INSTITUTE OF COMPUTER SCIENCE
Department of Computer Science and Mathematics

Master Thesis

Modeling Nomisma ontology and Comparing Solutions for Uncertainty

Presented by
Ram Sabah & Zeena Sabah

Submitted to
Dr. Karsten Tolle
Big Data Lab Director

Prof. Dr. Hendrik Drachsler
Scientific Director of studiumdigitale at Goethe-University

Frankfurt, June 07, 2022

Erklärung/Declaration

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

I hereby declare that I have created this work completely on my own and used no other sources or tools than the ones listed.

Ram Sabah

Frankfurt, June 07, 2022 Full Name

Erklärung/Declaration

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

I hereby declare that I have created this work completely on my own and used no other sources or tools than the ones listed.

Zeena Sabah

Frankfurt, June 07, 2022 Full Name

Acknowledgment

At this point, we would first like to thank Dr. Karsten Tolle for his support in writing our master thesis.
Thank you so much for always having an open ear for our questions and for guiding us in taking the right steps through the entire process.

We would like to thank Dr. David Wigg-Wolf for his support in creating the Nomisma-Cookbook and for his feedback, thank you very much.

Many thanks to Dr. Martin Doerr for responding to our inquiry regarding the uncertainty solution proposals issued in CIDOC-CRM.ORG.

We would also like to thank our second examiner Prof. Dr. Hendrik Drachsler for the time and effort, thank you very much.

Finally, a debt of gratitude to our mother Nawal and our brother Sam for all their support and love, for encouraging us to give our best and move forward. Without you, none of this would have been possible.

Abstract

This thesis deals with the modeling of the Nomisma-Ontology, the modeling and evaluation of solutions for handling uncertainties in the field of numismatics.

Ancient coins have archaeological, cultural, and historical importance. The publication and linking of knowledge about these coins offers an enriched understanding of cultural heritage. The Nomisma.org is a collaborative project that provides digital representations of numismatic concepts according to Linked Open Data principles. The modeling use cases of the Nomisma-Ontology introduced in this thesis are recommended practice examples, following which will promote better interoperability. This thesis also presents two suggested approaches for modeling the layers of a hoard vessel, one of which uses Nomisma-properties, and the other uses the RDF-Collection.

A large number of ancient coins are strongly damaged what make their identification a difficult task. In some situations, the archaeologist is uncertain about specific features of a coin - for example, it is uncertain whether Coin_x shows a picture of Titus or of Nero, it is uncertain that Coin_y was minted in the ancient city Comama. Although this information is unconfirmed, it is still very valuable and need to be published and made distinguishable from certain information. This thesis describes and evaluates approaches/solutions for dealing with such uncertainties. To evaluate the solutions, we run SPARQL queries on the data modeled according to each solution and measure the runtime and memory consumption of the queries. Overall, the solutions that requires minimal number of triples to express uncertainty has less query runtime. Memory consumption depends on the size of the queried data.

In order to the uncertainty solutions on large datasets of 100K- to 1M-coins, we created a web application, implemented the proposed solutions within it and used it to generate an RDF/XML (or Turtle) file from a given Excel/CSV file based on the selected solution. This web application, called SAUN, can also visualize the generated RDF/XML file as an RDF-graph as well as display and search all triples contained in the generated file.

Keywords: Nomisma.org, data modeling, handling uncertainty, RDF/XML, Turtle, SPARQL.

Contents

1. Introduction

Ancient coins carry rich information of cultural and historical events, they are very important archeological items. In this thesis, we will present modeling use cases of the Nomisma.org ontology, which provides digital representations of numismatic concepts based on the concepts of Linked Open Data and which can serve as a paradigm for dealing with digital corpora in archaeology (Wigg-Wolf, Tolle, & Kissinger, 2019). As well as modeling and evaluating solutions for dealing with uncertainties in the domain of numismatics.

This chapter aims to provide a motivation for these two main goals of this work and what is covered in each chapter.

1.1 Motivation

Numismatics the study or collection of currency, including coins and related objects is a very important research topic. Coins, perhaps in particular, carry rich and accurate information about cultural heritage and about the events that occurred during their time in circulation (Kemmers & Myrberg , 2011). They are identified by their physical characteristics such as size, shape, weight, material, etc. By appearance like marks, inscriptions, dates, and depiction, which usually refer to the ruling authority, to deity, and to events (Wickens). These features and more are important in identifying, classifying, and studying these artifacts, and the publication and linking of knowledge about them need to promote interoperability between the different hosting institutions. The need for unified collection, management and exchange of such knowledge was the motive to adopt the semantic web technologies for creating ontologies and providing a common digital representation of numismatic concepts (Moraitoua, Christodouloua, & Caridakisa, 2021).

Nomisma.org (Ackermann, et al.) is a collaborative project that provides pre-defined digital representation of numismatic concepts based on the linked open data principles. Though this project is still under implementation and is subject to constant expansion and correction, it provides up to this day a rich vocabulary in the domain of numismatics and over 7000 keywords in different numismatic concepts. Objects like coins when linked to Nomisma.org are displayed at resources like as Pleiades and Pelagios (Pelagios Network)

what will enrich their information with data from other open resources (Wigg-Wolf, Tolle, & Kissinger, 2019). This makes the data more visible and more available for research projects in other disciplines.

In this thesis we introduce some modeling use cases of the Nomisma-Ontology as recommended practical examples, following which will promote better interoperability. Alongside the Nomisma classes and properties, we will use, when needed, some terms from different ontologies like CIDOC-CRM and its extensions, Dublin-Core metadata terms, Skos, etc. to give examples.

Though this may sound smooth and straightforward, coins are not always perfectly preserved and have all their features clear and visible. Due to corrosion or wear occurring as a result of using thiese coins or due to improper preservation, they can be very difficult to identify even for experts. In such cases, numismatists often find themselves uncertain of the correctness in distinguishing the features of these coins. For instance, *it is uncertain whether Coin_x shows a portrait of Titus or of Niro*, *it is uncertain that Coin_x was minted in Comama*, *the production date of Coin_y is uncertain*, and many more of such examples of uncertainty. Uncertainty occurs not only because of the bad state of the object to be described, but also because of other reasons, types of uncertainty (Laskey, et al., 2008).

> *"Archaeological information is by its very nature complex and uncertain."*
>
> *(Cripps, Paul J., 2012, S. 487)*

There is not yet a standard method for modeling uncertainties, it is an issue still open for discussion and suggestive solutions. A goal of this thesis is to investigate solution suggestions for modeling uncertainties in the domain of numismatics. This work (Uncertainty Handling for Ancient Coinage., 2014) presented two modeling solutions and in dissections in communities under (CIDOC CRM, 2014) we found three more solution suggestions for modeling uncertainty. Based on our understanding of the situation, we also propose three other solutions, two of which based on the first two solutions proposed in (Uncertainty Handling for Ancient Coinage., 2014).

To test and compare these solution suggestions on datasets of different sizes, by measuring runtime and memory consumption when querying data, we created a web application that generates an RDF/XML (or Turtle) file from a given Excel or CSV file and a selected uncertainty modeling solution, that we implemented in the application.

Questions and Objectives

- **How to use the Nomisma-Ontology?**
  - Objective is to give modeling use cases of the Nomisma-Ontology and create the Nomisma-Cookbook.
- **How to model the layers of hoard vessel?**
  - Objective is to suggest solutions for modeling the layers within a hoard vessel.
- **How to model uncertainties in the domain of numismatics?**
  - Objective is to investigate, compare and evaluate the solution proposals exists for modeling uncertainties.

1.2 Structure of the Thesis

Chapter 2 describes the theoretical foundations to this thesis and is of two parts. First part deals with the semantic web, its technologies, and standards. second part gives an overview of archaeological artifacts with a focus on ancient coins and hoards.

Chapter 3 consists of two sections, the first presents few modeling use cases of the Nomisma.org-Ontology and the second describes two approaches for modeling the layers of a hoard vessel.

Chapter 4 presents eight approaches for modeling the uncertainties, and a short overview of RDF-Star.

Chapter 5 is about creating a web application and implementing the eight uncertainty modeling solutions within it to generate an RDF/XML file from the input data, which is in Excel or CSV.

Chapter 6 discusses the process and results of measuring the runtime- and memory consumption- of querying data modelled according to the uncertainty solutions.

2. Theoretical Foundations

This chapter deals with the theoretical foundations of the topics covered in this thesis. Beginning with a brief introduction to the semantic web with, its technologies and standards, then an overview of archaeological artifacts with a focus on ancient coins and hoards.

2.1 Semantic Web and Ontology Languages

Before we try to understand the semantic web, we must first define an important concept which is *"Data Integration on the web"*. Data integration on the web (Yu, 2011) is the process of collecting and merging data from multiple recourses and presenting it in a way useful to users. So, when we search for something on the web (may it be a person, an object, an event) intending to collect all the information possible about it, we may have to manually go through multiple websites to find what's needed (we self must integrate those information). However, if the thing we are looking for was presented on the web in a way understandable for our computers, they will be able to do the data integration for us retrieving whatever we want to know.

To make this possible, data on the web should be in the form of statements (Berners-Lee, 2006), i.e., to a resource (subject) there is a value (object) connected to it as a relationship property (predicate). Each statement represents an aspect, a piece of knowledge about the given resource. These statements need to be modeled in a way understandable for a software program (an agent) and represented in the same structure on the web. The agent should be able to collect these statements in their given structure. To make such data integration possible for the agent some requirements must be met (Yu, 2011):

- Statements represent knowledge about a "thing" on the web and must be modeled in a machine-understandable and processable manner.
- Statements must have a common pattern, i.e., the model of the statements must be a standard acceptable by all websites.
- The statements must follow predefined common terms and relationships (ontology) depending on the entity they describe. For instance, a novel has title, an ISBN, an author, and a price, while a person has name, birthday, age, etc.

- These common terms and relationships must first be defined, and the different websites must use those terms and relationships when creating the statements.
- The common terms and relationships used to create the statements must be understandable for the agent so that it can understand the statements at all.
- The agent should be able to draw conclusions based on its understanding of the terms and relationships. To give an example, comparing the ISBN number from different sources, taking into account the knowledge expressed by the common terms and relationships, it should be possible to conclude that it relates to the same novel.
- The agent must be able to process queries on the statements.

The semantic web makes this possible and lead to creating such agent by providing the technologies and standards required.

The term *"Semantic Web"* was originally introduced by World Wide Web Consortium (W3C) director Sir Tim Berners-Lee:

"The Semantic Web is an extension of the current Web in which information is given well-defined meaning, better enabling computers and people to work in cooperation."

(Berners-Lee et al.2001).

Another definition of the semantic web is by *Liyang Yu* in his book *A Developer's Guide to the Semantic Web:*

"The semantic web can be understood as a brand-new layer built on top of the current Web, and it adds machine understandable meaning (of "semantics") to the current Web."

(Yu, 2011).

Now that we have learned the definition of the semantic web, we will next look at some of its technologies and standards.

2.1.1 Resource Description Framework RDF and RDF Schema

Since the web is filled with a huge amount of non-machine-understandable information, almost exclusively intended for human consumption, it is very difficult to automate anything on the web or deal with it manually (Yu, 2011). The W3C proposed a solution to this issue

which is to use metadata to describe the information contained on the web, because metadata is machine-readable, the automated processing of resources would be possible. With this, the Resource Description Framework (RDF) was introduced by the W3C in early 1999 as a standard for describing metadata and supporting interoperability between applications that exchange machine-understandable information on the web. Since the introduction of the Semantic Web in 2001, the RDF specification has changed a lot so that it is used to describe any piece of knowledge that exists in the real world (Yu, 2011).

With RDF specification, concept, schema, syntax and semantics, RDF can be described as the language and framework for representing information and expressing statements on the web using formal vocabularies. It should be flexible and scalable so that all information can be presented at any point, and it should provide a mechanism to connect the distributed information on the web.

