Draft. [Online]. Available: https://rml.io/specs/rml/

[12] (2013, Mar.) SPARQL 1.1 Query Language. [Online]. Available: http://www.w3.org/TR/sparql11-query

[13] L. Feigenbaum and E. Prud'hommeaux, "Sparql by example: The cheat sheet," Slides, June 2009. [Online]. Available: https://www.iro.umontreal.ca/~lapalme/ift6281/sparql-1_1-cheat-sheet.pdf

[14] (2017, Jul.) Shapes constraint language (SHACL). [Online]. Available: https://www.w3.org/TR/shacl/

[15] FPS Social Security, "Everything you always wanted to know about social security," Brochure, 2022. [Online]. Available: https://socialsecurity.belgium.be/sites/default/files/content/docs/en/publications/everything-social-security-en.pdf

[16] Smals, "Untitled [dmfa erd]," Retrieved from PDF, 2022. [Online]. Available: https://www.socialsecurity.be/portail/glossaires/dmfa.nsf/ConsultFrAModifIntro/FC4377BDE6D228F0C125885C002B91C6/\protect\T1\textdollarFILE/DmfAOriginal_Introduction_FR.pdf

[17] ——, "LA DÉCLARATION TRIMESTRIELLE ONSS (DmfA) - GLOSSAIRE," 2022. [Online]. Available: https://www.socialsecurity.be/lambda/portail/glossaires/dmfa.nsf/web/glossary_home_fr

[18] Publications Office of the European Union, "Application profiles," https://op.europa.eu/en/web/eu-vocabularies/application-profiles, 2023.

[19] ——, "The official portal for European data," https://data.europa.eu/en, 2023.

[20] SEMIC action team, "Dcat application profile for data portals in europe," https://ec.europa.eu/isa2/solutions/dcat-application-profile-data-portals-europe_en/, 2020.

[21] Joinup, "Dcat-ap 2.0.0," https://joinup.ec.europa.eu/collection/semantic-interoperability-community-semic/solution/dcat-application-profile-data-portals-europe/release/200, 2020.

[22] Digitaal Vlaanderen, "Open standaarden voor linkende organisaties," https://data.vlaanderen.be/, 2023.

[23] RDF Mapping Language (RML), "Rmlmapper," https://github.com/RMLio/rmlmapper-java, 2022.

[24] The Apache Software Foundation, "Jena Fuseki," 2023. [Online]. Available: https://jena.apache.org/download/#apache-jena-binary-distributions

[25] A. Dimou, B. De Meester, and F. Kleedorfer. (2021, Nov.) The Function Ontology Unofficial Draft. [Online]. Available: https://fno.io/spec/

[26] T. Berners-Lee. (1998) Cool URIs don't change. [Online]. Available: http://www.w3.org/Provider/Style/URI

[27] TopQuadrant, "TopBraid SHACL API," https://github.com/TopQuadrant/shacl, 2022.

[28] P. Heyvaert, A. Dimou, and D. Chaves-Fraga. (2022, Feb.) RML Implementation Report Unofficial Draft. [Online]. Available: https://rml.io/implementation-report/

[29] M. Lefrançois, A. Zimmermann, and N. Bakerally, "A SPARQL extension for generating RDF from heterogeneous formats," in Proc. Extended Semantic Web Conference (ESWC'17), Portoroz, Slovenia, May 2017. [Online]. Available: http://www.maxime-lefrancois.info/docs/LefrancoisZimmermannBakerally-ESWC2017-Generate.pdf

[30] C. Debruyne and D. O'Sullivan, "R2rml-f: towards sharing and executing domain logic in r2rml mappings," in LDOW@ WWW, 2016.

[31] A. Sommer, N. Car, and J. Yu, "pySHACL," https://github.com/RDFLib/pySHACL, 2023.

[32] C. Debruyne and K. McGlinn, "Reusable shacl constraint components for validating geospatial linked data." CEUR, 2021.

[33] C. Bizer, T. Heath, and T. Berners-Lee, "Linked data: The story so far," International Journal on Semantic Web and Information Systems, vol. 5, pp. 1–22, 07 2009.

A CONSTRUCT query for mappings with reference and file naming

```
PREFIX rml: <http://semweb.mmlab.be/ns/rml#>

CONSTRUCT {
  ?s ?p ?o.
  ?LogicalSource rml:source "filename".
  ?sm rr:template ?newtemplate
}
FROM <http://kg.socialsecurity.be/mappings/dmfaxml/>
WHERE {
  {
    ?LogicalSource rml:source ?source.
  }
  UNION
  {
    ?tm rr:subjectMap ?sm.
    ?sm rr:template ?template.
    BIND ( REPLACE(?template, "NOREF", "ref") as ?newtemplate)
  }
  UNION
  {
    ?s ?p ?o.
    FILTER (?p != rml:source)
    MINUS {{
      ?tm rr:subjectMap ?s.
      ?s rr:template ?o.
    }
  }
}
```

Listing 13: CONSTRUCT query for DmfA to RDF mappings with reference and file naming

B Benchmark

B.1 Hardware

Table 3: Hardware configuration

CPU	Intel(R) Core(TM) i7-10700KF
RAM	Corsair Vengeance(R) LPX (2 x 8GB) DDR4 DRAM 3200MHz
Motherboard	MSI Z590 PRO WIFI