An RDF data model is used to describe information about the world. The main idea of this model is to divide the information into small sections (pieces of knowledge, statements) with well-defined semantics following simple rules in order for machines to understand and process it (Yu, 2011):

- every piece of knowledge represented in a statement, also called a *"triple"*, must be in the form of subject-predicate-object, and this order is unchangeable. Subjects and objects describe resources in the real world. Predicates, also called *properties*, represents the relationship between subjects and objects.
- Names of resources and properties must be globally identified by e.g., a Uniform Resource Identifier (URI). While URL (Uniform Resource Locator) can identify and retrieve only resources that are directly retrievable on the Web like a webpage, a URI (Uniform Resource Identifier), a general form of identifier, is used to identify and retrieve anything on the web.
- Objects, also called *"property value"*, can hold either a literal or a resource. RDF literals are simple raw text data such as names, ages, dates, measurements, etc. Literals can optionally be language specified by giving a tag representing the language in which the raw text is written, for example *en* for English, *de* for German (Deutsch), *ar* for Arabic, etc. And can also be datatype specified by giving the type URI of the data represented in

the literal, for example *string* for text, *integer, decimal, double* for numbers, etc. The data type defined in ***XML-Schema*** is often used for these URIs, however any other data type URI can also be used. A literal holding the letter *A* with a language tag *en* is different from that holding the same letter *A* with datatype *string* and both are different from the one holding the letter *A* without language tag or datatype. Statements, each containing one of these literals as their object, are also completely different and cannot be inferred from one another (Yu, 2011).

- An RDF statement or a set of RDF statements can be represented as an RDF graph where subjects and objects are nodes in the graph and predicates (properties) are edges connecting a subject-node to an object-node.
- An RDF statement can only model a single piece of knowledge, a binary relationship between a given subject and object. To model multi-relationships, an intermediate resource known as ***Blank-node*** is used. With the use of blank nodes, we can model the so-called ***RDF-Reification***, a statement about a statement. A blank node has no URI as an identifier, instead it is assigned a local identifier so it can be referenced within an RDF document.

To summarize the above, an RDF graph data model represents statement in the form of subject, predicate, and object. The subject and object are represented as nodes in the graph and the predicate is the edge connecting them. Each set of subject, predicate and object is called an ***RDF-Triple***. Predicates are also called Properties and objects are property-values. Nodes (subjects and objects/property-values) can be URIs, literals, or blank nodes, and edges (predicates/properties) are URIs.

In Order to store and transmit the RDF graph model, an RDF serialization syntax is required. For this purpose, there are several serialization syntaxes like Notation 3, Turtle, N-Triples and RDF/XML. In this thesis we will focus on the RDF/XML serialization syntax.

2.1.2 RDF/XML Serialization Syntax

RDF/XML is a serialization syntax defined by the W3C specifications used to represent an RDF graph as an XML document (Yu, 2011). AS previously mentioned, RDF uses URIs as identifiers for nodes (subjects, objects) and edges (predicates). A set of these URIs are called

vocabularies (also called ontologies, will be discussed later) and composed of a namespace name (a leading string commonly shared in each set of URIs) followed by vocabulary names (syntax-, class-, property-, and resource names).

The URI namespace-name string is associated with a namespace-prefix (usually rdf: for RDF-Syntax, and rdfs: for RDF-Schema) in the namespace declaration part of an XML document. These namespaces are usually used within an XML document as qualified names (QNames), which contain a namespace prefix and a local part of the vocabulary name, separated by a single colon (example, rdf:subject, rdf:object). The RDF namespace prefix for RDF-Syntax shared by all RDF vocabulary names is:

```
http://www.w3.org/1999/02/22-rdf-syntax-ns#
```

With this vocabulary URI can be abbreviated, e.g., instead of writing *`http://www.w3.org/1999/02/22-rdf-syntax-ns#Description`* each time we want to describe an item within an XML document, we write *`rdf:Description`*. However, subjects, objects, and data type URIs cannot be abbreviated like this. Therefore, some RDF documents uses XML entities, that can associate a prefix with a namespace name (the shared string) as follows:

```
<!DOCTYPE rdf:RDF [<!ENTITY ex "http://www.example.org/example#">
```

Or, using the xml:base attribute along with rdf:ID for the subject, e.g., rdf:ID="item_1". The xml:base controls which base is used to fix the rdf:ID value. RDF will then generate the URI as follows: xml:base + "#" + item_1. To use the xml:base attribute with, for example rdf:about or rdf:resource, the "#" must be added to the name, e.g., rdf:about="#item_1". **List 2. 1** is an example of a simple RDF/XML document.

RDF-Vocabulary-Names:

The RDF Vocabulary consists of only the following names preceded by the namespace prefix (RDF 1.1 XML Syntax, 2014), (Yu, 2011):

Syntax names: rdf:RDF, rdf:Description, rdf:about, rdf:resource, rdf:ID, rdf:nodeID, rdf:datatype, rdf:parseType, rdf:li.

Class names: rdf:Statement, rdf:Property, rdf:XMLLiteral, rdf:List, rdf:Seq, rdf:Bag, rdf:Alt.

Property names: rdf:subject, rdf:predicate, rdf:object, rdf:type, rdf:value, rdf:first, rdf:rest_n.

Resource names: rdf:nil.

Example of an RDF/XML document describing that; Item_1 is of type coin with value 1d, represented in a graph in **Figure 2. 1** can be written as shown in **List 2. 1**.

List 2. 1: Example of using RDF/XML format to describe that item_1 is of type coin and has value 1d.

```
1. <?xml version="1.0"?> <!--XML version-->
2. <!--Using XML Entity-->
3. <!DOCTYPE rdf:RDF [<!ENTITY ex "http://example.org/example#">]>
4. <!--Namespace declaration-->
5. <rdf:RDF
6.         xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
7.         xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
8.         xml:base="http://www.example.org/coin_collection#">
9. <!--Item description-->
10.         <rdf:Description rdf:about="#item_1"> <!-or, rdf:ID="item_1"-->
11.            <rdf:type rdf:resource="&ex;coin"/>
12.            <rdf:value rdf:datatype="&ex;coin_value"> 1 d </rdf:value>
13.         </rdf:Description>
14. </rdf:RDF>
```

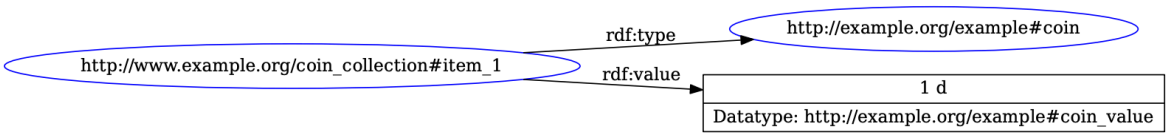


Figure 2. 1: RDF Graph the RDF/XML file describing that item_1 is of type coin and has value 1d.

Line 1 indicating the version of XML being used, lines 2,4 and 9 are comments inside of <!- - --> these brackets. Line 3 shows the use of an XML entity to associate the namespace string "*http://example.org/example#"* to a prefix "*ex*" so there is no need to rewrite the whole namespace name string every time we want to use it in the subject, object or datatype position

(lines 11 “&ex;coin” and line 12 “&ex;coin_value”). Line 4 creates an <rdf:RDF element, showing that this XML document is an RDF model, that ends at line 14 with </rdf:RDF>.

Lines 6 and 7 include namespace names declarations by using an xmlns attribute and the namespace prefix by which the namespace name is going to be referenced inside the document. In line 8 we used the xml:base attribute, this also to shorten the document by avoiding writing the whole namespace name string every time it is required (line 10 "#item_1"). We can skip adding a "#" to the rdf:about value here if we used rdf:ID instead of rdf:about, because when the RDF parser indicates the xml:base attribute, it will generate the URI by combining the namespace name string defined with xml:base with a "#" and then the value of rdf:ID. With rdf:Description in line 10 we say that these lines 10-13 (ending with </rdf:Description) are describing the item specified in rdf:about "item_1" which URI is “*http://example.org/example#item_1”* and that is of type coin with value 1d.

For a quick summary, RDF/XML is an RDF serialization syntax used to represent an RDF graph as an XML document. RDF uses URIs as identifiers for subjects, objects, and properties. A set of such URIs is called a vocabulary (an ontology) created for a specific reason/domain, i.e., a vocabulary is domain specific.

RDF provided a set of terms URIs (RDF vocabulary) to represent an RDF graph model as an XML document. In this section we have listed those terms and gave an example on how to use a few to describe that an item “item_1” is a “coin” and has value 1d. We also showed how to define namespaces and reference them within a subject, object, and datatype. So, in order to describe an item in the real world we must use classes and properties (vocabulary/ontology). To have a common language shared, processed, and understandable by distributed applications, there must be shared rules for creating these vocabularies/ontologies. In the next section "RDF Schema" we will learn how such a common vocabulary, with its classes, sub-classes, properties, and the relations between them are defined.

RDFS an RDF Schema

RDF Schema or RDFS is a W3C recommendation for describing RDF vocabulary. It provides language constructs (standard classes and properties) used to describe new classes and properties (a vocabulary) within an application domain.

"RDFS is a recommendation from W3C and it is an extensible knowledge representation language that one can use to create a vocabulary for describing classes, sub-classes and properties of RDF resources"

(Yu, 2011)

The latest version of RDFS was published in February 2004. Like RDF terms discussed earlier, RDFS terms are identified by pre-defined URIs, all share the following leading string:

```
http://www.w3.org/2000/01/rdf-schema#
```

which is usually associated with the namespace-prefix ‚rdfs', and that RDFS can be written in RDF/XML format as well as any vocabulary created by it. The following are RDFS groups of terms for defining classes and properties (Yu, 2011), (Brickley, Guha, & McBride, 2014):

Groupe of terms for defining classes: rdfs:Resource, rdfs:Class, rdfs:Literal, rdfs:Datatype.
Groupe of terms for defining properties: rdfs:subClassOf, rdfs:subPropertyOf, rdfs:range, rdfs:domain, rdfs:label, rdfs:comment.
Container classes and properties: rdfs:Container, rdfs: ContainerMembershipProperty, rdfs:member.
Utility properties: rdfs:seeAlso, rdfs:isDefinedBy.
The term rdfs:Resource represents the root class and every class defined using RDFS terms is a sub-class of rdfs:Resource. In other words, every item described in RDF is an instance of rdfs:Resource. The following
List 2. ***2*** is an example of defining classes and properties for the vocabulary "my_vocabulary", with graph illustrated in **Figure 2. 2**.

List 2. 2: Defining classes and properties for the vocabulary "my_vocabulary".

```
1.<?xml version="1.0"?> <!--XML version-->
2.<!--Using XML Entity-->
3.<!DOCTYPE rdf:RDF [
4.     <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">
5.     <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#">
6.     <!ENTITY myVocab "http://my-vocabulary.com/my-vocabulary#">]>
7.<!--Namespace declaration-->
8.<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
9.         xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
10.        xml:myVocab="http://my-vocabulary.de/my-vocabulary#">
11.<!--Classes definition-->
12.  <rdfs:Class rdf:about="&myVocab;Class_1">
13.    <rdfs:label xml:lang="en">Class_1</rdfs:label>
14.  </rdfs:Class>
15.  <rdfs:Class rdf:about="&myVocab;Class_2">
16.  </rdfs:Class>
17.  <rdfs:Class rdf:about="&myVocab;Class_3">
18.    <rdfs:subClassOf rdf:resource="&myVocab;Class_1"/>
19.  </rdfs:Class>
20.<!--Properties definition-->
21.  <rdf:Property rdf:about="&myVocab;property_1">
22.    <rdfs:label xml:lang="en">Property_1</rdfs:label>
23.    <rdfs:domain rdf:resource="&myVocab;Class_1"/>
24.    <rdfs:range rdf:resource="&myVocab;Class_2"/>
25.  </rdf:Property>
26.  <rdf:Property rdf:about="&myVocab;property_2">
27.    <rdfs:subPropertyOf rdf:resource="&myVocab;property_1"/>
28.    <rdfs:range rdf:resource="&myVocab;Class_3"/>
29.  </rdf:Property>
30.  <rdf:Property rdf:about="&myVocab;property_3">
31.  </rdf:Property>
32.</rdf:RDF>
```

Lines 1-10 specify the XML version and define the namespaces using XML entity and xmlns attribute. Lines 12-19 define three classes using rdfs:Class (Class_1 - Class_3) with Class_3 being a subclass of Class_1. Lines 21-31 define three properties (property_1 - property_3) using rdf:Property. Property_1 has domain Class_1 and range Class_2, meaning that, the subject used with this property must be is an instance of Class_1, and the object is an instance of Class_2. For example, may property_1 be "parent_of", Class_1 be

"Father" and Class_2 be "Son". With the domain restriction the subject must be a father and with the range restriction the property-value must be a son. However, these restrictions are optional, it is not mandatory that a property be domain and range restricted, e.g., property_3 (lines 30, 31). Property_2 (lines 26-29) is a sub-property of property_1 and it has a different range specification, namely Class_3. To give an example, may property_2 be "hasName" and Class_3 be "Name" we can represent that an instance of class "Father" hasName an instance of class "Name", or an instance of another class e.g., a superclass class "Person" hasName an instance of class "Name". See that the instance of a class in the subject position is not fixed, while it is in the object position, the property_value must be an instance of class "Name". **Figure 2. 2** shows the graph representation of **List 2. *2***, for a compact and a better visibility we short cut the namespaces into a prefix:Class_#/property_#.