B.2 Storage strategies

Table 4: Storage strategies experiment setup

	Default graph strategy	Named graph strategy
Endpoint	Apache Jena Fuseki	
Number of queries	100	
Query	`CONSTRUCT{ ?s ?p ?o. } WHERE{ <urn:ss:000-DmfAOriginal0> (<>|!<>)* ?s . ?s ?p ?o . }`	`CONSTRUCT{ ?s ?p ?o. } FROM <urn:ss:000> WHERE{ ?s ?p ?o . }`
Dataset size	5862000 triples	5862000 quadruples

Table 5: Storage strategies experiment results

Response time (ms)	Default graph strategy	Named graph strategy
Mean	43.6522	36.5589
Standard deviation	2.9888	1.8716

B.3 SPARQL-based constraint optimization

B.3.1 Checksum

Table 6: Checksum experiment setup

	Integer version	Substring version
Endpoint	TopBraid SHACL Processor	
Number of runs	100	
Shape's select	Listing [14]	Listing [15]
Number of triples to verify	5000	

```
SELECT $this ?value
WHERE {
    $this $PATH ?value .
    BIND( FLOOR(?value / 100) AS ?number )
    BIND( ?value - (100 * ?number) AS ?check )
    BIND( ?number - (97 * FLOOR(?number / 97)) AS ?rest )
    BIND( 97 - ?rest AS ?check2 )
    FILTER ( ?check != ?check2 )
}
```

Listing 14: Integer version of SELECT query for checksum shape

```
SELECT $this ?value
WHERE {
    $this $PATH ?value .
    BIND( STR(?value) as ?stringvalue )
    BIND( xs:integer(SUBSTR(?stringvalue, 1, STRLEN(?stringvalue) - 2)) AS ?number )
    BIND( xs:integer(SUBSTR(?stringvalue, STRLEN(?stringvalue) - 1)) AS ?check )
    BIND( ?number - (97 * FLOOR(?number / 97)) AS ?rest )
    BIND( 97 - ?rest AS ?check2 )
    FILTER ( ?check != ?check2 )
}
```

Listing 15: Substring version of SELECT query for checksum shape

Table 7: Checksum experiment results

Response time (s)	Integer version	Substring version
Mean	2.8994	3.1042
Standard deviation	0.0905	0.1348

B.3.2 Code Existence

Table 8: Code existence experiment setup

	Bound checking version	Counting version
Endpoint	TopBraid SHACL Processor	
Number of runs	100	
Shape's select	Listing [16]	Listing [17]
Number of triples to verify	5000	

```
SELECT $this ?value
WHERE {
    $this $PATH ?value.
    OPTIONAL{
        ?awr a an6:ActivityWithRisk;
            an6:Code ?value;
        .
    }
    FILTER(!BOUND(?awr))
}
```

Listing 16: Bound checking version of SELECT query for code existence shape

```
SELECT $this ?value
WHERE {
    {
        SELECT $this ?value (SUM (?match) as ?nbrMatch)
        WHERE {
            $this $PATH ?value.
            ?awr a an6:ActivityWithRisk;
                an6:Code ?allowedvalue;
            .
            BIND (IF (?value = ?allowedvalue, 1, 0 ) AS ?match)
        }
        GROUP BY ?value $this
    }
    FILTER(?nbrMatch = 0)
}
```

Listing 17: Counting version of SELECT query for code existence shape

Table 9: Code existence experiment results

Response time (s)	Bound checking version	Substring version
Mean	3.2291	3.5601
Standard deviation	0.1181	0.1164

B.4 Target selection

Table 10: Target selection experiment setup

	Parent target	Child target
Endpoint	TopBraid SHACL Processor	
Number of runs	100	
Shape's select	Listing [18]	Listing [19]

```
SELECT $this
WHERE {
    {
        SELECT $this (COUNT(?seqNbr) as ?seqNbrOcc)
        WHERE {
            $this ont:R_90007_90017/ont:NaturalPersonSequenceNbr ?seqNbr .
        }
        GROUP BY ?seqNbr $this
    }
    FILTER(?seqNbrOcc > 1)
}
```

Listing 18: Parent target SELECT query for unicity shape

```
SELECT DISTINCT $this ?value
WHERE {
    $this ont:NaturalPersonSequenceNbr ?value .
    $this ^ont:R_90007_90017 ?employerDeclaration .
    ?employerDeclaration ont:R_90007_90017 ?other .
    ?other ont:NaturalPersonSequenceNbr ?valueOther .
    FILTER ($this != ?other && ?value = ?valueOther)
}
```

Listing 19: Child target SELECT query for unicity shape

Table 11: Unicity Shape with parent target experiment results

Response time (s)	Number of natural persons				
	1000	2000	3000	4000	5000
Mean	2.3119	2.3802	2.4456	2.5146	2.5520
Standard deviation	0.0599	0.0559	0.0725	0.0560	0.0822

Table 12: Unicity shape with child target experiment results

Response time (s)	Number of natural persons				
	1000	2000	3000	4000	5000
Mean	3.8965	6.9950	12.9321	20.5664	31.5331
Standard deviation	0.2048	0.3275	0.3498	0.6771	1.2384