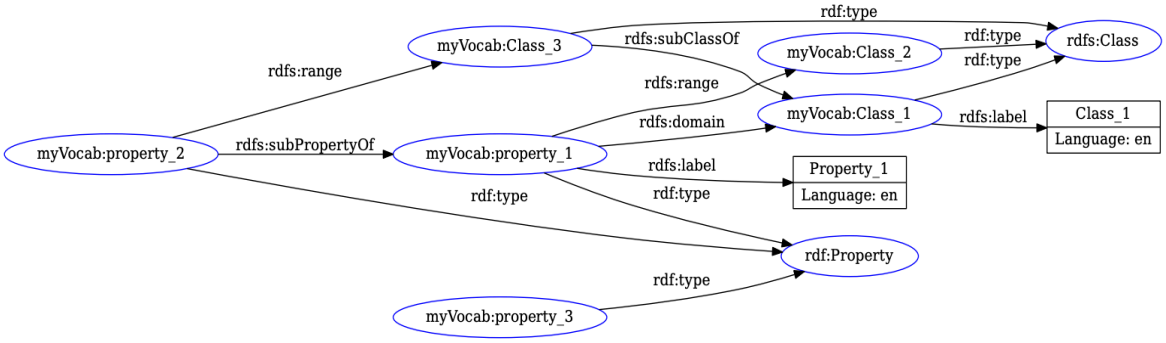


Figure 2. 2: Graph representation of our vocabulary in ***List 2. 2***.

At this point, we presented RDFS a W3C recommendation for describing RDF vocabulary and listed its main terms. We used those terms and created a vocabulary defining classes and properties and the relation between them. To provide a general overview, we have not defined a domain for our vocabulary, nor have we given specific labels to the classes and properties. However, we provided simple examples on what these classes and properties can be and how to use them. In **Appendix 6** we briefly introduce two more Semantic Web standards namely SKOS and OWL. SKOS is RDF vocabulary for representing KOSs (taxonomies, thesauri, classification schemes, and more), and publishing them into the semantic web, and OWL a Web Ontology Language used to define ontologies.

2.1.3 SPARQL Query Language

SPARQL, stands for SPARQL Protocol and RDF Query Language, is an RDF query language and data access protocol for the Semantic Web. The SPARQL language includes IRIs, Internationalized Resource Identifiers, which are a subset of RDF URI References. On January 2008 SPARQL 1.0 became a standard by the W3C's SPARQL Working Group, and a newer version SPARQL 1.1 was released on 21 March 2013 as a W3C official recommendation. A SPARQL 1.1 definition by the W3C SPARQL Working Group (W3C, SPARQL 1.1, 2013):

"SPARQL 1.1 is a set of specifications that provide languages and protocols to query and manipulate RDF graph content on the Web or in an RDF store."

SPARQL 1.1 (W3C, SPARQL 1.1, 2013) **includes various specifications, to name a few:**

SPARQL 1.1 Query Language for RDF.

SPARQL 1.1 Query Results JSON- CSV- TSV- and XML Formats

SPARQL 1.1 for exchanging answers to SPARQL queries, in JSON, CSV and TSV

SPARQL 1.1 Query Language for executing queries distributed over different SPARQL endpoints.

SPARQL 1.1 Update Language for RDF graphs.

SPARQL 1.1 Protocol for RDF - means for exchanging SPARQL queries and update requests to a SPARQL service.

In this thesis we will only be concentrating on SPARQL query language (SPARQL 1.1 Query Language for RDF). The query language provides four different query forms; SELECT, ASK, DESCRIBE, and CONSTRUCT. Also here, we will be focusing on the SELECT query, still, give a brief overview to the remaining three queries, summarized from (W3C, SPARQL 1.1, 2013) / SPARQL 1.1 Query Language.

SPARQL is like RDF in the concept of triples (or triple pattern), i.e., subject, predicate, and object. However, with the difference that SPARQL triples can contain variables at any position. The variables can be understood as place holders, defined by a **'?'** character followed by a user determined string, e.g., ?variableName. Terms (subjects, predicates, and objects) defined in a vocabulary with a URI are used by either writing the entire URI inside

of angle brackets <>, or the short cut using the namespace prefix with the term's name e.g., myVocab:property_1. Namespace names in RDF/XML are defined using xmlns: namespace prefix as the example below:

```
xmlns:myVocab="http://my-vocabulary.de/my-vocabulary#"
```

in SPARQL they are defined with the keyword PREFIX namespace-prefix: as the example below:

```
PREFIX myVocab: <http://my-vocabulary.de/my-vocabulary#>
```

Defining a base URI is similar to that in RDF/XML, using the BASE directive at the first line of a query, example:

```
BASE <http://my-vocabulary.de/my-vocabulary#>
```

A triple pattern in SPARQL may look as follows (Yu, 2011):

```
1. <http://.../exampleSubject> <http://.../examplePredicate>
   <http://.../exampleObject>.

2. <http://.../exampleSubject> ex:examplePredicate ?varName.

3. ex:exampleSubject ?varName "value".

4. ?varName ex:examplePredicate ex2:exampleObject.

5. ?varNameSubj ex:examplePredicate ?varNameObj.

6. ?varNameSubj ?varNamePred ?varNameObj.
```

Another concept of SPARQL is called graph pattern, which is a collection of triple patterns within the curly brackets {}, example in **List 2. *3***. The graph pattern is used to select triples from a given RDF graph.

List 2. 3:Example collection of triple patterns, i.e., Graph Pattern.

```
{
    ?item_1 ex:hasWeight ?weight.
    ?item_1 ex:hasWidth ?width.
    ?item_1 ex:hasHeight ?height.
    ?item_1 ex:hasCreationDate ?date.  }
```

As you can see the subject variable '?item_1' is the same in every triple. This means that its value is the exact same in every line result. This graph pattern will try to find all resources that has all four properties (hasWeight, hasWidth, hasHeight, and hasCreationDate) defined. In this thesis we will only be using the SELECT query (for the test in Chapter 6), therefore it's the only query to be explained here. However, we would like to suggest these works (Yu, 2011) and (Lerning SPARQL) for any interest in learning more about SPARQL.

The SELECT query form is used to assembly standard queries. It directly returns the queried variables with new variable bindings into the solution. The SELECT query form has the following structure: a base directive BASE (optional), list of prefixes with keyword PREFIX, result description with clause SELECT, selected graph templet with clauses (FROM, FROM NAMED), query pattern with clause WHERE {…}, condition filters (query dependent) with clauses (FILTER, FILTER NOT EXISTS, MINUS), query modifiers (optional) with clauses (ORDER BY, LIMIT).

The BASE directive and PREFIX definition are as described above. The SELECT clause specifies the variable bindings, to be returned from the query. Those variable bindings are user determined preceded by a question mark '**?**'. The FROM and FROM NAMED clauses are to specify the RDF data set(s) (or named graph(s)) to be queried against. This can however be optional in some cases, depending on the SPARQL querying service. In some SPARQL querying services the dataset can be uploaded to the service and no FROM/FROMNAMED clauses need to be used. If these clauses were used, then the dataset uploaded to the service will be overridden and the specified dataset is the one to be used. Multiple number of FROM and FROM NAMED clauses can be used in one query. The resulting dataset/graph consist of the merged graphs referred to in the FROM clauses, and a set of graphs, one from each FROM NAMED clause (W3C, 2013).

```
FROM <http://.../exampleDataset>
FROM NAMED <http://.../examples/item1>
FROM NAMED <http://.../examples/item2>
```

The WHERE clause contains the triple-/graph pattern that specify what to query for in the given dataset. To give a query example:

```
1. PREFIX myVocab: <http://my-vocabulary.de/my-vocabulary#>
2. PREFIX ex: <http://www.example.org/list_of_items#>
```

```
3.
4. SELECT ?varName WHERE{
5. <http://.../exampleSubject> ex:examplePredicate ?varName.
6. }
```

Lines 1-2 specify the namespace names. In line 4 the SELECT clause specify a binding variable '?varName' to hold the query result, and starts the WHERE clause. Line 5 shows the triple pattern that says, for the specified subject `<http://.../exampleSubject>` return the property value, stored in the given dataset, and bind it in `?varName`.

To return all triples in a given dataset, we can use the following basic query:

```
1. PREFIX myVocab: <http://my-vocabulary.de/my-vocabulary#>
2. PREFIX ex: <http://www.example.org/list_of_items#>
3.
4. SELECT *
5. FROM <http://.../exampleDataset>
6. WHERE{?s ?p ?o.}
```

With SELECT * (line 4) we tell SPARQL to select everything within the dataset, specified via the clause FROM (line 5), WHERE the query result is triples of subjects, predicates and objects (line 6 {?s ?p ?o}).

To specify conditions, what results should (not) be returned, we can use the clauses (FILTER, FILTER NOT EXISTS and MINUS). Note that, FILTER NOT EXISTS, and MINUS are equivalent. To give an example:

```
4. SELECT ?s ?w WHERE{
5. ?s ex:hasWeight ?w.
6. FILTER(?w <= "12")
7. }
```

We skipped the namespace names definition, for being the same in all examples. Starting with line 4 with the SELECT clause and '?s' ?w the binding variables, WHERE ?s hasWeight with value ?w. In line 6, we specify a condition, telling SPARQL to only return results with '?w' smaller or equal to 12, using `FILTER(?w <= "12")`.

If we used `FILTER NOT EXISTS`, or `MINUS` the result will be only those with '?w' bigger than 12, as the example below:

```
4. SELECT ?s ?w WHERE{
5. ?s ex:hasWeight ?w.
6. FILTER NOT EXISTS(?w <= "12") #or MINUS (?w <= "12")
```

```
7.}
```

To specify the order by which we want the results to appear, we can use the `ORDER BY` clause with optionally an ascending- ***ASC()*** or descending- ***DESC()*** order. To limit the query result to a specific number, we can use the clause LIMIT. To give an example:

```
4. SELECT ?s ?w WHERE{?s ex:hasWeight ?w.}
5. FILTER(?w <= "12")
6. ORDER BY ASC(?w)
7. LIMIT 10
```

This query will only return 10 triples from the given dataset, ordered ascendingly.

So far we have learned about the Resource Description Framework RDF and how it is used to describe pieces of knowledge in a way understandable for machines. We also learned about the RDF/XML syntax and how it is used to express an RDF knowledge graph as an XML document. We also learned how to use the RDF-Schema RDFS and the Web Ontology Language OWL to create an ontology with few defined classes, properties and the relation between them. And finally, we learned how to query RDF datasets using the SPARQL Query Language. All these were components of the semantic web and specified as W3C standards.

The Semantic Web isn't just about publishing data on the web, it's about linking them. When we publish and connect these machine-readable RDF datasets together, we create an interconnected web of data that can be explored and understood by both humans and machines. This brings us to the concept of Linked Open Data.

2.1.4 Linked Open Data

Linked Data is simply the data that has an explicitly defined meaning, published on the web, machine readable, linked- and can be linked- to other datasets (Yu, 2011, S. 409). The concept of Linked Data was brought by Tim Berners-Lee in 2006 with the idea publishing structured data on the web using RDF and linking it to different data sources using RDF links to create the Web of Data.

"The Semantic Web isn't just about putting data on the web. It is about making links, so that a person or machine can explore the web of data."

(Berners-Lee, 2006)

There are four main rules (Berners-Lee, 2006), (Yu, 2011, S. 412-413) to ensure a correct application of the idea of Linked Data:

1. Using URIs to name things, as we explained in the previous sections,
2. Use HTTP URIs so that these names can be found by agents (humans and machines), emphasizing rule 1, making sure that the identifiers are globally unique,
3. URIs should provide useful information when looked up by an agent, for instance, when defining a person, we refer to them with a URI where we use analogies to describe information about that person, not a URI of a website that, for example, belongs to that person,
4. Linking to other URIs, include links to direct agents to other (related) resources.

Linking data can be made in various ways, to give examples:

1. Linking two people within an RDF document '<http://example.org/people#>' <http://example.org/people#**person_1**> **:relatedTo** <http://example.org/people#**person_2**>

This would be represented in RDF/XML as:

```
<rdf:Description rdf:about="http://example.org/people#person_1">
  <:relatedTo rdf:resource="http://example.org/people#person_2"/>
</rdf:Description>
```

2. Linking to different resources, example:

```
<rdf:Description rdf:about="http://example.org/people#person_1">
  <:hasEMail rdf:resource="eMailIs:person_1@example.com"/>
  <:worksAt rdf:resource="http://example.org/company_xy"/>
  <rdfs:seeAlso rdf:resource="http://example.org/people#person_3"/>
</rdf:Description>
```

Linked Open Data is simply Linked Data with open license --Creative Commons as example of open license -- and can be reused for free. Not all Linked Data are open data, it can be Linked Data that only used internally and not open to the public.

In the next sub-section, we will present an overview of presenting archaeological artifacts on the web focusing on ancient coins and coin finds, around which this thesis revolves.

2.2 Presenting Archaeological Artifacts on The Web

Ancient artifacts, including coins, have archaeological, cultural and historical importance. The study, publication and linking of knowledge about these artifacts offers an enriched understanding of cultural heritage. The need for unified collection, management and exchange of data between different cultural heritage institutions was the motive to adopt the semantic web technologies for creating ontologies and providing a common basis for organizing, modeling and managing cultural data (Moraitoua, Christodouloua, & Caridakisa, 2021).

Many ontologies were created for the field of cultural heritage, to give few examples: the International Committee of Documentation Conceptual Reference Model (CIDOC CRM) of the International Council of Museums (ICOM) and its extentions (CRMinf, CRMarchaeo,CRMgeo, CRMdig, and more) which provides classes and properties for varios topics of the cultural heratig domain The Architecture of Knowledge (ArCo) ontology network, a collaborative work that reuses existing ontologies to build a network of standard for representing cultural heritage data. And publishing data of the Italian Ministry of Cultural Heritage (ICCD) as linked open data. Nomisma.org, a collaborative project that provides stable HTTP based URIs for representing numismatic concepts based on the principles of Linked Open Data.

The focus of this thesis is on using the Nomisma.org-Ontology for representing numismatic objects on the web like ancient coins and hoards. Before we dive into the modeling process in the next chapter, we would first like to give a brief overview about ancient coins.

Ancient Coins as Carrier of Cultural Heritage

Ancient coins are a very rich historial documents and archealogical objects for the information they hold about all classes of socity from their production authority to the diffrent archaeologically traceable stages being in circulation (Kemmers & Myrberg , 2011). They bear an ensemble of features enriched with historical cultural- and social-events that require an interdisciplinary approach to understand all their dimensions, their context, and the societies they link.

A coin's context refers to the different stages of a coin's lifecycle, where each stage can be examined individually. Four main life stages of coins can be identified, which are:

- **Primary context:** Production
- **Secondary context:** Use
- **Tertiary context:** Deposition or loss, becoming archaeological objects.
- **Quaternary context:** Retrieval and submission to antiquarian or scientific studies.

Understanding these four stages involves understanding other contextual aspects such as: temporal, social, ideological, functional, and geographical features surrounding each of these stages. These contextual aspects can give a broader or more in-depth perspective in observing these stages. For instance, understanding the social context of a coin as an object means understanding the functionality of coinage in society and the interactions between people, between people and objects and between societies, "*the use of coins involves all levels of society"* (Kemmers & Myrberg , 2011).

Coins are historical-archaeological sources, they provide us with evidence from ancient times. For instance, the production of a coin is mostly related with an issuing authority, the state, or the ruling power at its time. The issuer determined the purity of the metal, weight, and the design (e.g., shape, color, iconography) and inscription (legends) stamped on coins, that identify the legal state of it (Wickens). The value of the coin (denomination) and the mint itself was mostly determined and run by the state. These dimensions give a more informative and enforced meaning of the world surrounding the coins embodying them.

Not only individual coins and its contexts give broad analysis chances, but also studying group of coins, that were buried together in antiquity and later found in modern time, helps evaluating the chronology of coins and analyzing the circulation patterns of them (Gruber , et al., 2012, S. 264). They can help finding a correlation between their quality and the time in which they were deposited/buried and learning about other aspects like the economic conditions of that time period (Oras, 2013).

These groups of coins, with groups of other valuable ancient items, are archaeologically known as 'Hoards'. Coins, for example, may have been placed within a hoard's vessel at once and the coins themselves may sometimes be arranged according to

their quality, value or according to various standards. In other times, they could have been stored over a long period of time (Walker, 2018). In either case, such placements may form layers, whereby each layer can be examined with its content separately. Therefore, it is required to find a modeling approach for representing these layers.

2.3 Summary

In this chapter we introduced the theoretical foundations of this thesis starting with the semantic web, describing it as an extention of- or a layer on top of the current web, that gives well-defined meaning to the information, making them understandable for both humans and machines. We learned how RDF is used to modeling the statements to represent knowledge in a machine understandable manner. We learned how to represent pieces of knowledge as an RDF garph and to convert that into the RDF/XML serialisation syntax, and about RDF Shame, which we used to create a simple ontology.

The agent (human or machine) must be able to process queries on the statements. SPARQL, an RDF query language and data access protocol for the Semantic Web. The semantic web is not only about making data maschine readable, its about linking those data and in somecases making them open to the public. Linked Open Data, was introduced as the last topic of the first part of this chapter (the Semantic Web and its technologies).

The second part of this chapter gave an overview of the importance of archaeological artifacts in understanding cultural heritage, with a focus on ancient coins. And that the study of 'hoards', groups of valuable archaeological objects, offers a broader perspective on the interactions between societies. The next chapter presents use cases of modeling information about ancient coins using the numismatic concepts provided by Nomisma.org, as well as two approaches for modeling the layers of a hoard vessel.

3 Modelling Structured Data

The machine-readable data is called structured data, it is the data that conforms to a predefined model or format and is used to create websites that are easily understood by search engines (Structured vs. Unstructured Data: A Complete Guide). We learned that these predefined formats are called 'Ontologies', and they are domain specific, meaning, they were created to describe and represent an area of knowledge. The area of knowledge we will be focusing on in this thesis is the area of numismatics and cultural heritage. In the previous chapter we learned names of several ontologies in these domain areas including the Nomisma ontology and CIDOC CRM, to give examples.

In this chapter we will present some approaches for modeling structured data in the domain of numismatics using the Nomisma ontology, giving some use cases of this ontology as practice suggestions, of which adherence will provide better interoperability.

3.1 Modelling Nomisma Ontology

Nomisma.org is a collaborative project, Initiated in 2010 by the American Numismatic Society, to provide stable http based URIs for the digital representation of numismatic concepts based on the principles of Linked Open Data (Berners-Lee, 2006). These stable digital representations provide access to reusable information on numismatic concepts to improve interoperability between distributed numismatic collections and provide connections to other areas of study.

Nomisma.org provides not only a wide range of defined classes and properties under the Nomisma-Ontology, but also over 7000 predefined Nomisma-IDs, as instances of the Nomisma classes, for representing numerous numismatic concepts, distributed collections of numismatics and cultural heritage as well as fields of numismatic study. These predefined property values include various aspects such as: object -type, -material, art/and place of -manufacture, -issuer, -region, -period and much more.

3.1.1 Best practices Use Cases

This section introduces some use cases of the Nomisma.org ontology and IDs. The modeling use cases are for the classes and properties listed below. For more use cases please see **Appendix 1**. The classes/properties are as defined in (Nomisma Ontology, 2021).

Content: *Ethnic, Collection and ReferenceWork, Corrosion and Wear, Denomination, Hoard, Appearance, Object Type.*

Ethnic

Class: nmo:Ethnic. Generally, a tribal or ethnic name appearing on a numismatic object. Distinct from personal authority as indicated by the presence of ruler's name.

Class Instances: nm:indo-sasanians, nm:dyrrhachii, nm:zaeelii, nm:volga_bulgars.

Object property: nmo:hasLegend. Describes the inscription or printing placed on a numismatic object as part of the production process.

Use case: Coin_1 has the word ‘zaeelii’ inscripted on it.

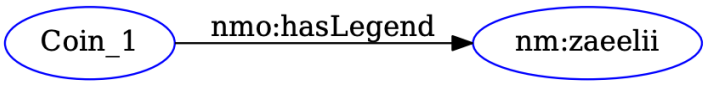


Figure 3. 1: Coin_1 has an inscription ‚zaeelii ‘, which is an instance of ‚Ethnic ‘.

```
<rdf:Description rdf:about="Coin_1">
    <nmo:hasLegend rdf:resource="nm:zaeelii"/>
</rdf:Description>
```

Collection and ReferenceWork

Class: nmo:Collection. A general term referring to the objects held by an institution, an individual or the holding entity itself.

Class instances: nm:vatican_museums, nm:goethe_uni_frankfurt_arch_wiss, nm:uva, …

ObjectProperty: nmo:hasCollection. Identifies the collection or repository that holds (or has held) the numismatic object.

Use case: Coin_1 falls within the "Coin Collections of the Goethe University Frankfurt am Main".

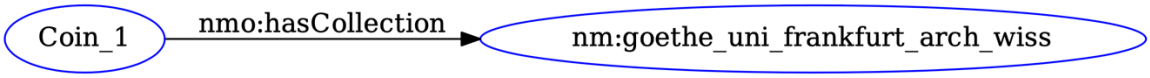


Figure 3. 2: Coin_1 contains in the coin collection of the Goethe-University.

RDF/XML:

```
<rdf:Description rdf:about="Coin_1">
     <nmo:hasCollection rdf:resource="nm:goethe_uni_frankfurt_arch_wiss"/>
</rdf:Description>
```

Class: nmo:ReferenceWork. A published work of reference relevant to a numismatic object.

Class instances: nm:igch, nm:belfort, nm:prou_mero, nm:head_1911.

Example: "igch" stands for "An Inventory of Greek Coin Hoards".
Object Property: nmo:hasReferenceWork. A published work of reference relevant to a numismatic object.

Use case:

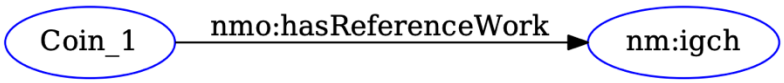


Figure 3. 3: Coin_1 was published in a reference work "An Inventory of Greek Coin Hoards".

RDF/XML:

```
<rdf:Description rdf:about="Coin_1">
     <nmo:hasReferenceWork rdf:resource="nm:igch"/>
</rdf:Description>
```

Corrosion and Wear

Class: nmo:Corrosion. Degradation to a numismatic object due to chemical reaction with its environment.

Class instances: nm:heavily_corroded, nm:extremely_corroded, nm:lightly_corroded, …

Object Property: nmo:hasCorrosion

Use case: Coin_1 is lightly corroded

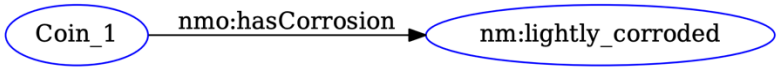


Figure 3. 4: Coin_1 is lightly corroded.

RDF/XML:

```
<rdf:Description rdf:about="Coin_1">
    <nmo:hasCorrosion rdf:resource="nm:lightly_corroded"/>
</rdf:Description>
```

Class: nmo:CoinWear. Wear on a coin or similar objects that resulted from its use.

Class instances: nm:extremely_worn, nm:lightly_worn, nm:little_to_no_wear, …

Class: nmo:DieWear. Wear on a die that resulted from its use.

Object Property: nmo:hasWear. Describes wear visible on a numismatic object due to use or circulation, commonly in terms prescribed by "Abnutzung und Korrosion":

Use case:

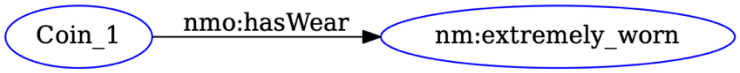


Figure 3. 5: Coin_1 is extremely worn due to its use

RDF/XML:

```
<rdf:Description rdf:about="Coin_1">
    <nmo:hasWear rdf:resource="nm:extremely_worn"/>
</rdf:Description>
```

We first mistakenly thought that corrosion is some sort of wear and created a use case of an object that has wear with type of wear being corrosion. This was corrected by Dr. Tolle explaining that wear and corrosion being two different things that are also different in time:

- **Wear:** Is the wear and tear that occurs during use (circulation) of the coin.
- **Corrosion:** Describes the process of corrosion (i.e., the change in material) while the coin is hidden in the ground.

Denomination

Class: nmo:Denomination. Term indicating the value of a numismatic object.

Class instances: nm:chalkous, nm:large_ae2, nm:3-Taler, nm:onkia, nm:denarius, …

Object Property: nmo:hasDenomination

Use case: Coin_1 has value of denarius

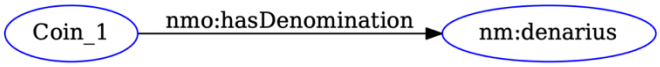


Figure 3. 6: Coin_1 has value of denarius.

RDF/XML:

```
<rdf:Description rdf:about="Coin_1">
    <nmo:hasDenomination rdf:resource="nm:denarius"/>
</rdf:Description>
```

Hoard

Class: nmo:Hoard. A group of numismatic objects: a store of wealth concealed with intent of recovery but which, for reasons not usually known to us, was not recovered until modern times.

Class instances: nm:normanby_hoard, nm:lliria_hoard, nm:brauweiler_hoard, …

Example: Lliria_Hoard. "Was found in 1999, at the Duc of Lliria, a hoard of 5,991 Roman denarii was uncovered, hidden under the floor of a house.**"** (Nomisma.org, 2016)

Object Property: nmo:hasFindspot. Describes the location of the discovery of an object, whether by accident or in archaeological excavation.

Use case: Hoard_1 was found in year '1984', in location_, and contains of table of content_h1.

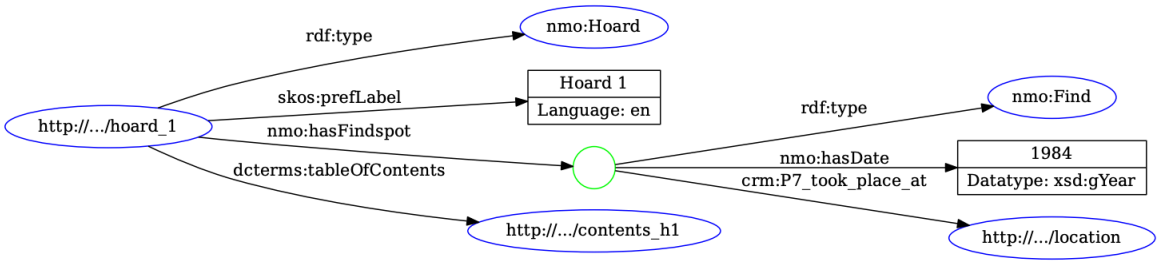


Figure 3. 7: Hoard_1 was found in year '1984', in location_, and contains of table of content_h1.

RDF/XML:

```
<nmo:Hoard rdf:about="http://.../hoard_1">
    <skos:prefLabel xml:lang="en">Hoard 1</skos:prefLabel>
    <nmo:hasFindspot>
        <nmo:Find>
            <nmo:hasDate rdf:datatype="xsd:gYear">1984</nmo:hasDate>
            <crm:P7_took_place_at rdf:resource="http://.../location"/>
        </nmo:Find>
    </nmo:hasFindspot>
    <dcterms:tableOfContents rdf:resource="http://.../contents_h1"/>
</nmo:Hoard>
```

Appearance

Class: nmo:Mintmark. A mint mark is a letter, symbol or inscription on a numismatic item indicating the mint where the numismatic object was produced.

Class: nmo:SecondaryTreatment. Anomalies or unusual features which occurred after the actual production of a numismatic object.

Instances of class: SecondaryTreatment: nm:cut, nm:clipped, nm:bent, nm:halved, nm:graffito, …

Object Property: nmo:hasObverse. The face of a numismatic object, carrying the representation, badge, or inscription of the issuing authority.

Object Property: nmo:hasReverse. The face of a numismatic object opposed to the obverse, or 'tails' in english.

Object Property: nmo:hasAxis. The directional relationship between the obverse and reverse.

Object Property: nmo:hasIconography. The iconography on a numismatic object.

Object Property: nmo:hasControlmark. A letter, symbol, monogram, or an inscription on a numismatic object intended to distinguish it as part of a group of numismatic objects within an issue or series.

Object Property: nmo:hasCountermark. A mark or symbol punched into a coin or similar object at some point during its time in circulation.

Object Property: nmo:hasPortrait. A portrait on a numismatic object, of a person or deity.

Object Property: nmo:hasLegend. The inscription or printing placed on a numismatic object as part of the production process.

Object Property: nmo:hasSecondaryTreatment. Unusual features on a numismatic object occurred after the actual production, e.g., ‘graffito’ a lettering or similar mark.

Use case: Coin_1 has bears date ‘231’, has an iconography of ‘athena’, has some inscription ‘Iω MIRCEA VOEVODA’, a control mark and its edge are serrated.

Use case: Coin_1 has control mark ‘M’, which is a mint mark.

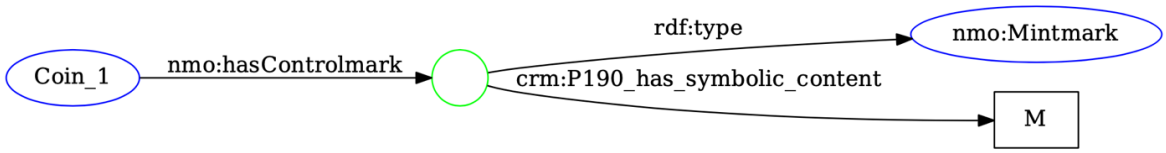


Figure 3. 8: Coin_1 has a control mark 'M', referring to the mint.

RDF/XML:

```
<rdf:Description rdf:about="Coin_1">
    <nmo:hasControlmark>
        <nmo:Mintmark>
            <crm:P190_has_symbolic_content> M </crm:P190_has_symbolic_content>
        </nmo:Mintmark>
    </nmo:hasControlmark>
</rdf:Description>
```

Use case: Coin_1 has counter mark, an iconography of what appeared to be 'apollo', but this is uncertain (more to modeling uncertainty in the next section).

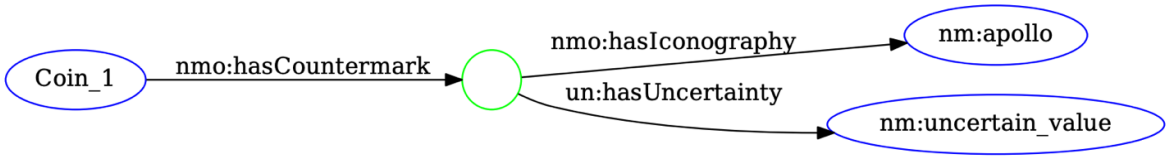


Figure 3. 9: Coin_1 has an iconography of a deity could be 'apollo', punched to it during its circulation, not part of its production.

Use case: Coin_1 has obverse with portrait of 'titus', legend 'T CAES IMP VESP CEN', and a reverse with a graffito 'CA'.

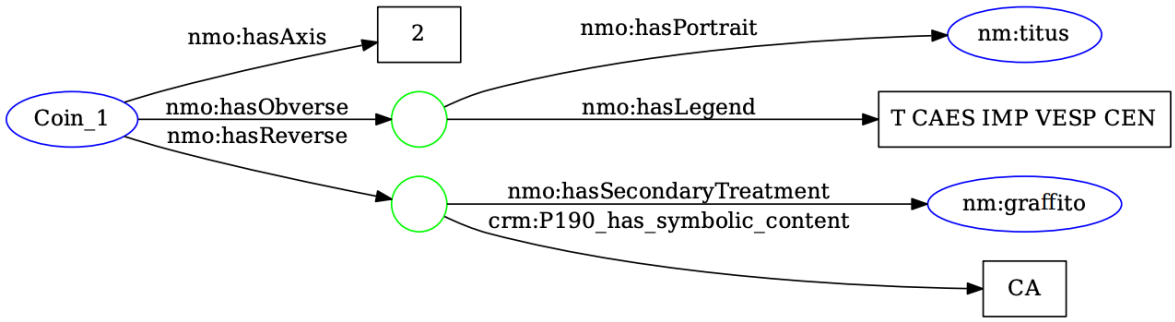


Figure 3. 10: Describing the obverse and reverse of Coin_1

Turtle-Syntax:

```
:Coin_1 nmo:hasAxis " 2 " ;
```

```
    nmo:hasObverse [ nmo:hasLegend " T CAES IMP VESP CEN " ;
            nmo:hasPortrait nm:titus ] ;
    nmo:hasReverse [ nmo:hasSecondaryTreatment nm:graffito ;
            crm:P190_has_symbolic_content "CA" ] .
```

Object Type

Class: nmo:ObjectType. The type of a numismatic object.

Class instances: nm:coin, nm:Die, nm:token, nm:seal, nm:amulet, …

Object Property: nmo:hasObjectType

Use cases:

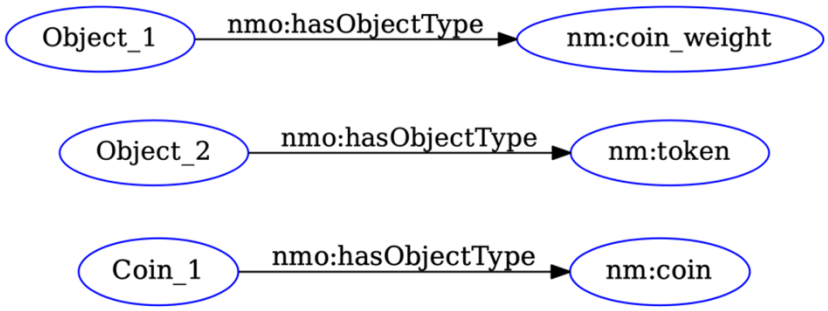


Figure 3. 11: The different pre-defined class instances of object type

RDF/XML:

```
<rdf:Description rdf:about="Object_1">
    <nmo:hasObjectType rdf:resource="nm:coin_weight"/>
</rdf:Description>

<rdf:Description rdf:about="Object_2">
    <nmo:hasObjectType rdf:resource="nm:tokin"/>
</rdf:Description>

<rdf:Description rdf:about="Coin_1">
    <nmo:hasObjectType rdf:resource="nm:coin"/>
</rdf:Description>
```

These were only few cases of using the Nomisma-ontology with its pre-defined class instances. One of the main objectives of this thesis was to create some sort of a cookbook containing these use cases.

3.1.2 Nomisma.org CookBook

The idea was to create a wiki-page containing use cases of the Nomisma-Ontology. We created the wiki-page 'Nomisma.org Cookbook', first approach was inserting all classes and properties of the Nomisma-Ontology with the use cases in one page. As serialization syntax we only used the RDF/XML, and we did not include examples of the existing instances of the Nomisma classes. The page became too large – over 90 pages in word for **Appendix 1** to give an example – and with a wiki-page one cannot edit the classes and properties individually nor can include individual editor's name. Therefore, Dr. Tolle suggested creating a 'Post' for each class and each property instead of one large page and to include examples of the existing class instances, which we implemented accordingly (Nomisma.org-Cookbook, 2022).

Also, for the serialization syntax, Prof. Heath (Nomisma.org) suggested that the current RDF/XML might be easier to read in Turtle. Therefore, we used the Turtle syntax in some use cases, especially those that would have been too long in RDF/XML (in **Appendix 1** we included both).

Feedback by Dr. Wigg-Wolf (Nomisma.org) on the Nomisma-Cookbook:

> *"aus Sicht der Numismatik kann ich eigentlich nur sagen, dass das Cookbook genau das ist, was wir brauchen! Wie Prof. Andrew Meadows gesagt hat, wir hätten vor Jahren damit anfangen sollen. Es ist leicht verständlich und wird für viele, die mit der Ontologie arbeiten wollen aber sie noch nicht kennen und verstehen, sehr hilfreich sein.*
> *Wir werden nach und nach an den Beispielen für die einzelnen Einträgen arbeiten müssen. Aber eine sehr gute Grundlage ist da. Danke!"*

3.2 Modeling the Layers of a Hoard Vessel

A hoard is a collection of valuable items (e.g., coins), that were buried or lost in the antiquity and have been found in the modern time. The items in a hoard vessel may have been placed all at once or over time, they could also be ordered according to certain scales such as material or value, to give examples. In any of these cases, there is the possibility of separating the contents of the vessel into layers, while maintaining its sequence inside the vessel. With this emerged the need for a solution to modeling such layers.

In this section we would like to suggest two approaches (solution 1 & 2) to modeling these layers and their content. For this purpose, we will be referring to the coin finds as 'Hoards' as a general term used in the Nomisma-Ontology to represent a lost or deposited numismatic object or group of numismatic objects. We will use the keyword 'item' to refer to the individual content (coin, jewel, medallion, etc.) within these layers.

3.2.1 Solution 1: Using Nomisma-Properties for Sequencing the Units of Stratification

To describe the layers of hoard vessel we will be using three of the Nomisma.org properties provided to describe the sequence of the units of stratification. These three properties are: isAbove, isBottom and isEqual, and they are based on the Harris Matrix system for units of archaeological stratification. We can use these properties to describe the layers of a hoard vessel and thereby the individual items they contain. **Figure 3.** ***12*** below shows how to represent the layers of a hoard in form of Harris matrix, and **Figure 3.** ***13*** illustrates the Harris matrix of the individual items contained in each layer.

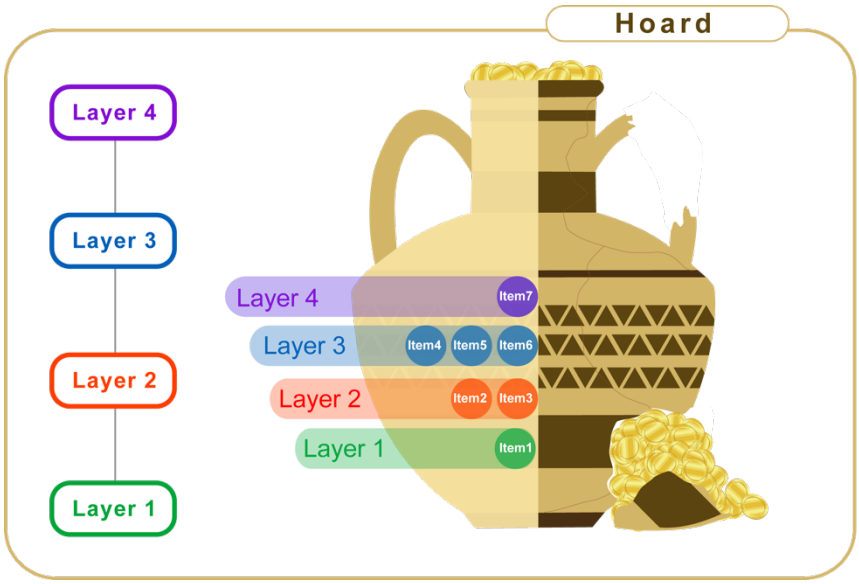


Figure 3. 12: Representing the layers of a hoard in form of Harris Matrix

The hoard in this example (**Figure 3.** ***12***) consists of four layers, where these layers can be distinguished by the time interval the items were poured into the vessel (land deposit), by the item's material (savings hoard), or any other feature. And the triple representation of these layers is with using the Nomisma properties isAbobe and isBelow as follows:

- Layer_1 nmo:isBelow Layer_2
- Layer_2 nmo:isAbove Layer_1 Layer_2 nmo:isBelow Layer_3
- Layer_3 nmo:isAbove Layer_2 Layer_3 nmo:isBelow Layer_4
- Layer_4 nmo:isAbove Layer_3

Supposing, layer 1 contains 1 item (1), layer 2 has 2 items (2 & 3), layer 3 contains 3 items (4, 5 & 6), and layer 4 has 1 item (7). The Harris matrix representation of these items corresponds to that of the layers that contain them.

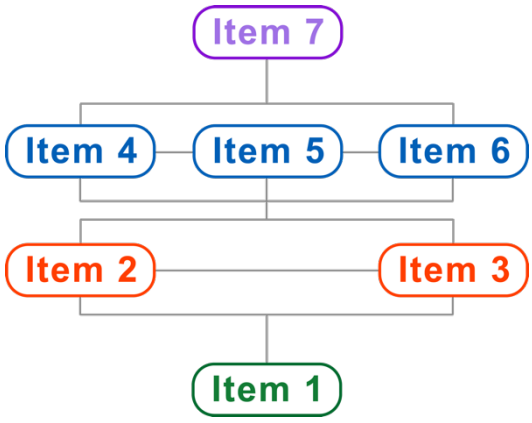


Figure 3. 13: Illustrates the Harris Matrix of the individual items contained in each layer

To represent this as RDF graph (**Figure 3.** ***14***), RDF/XML, Turtle respectively we will be using the following properties:

- *`dcterms:tableOfContent`* for the full content of the hoard,
- *`crm:P167i_includes`*, for this table of content includes layers 1-7 (we will only include 2 layers here for a better visibility), and
- each layer is of type *`nmo:StratigraphicUnit`*.

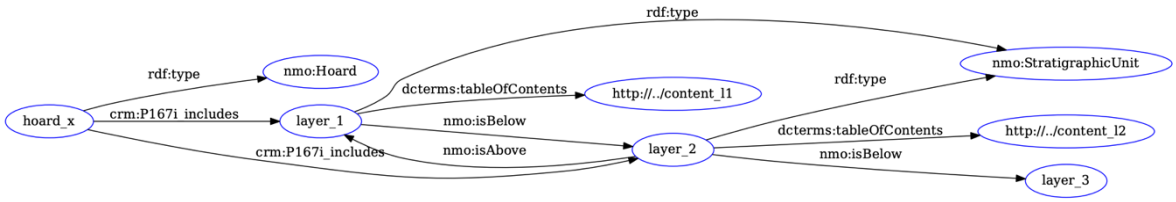


Figure 3. 14: RDF-Graph representation of hoard_x using the Nomisma properties isAbove and isBelow to describe the different layers.

And in RDF/XML, and Turtle formats respectively:

```
<nmo:Hoard rdf:about="hoard_x">
    <crm:P167i_includes>
        <nmo:StratigraphicUnit rdf:about="layer_1">
            <dcterms:tableOfContents rdf:resource="http://../content_l1"/>
            <nmo:isBelow rdf:resource="layer_2"/>
        </nmo:StratigraphicUnit>
    </crm:P167i_includes>
    <crm:P167i_includes>
        <nmo:StratigraphicUnit rdf:about="layer_2">
            <dcterms:tableOfContents rdf:resource="http://../content_l2"/>
            <nmo:isAbove rdf:resource="layer_1"/>
```

```
            <nmo:isBelow rdf:resource="layer_3"/>
        </nmo:StratigraphicUnit>
    </crm:P167i_includes>
      …
</nmo:Hoard>
```

```
:hoard_1 a nmo:Hoard ;
    crm:P167i_includes :layer_1, :layer_2 , :layer_3 , :layer_4 .
        <http://example.org/layer_2> .

:layer_1 a nmo:StratigraphicUnit ;
    nmo:isBelow :layer_2 ;
    dcterms:tableOfContents <http://../content_l1> .

:layer_2 a nmo:StratigraphicUnit ;
    nmo:isAbove :layer_1  ;
    nmo:isBelow :layer_3  ;
    dcterms:tableOfContents <http://../content_l2> .
…
```

Figure 3. ***15*** shows the graph representation of modeling the individual items within the layers using the inverse property of crm:P167i_includes. The RDF/XML and turtle syntaxes of this model can be found below.

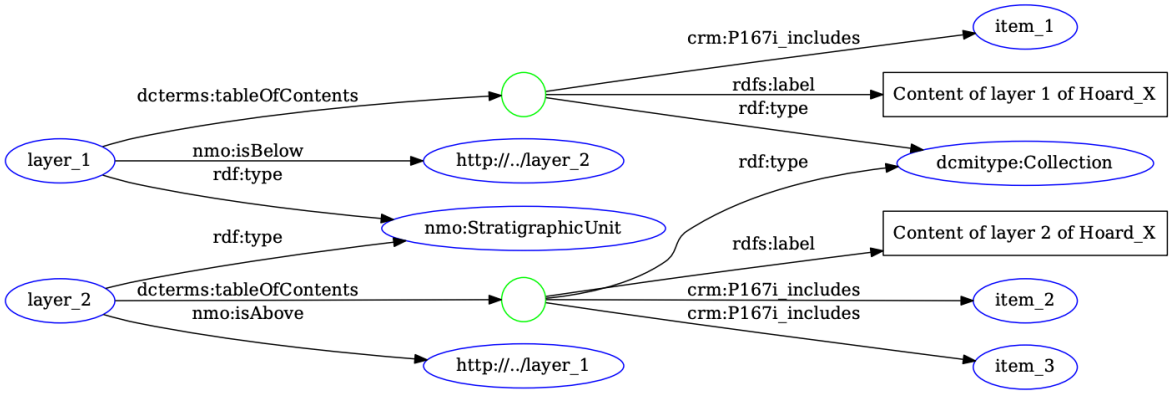


Figure 3. 15: RDF-Graph representation of layers 1-2, each with its items included using CIDOC-CRM inverse property P167i_includes.

In RDF/XML and Turtle respectively:

```
<nmo:StratigraphicUnit rdf:about="layer_1">
    <dcterms:tableOfContents>
        <dcmitype:Collection>
            <rdfs:label>Content of layer 1 of Hoard_X</rdfs:label>
```

```
            <crm:P167i_includes rdf:resource="item_1"/>
        </dcmitype:Collection>
    </dcterms:tableOfContents>
    <nmo:isBelow rdf:resource="http://../layer_2"/>
</nmo:StratigraphicUnit>
<nmo:StratigraphicUnit rdf:about="layer_2">
    <dcterms:tableOfContents>
        <dcmitype:Collection>
            <rdfs:label>Content of layer 2 of Hoard_X</rdfs:label>
            <crm:P167i_includes rdf:resource="item_2"/>
            <crm:P167i_includes rdf:resource="item_3"/>
        </dcmitype:Collection>
    </dcterms:tableOfContents>
    <nmo:isAbove rdf:resource="http://../layer_1"/>
</nmo:StratigraphicUnit>
```

```
:layer_1 a nmo:StratigraphicUnit ;
    nmo:isBelow <http://../layer_2> ;
    dcterms:tableOfContents [ a dcmit:Collection ;
            rdfs:label "Content of layer 1 of Hoard_X" ;
            crm:P167i_includes :item_1] .

:layer_2 a nmo:StratigraphicUnit ;
    nmo:isAbove <http://../layer_1> ;
    dcterms:tableOfContents [ a dcmit:Collection ;
            rdfs:label "Content of layer 2 of Hoard_X" ;
            crm:P167i_includes :item_2,
            :item_3 ] .
```

Modeling each of these items can be made separately and be referenced here within the hoard model. A full model for this solution can be found in **Appendix 2**.

3.2.2 Solution 2: Using RDF-Collection

RDF provides a pre-defined 'Collection Vocabulary' to describe a common group of resources/items of certain members and in a given specific order. This vocabulary includes four keywords: rdf:first, rdf:rest, rdf:List, and rdf:nil. **Figure 3.** ***16*** and **Figure 3.** ***17*** shows the RDF graph of modeling a hoard's content layers and the items within each layer respectively, using RDF-Collection vocabulary, with the RDF/XML and Turtle syntaxes below.

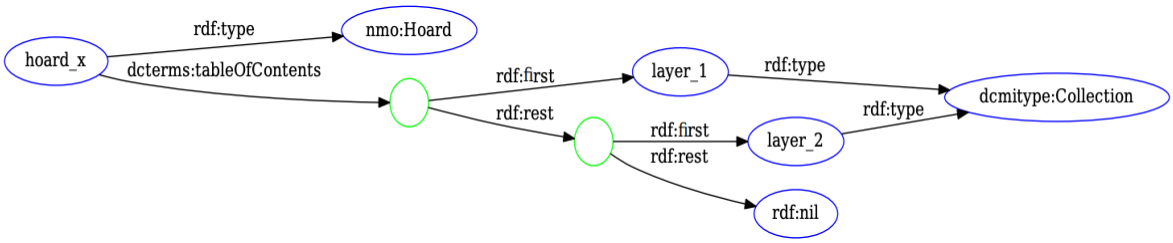


Figure 3. 16: Modeling layers of a hoards using RDF-Collection.

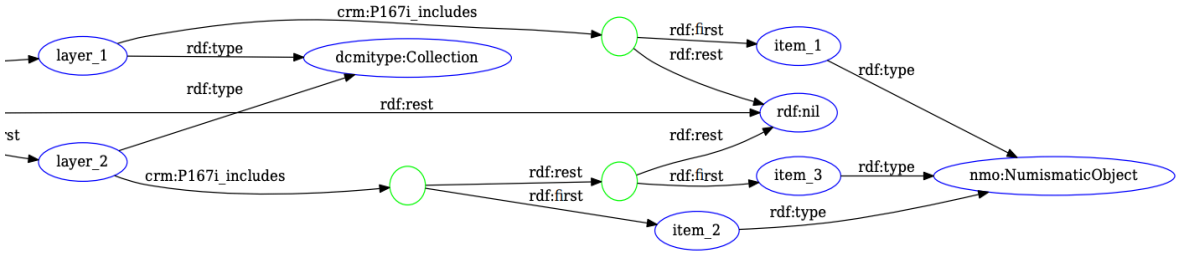


Figure 3. 17: RDF-Graph representation of layers 1-2, each with its items included using CIDOC-CRM inverse property P167i_includes.

In RDF/XML and Turtle respectively:

```
<nmo:Hoard rdf:about="hoard_x">
    <dcterms:tableOfContents rdf:parseType="Collection">
        <dcmitype:Collection rdf:about="layer_1">
            <crm:P167i_includes rdf:parseType="Collection">
                <nmo:NumismaticObject rdf:about="item_1"/>
            </crm:P167i_includes>
        </dcmitype:Collection>
        <dcmitype:Collection rdf:about="layer_2">
            <crm:P167i_includes rdf:parseType="Collection">
                <nmo:NumismaticObject rdf:about="item_2"/>
                <nmo:NumismaticObject rdf:about="item_3"/>
            </crm:P167i_includes>
        </dcmitype:Collection>
    </dcterms:tableOfContents>
</nmo:Hoard>
```

```
:hoard_x a nmo:Hoard ;
    dcterms:tableOfContents ( :layer_1 :layer_2 ) .

:item_1 a nmo:NumismaticObject .
:item_3 a nmo:NumismaticObject .
:item_4 a nmo:NumismaticObject .

:layer_1 a dcmit:Collection ;
```

```
    crm:P167i_includes ( :item_1 ) .
:layer_2 a dcmit:Collection ;
    crm:P167i_includes ( :item_2 :item_3 ) .
```

As it shows in the RDF/XML syntax, items must be modeled inside the hoard model and not referenced (rdf:resource="item_i" is not allowed here, only rdf:about="item_i" is accepted). For example:

```
<dcmitype:Collection rdf:about="layer_1">
   <crm:P167i_includes rdf:parseType="Collection">
      <nmo:NumismaticObject rdf:about="item_1">
         <nmo:hasMint rdf:resource="nm:comama"/>
         <nmo:hasWeight rdf:datatype="xsd:decimal">12.1</nmo:hasWeight>
         <foaf:depiction rdf:resource="https://.../image"/>
      </nmo:NumismaticObject>
   </crm:P167i_includes>
</dcmitype:Collection>
```

Also, the fact that each member of an RDF-Collection is connected via an empty node means that this collection container is closed, because other RDF documents cannot access the blank nodes (Yu, 2011). To maintain the same pattern with every entry RDF/XML provides a notation, namely 'rdf:parseType' with value 'Collection', to describe a close container. This eliminates the need to write the keywords rdf:first, rdf:rest, and rdf:nil manually (Yu, 2011), as shown in the RDF/XML lines above.

3.3 Summary

In the first part of this chapter, we introduced the Nomisma.org, a domain specific ontology of numismatic concepts, and gave some best practice suggestions of modeling examples, following which promotes better interoperability. We created use cases for almost every term in the Nomisma-Ontology and posted them within a wiki page under Nomisma-CookBook.

The second part of this chapter presents two approaches for modeling the layers of a hoard vessel. One approach handles the layers as stratigraphic units – a Nomisma-class– and uses the Nomisma-properties isAbove, isBelow and isEqual—based on Harris-Matrix – to describe the position of each layer within the vessel. The other approach uses the RDF-

Collection, a closed container, within which the entries are organized by the order of inclusion.

Information about coins as archaeological objects is complex and, in many cases, involve uncertainty. But even uncertain information in this matter can be of great value, and here emerges the need of a way to model this information and distinguish it from reliable ones. The next chapter introduces eight approaches to modeling uncertainties in numismatic.

4. Modeling Uncertainties in Ancient Coins

A large number of ancient coins show strong signs of wear and corrosion for a variety of reasons, e.g., accidentally damaged during their circulation period or eroded due to their state of preservation. This detracts their informative features e.g., depiction, date or symbols referring to the place of mint, making them unrecognizable even for experts. Identifying these features can occur in many different degrees; certain if the feature clearly identifiable or uncertain where the feature cannot be (clearly) identified; uncertain with one possibility, uncertain with specific number of possibilities, uncertain with many possibilities or uncertain with unlimited possibilities.

This chapter deals with investigating approaches for handling uncertainties in the numismatic domain.

4.1 Approaches for Modelling of Uncertainties in Ancient Coins

There is still no uniform method for modeling uncertainties, some datasets been referring to it as text in the property object or in the coin (title) description, and some been using Nomisma predefined IDs for uncertainty, illustrated in **Figure 4. *1***. These modeling examples may seem clear and simple, but things are easily misinterpreted when shared as open data. For instance, if there are multiple value options for the Mint, including them in a text would be unclear and may not be simple to query. Also, creating an object value (such as a nomisma ID) for each possibility is inefficient. For simplicity, we will consider the following situation in all modeling examples of this section:

- *Coin_1* been a well preserved and can un-confusingly be identified, it was certainly minted in *Comama*.
- *Coin_2* poorly preserved, has unclear, uncertainly recognizable features, could be minted in *Comama* but is uncertain.

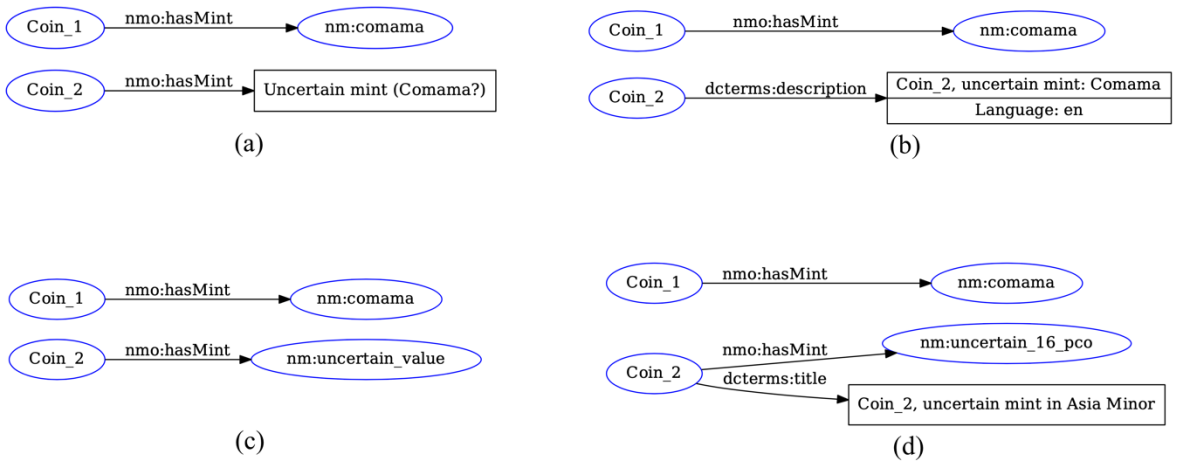


Figure 4. 1: Coin_1 minted in "Comama" is certain, but for Coin_2 this is uncertain. (a) as text for the value of property hasMint, (b) as text in the coin description, (c) un-specified Nomisma-id for uncertain features in general and (d) using specified Nomisma-id for uncertain mint like; "Uncertain Mint 16" for uncertain "Ptolemaic Mint 16 in Asia-Minor". Nomisma.org provides a number of specified IDs for possible uncertain features (concepts) *(Laskey K. J., 2008)*.

And in RDF/XML:

(a)

```
<rdf:Description rdf:about="Coin_2">
    <nmo:hasMint>Coin_2, ncertain mint (Comama?)</nmo:hasMint>
</rdf:Description>
```

(b)

```
<rdf:Description rdf:about="Coin_2">
    <dcterms:description>Coin_2, ncertain mint (Comama?) </dcterms:description >
</rdf:Description>
```

(c)

```
<rdf:Description rdf:about="Coin_2">
    <nmo:hasMint rdf:resource="nm:uncertain_value"/>
</rdf:Description>
```

(d)

```
<rdf:Description rdf:about="Coin_2">
    <nmo:hasMint rdf:resource="nm:uncertain_16_pco"/>
    <dcterms:title>Coin_2, uncertain mint in Asia Minor</dcterms:title>
</rdf:Description>
```

Uncertainty modeling has long been an area of active research and a discussion topic in many communities. A simple way to add uncertainty is to use RDF_Reification (statement about a

statement) to say that Coin_2 minted in comama is uncertain, shown in **Figure 4. *2***. However, this would mean using five to six more triples (triple explosion).

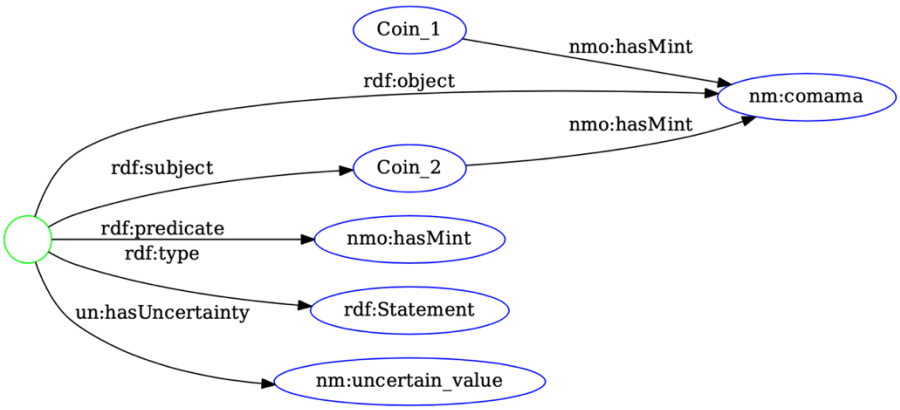


Figure 4. 2: Using RDF-Reification to express that it is uncertain whether Coin_2 was minted in Comama, while for Coin_1 is certain.

The following are approaches (solutions) for modeling uncertainties. The first two solutions were presented in (Uncertainty Handling for Ancient Coinage., 2014), the next three solutions (S3-S5) are our interpretation of the discussions by (Doerr M. e., 2014) in the CIDOC CRM community. We also suggest three further approaches (S6-S8), two of which (S6 and S7) based on S1 and S2 respectively. To explain the different approaches, we take the situation of Coin_1 minted in Comama is certain, and for Coin_2 this is uncertain.

4.1.1 Solution 1: RDF-Reification-Based Approach by CIDOC CRM

An approach similar to RDF-Reification based on CIDOC CRM (Uncertainty Handling for Ancient Coinage., 2014) (E13 Attribute Assignment) in conjunction with the Research Spacers (RS) projects (BM Association Mapping) (Alexiev, Confluence, 2013), in **Figure 4. 3**.

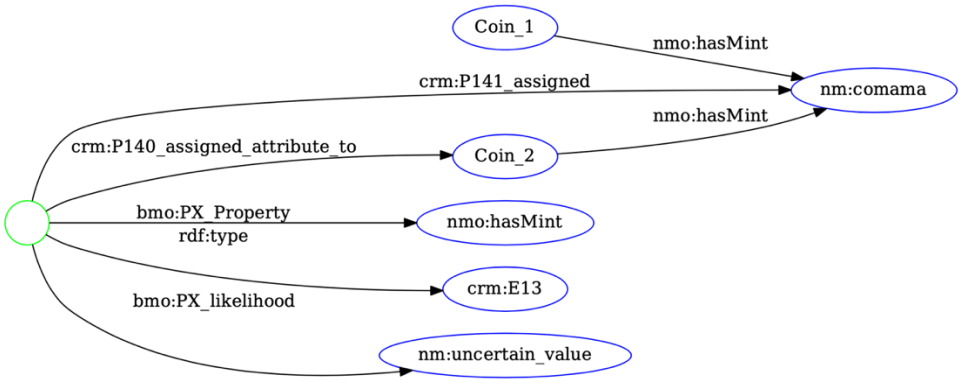


Figure 4. 3: CDOC_CRM/ RS Uncertainty modeling similar to RDF_Reification. Coin_2 minted in Comama is uncertain, while for Coin_1 it is certain.

As in the standard RDF-Reification, five additional triples are required to express the uncertainty of one feature. Also, querying for coins with certain properties only, nm:comama as certain mint in this example, is complex.

4.1.2 Solution 2: Adding a Resource Directly to the Property Path

A proposal by (Tolle & Wigg-Wolf, 2014) for modeling the uncertainty is given by adding an additional resource directly to the property path.

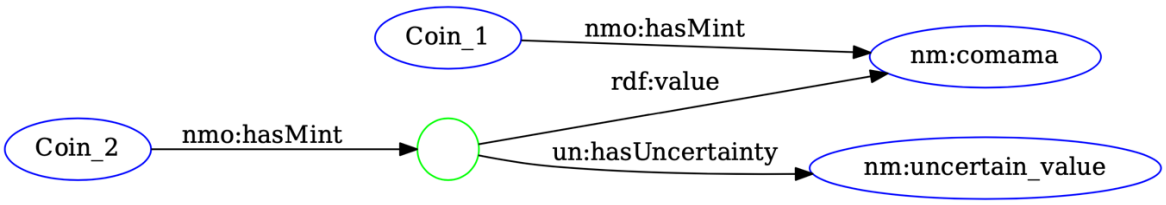


Figure 4. 4: Expressing uncertainty for coin_2 minted in Comama

With this approach fewer triples are needed, and querying for certain features is simple. For example, using SPARQL and giving the following query:

$$SELECT\ ?coin\ WHERE\ \{\ ?coin\ nmo{:}hasMint\ nm{:}comama\ \}$$

The result will return the coin that is certain to be minted in "Comama", i.e., Coin_1. Though, for Coin_2 a more complex query must be written. This is different to the approach of

CIDOC_CRM/ RS, where this query returns both Coin_1 and Coin_2. Whereas to retrieve only the certain results (Coin_1) require a more complex query (**Appendix 3**). With this approach of (Tolle & Wigg-Wolf, 2014) much less additional triples are required (two more triples) and inferring information from uncertain data can be avoided, while them still in reach if needed. However, it is not possible to fix the property's ranges (Tolle & Wigg-Wolf, 2014). One potential option is to define, for example, a X_Property_Class as the scope of (each) Nomisma-property, but this would mistakenly mean that the blank-node (with ID eventually) to be derived as an instance of this scope class. With reification and the CIDOC CRM/RS approach such instantiation can be done easily. Although, strict definitions of properties' domains and ranges may prevent them from being reused.

4.1.3 Solution 3: Assigning Reliability Instead of Uncertainty

Solutions 3-5 are proposed in the "45th joint meeting of the CIDOC CRM SIG and Issue 349: Belief Values" (45th joint meeting, 2009, S. 2) (Doerr M. e., 2014), to handle uncertainty. It was not easy translating these into graphs and then into RDF/XML. Therefore, we contacted Dr. Doerr (FOURTH Greece, Chair, CIDOC-CRM Special Interest Group) to ask for his opinion of our interpretation of solutions (S3-S5). We will include his opinion on each of the solutions after reviewing them.

This solution 3, by (Niccolucci, 2016), proposes the following; instead of assigning uncertainty is to assign reliability as subclass of E16 Measurement, that measures a dimension, and assert a numerical value as E60 number for the reliability through P90 has value. The activity of assessing the reliability of a statement is treated as E13 Attribute Assignment. The result of the measurement (reliability) can also be a function or an ordinal value. We tried to translate this into RDF/XML, the result is illustrated in **Figure 4. *5*** and **Figure 4. *6***.

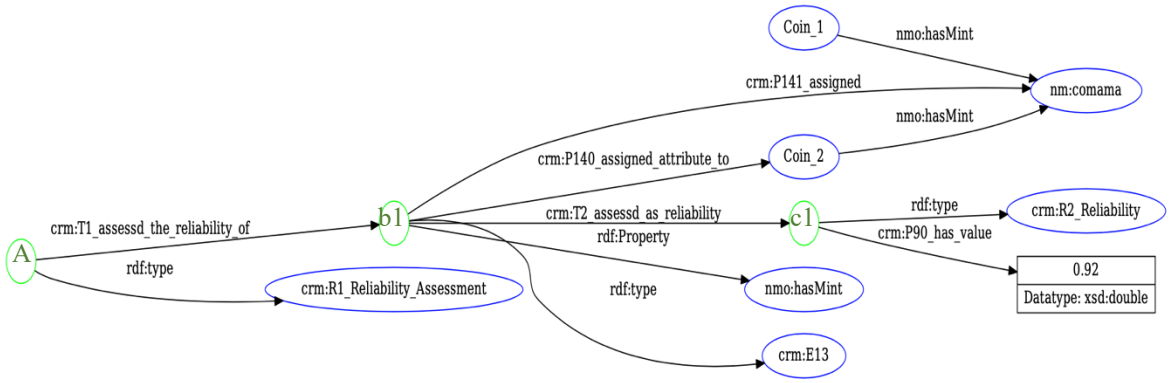


Figure 4. 5: Solution 3; assigning reliability instead of uncertainty and passing a reliability value through P90_has value. Treating the reliability assessment as E13_ Attribute Assignment.

E16 Measurement has properties: P39 measured and P40 observed dimension: E54 Dimension, which has property P90 has value. So, we defined R1, R2, T1 and T2 as follows:

- R1_Reliability_Assessment: subClass of E16_Measurment
- R2_Reliability: subClass of E54_Dimension
- T1_assessed_the_reliability_of: subProperty of P40_ observed_dimension
- T2_assessed_as_reliability: subProperty of P39_measured

We thought of the blank-node-A as the wrapper for all blank-nodes-b that is meant to assess the reliability of the statement it delivers. For instance, the blank-node-b1 refers to the following: the statement *"Coin_2 minted in Comama"* has a reliability of *"0.92"*. If we want to add a new reliability value: *"Coin_2 minted in Cretopolis"* with reliability of *"0.08"*, *"Coin_2 shows a portrait of Titus"* or a statement about another coin e.g., "Coin_3 minted in Comama" (comama or some other place) that also have some reliability value, we will deliver any of those statements using a blank-node (e.g., b2) and assign the reliability of it within A as illustrated in **Figure 4.** *7*.

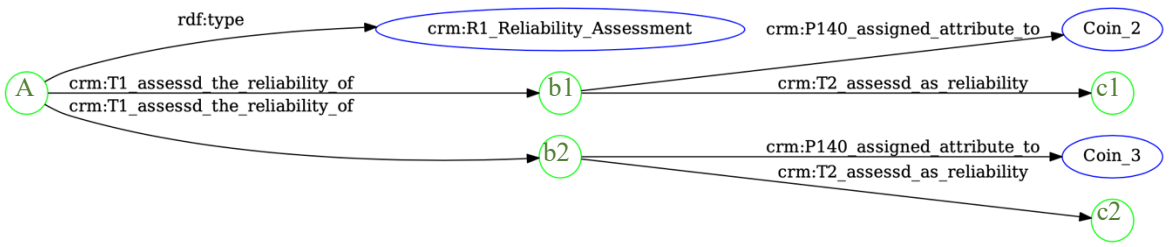


Figure 4. 6: Solution 3; asserting the reliability of multiple statements of different coins.

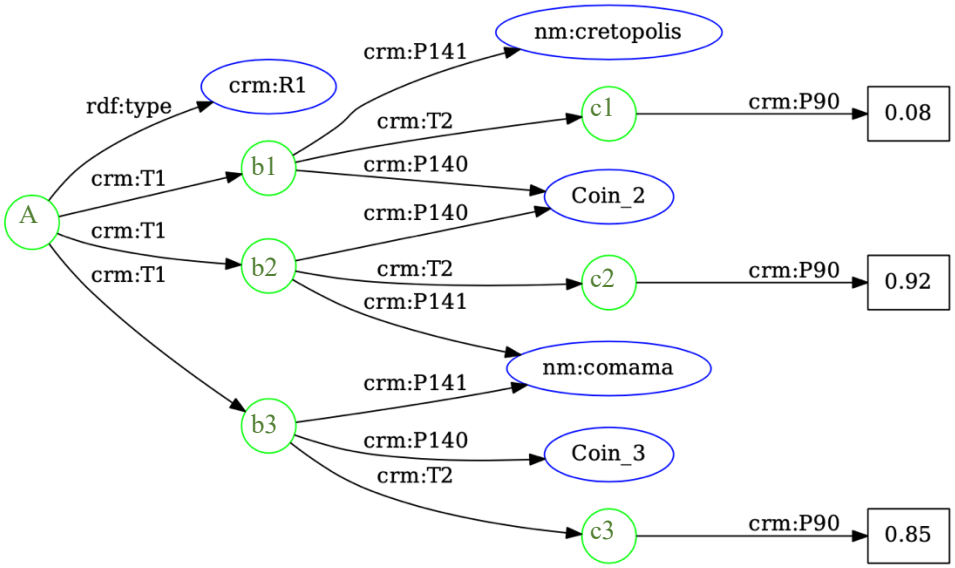


Figure 4. 7: Solution 3; asserting the reliability of two statements about the Coin_2 minted in Comama with reliability 0.92, or minted in Cretopolis with reliability 0.08, and another statement of a different coin; Coin_3 minted in Comama with reliability 0.85.

Dr. Doerr means that he would not use this approach because it is not really a Measurement. Instead, he would create a new property for E13 "confidence value". Translating this into an RDF graph, we came out with the following model:

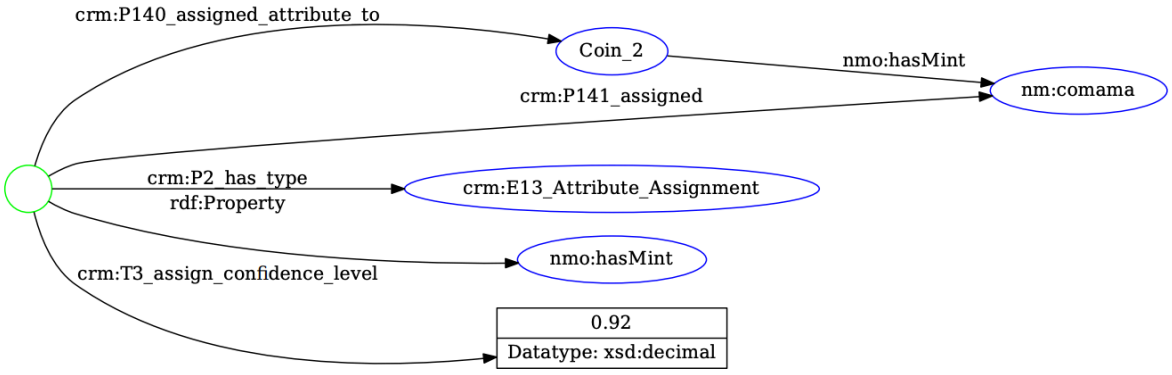


Figure 4. 8: Solution suggesting creating a new property of E13 'confidence value', instead of the idea proposed in solution 3, which is R1_Reliability_Assessment as sub-class of E16_Measurment.

With T3 restricted to domain E13:

```
<rdf:Property rdf:about="T3_assign_confidence_level">
```

```
  <rdfs:domain rdf:resource="crm:E13_Attribute_Assignment"/>
</rdf:Property>
```

If we translated Dr. Doerr's suggestion correctly, we could see that this solution is very close to solution 1.

4.1.4 Solution 4: Using CRM*inf* and Assigning Belief Values

This solution is also proposed by (Niccolucci, 2016) and in (45th joint meeting, 2009, S. 2), the idea is to use properties/classes from CRMinf, namely J4_that and J5_holds_to_be together with I4_Proposition_set to assign belief values (I6) to statements obtained through observations. As a first interpretation of this idea, we got the following model **Figure 4. *9*** based on RDF-Reification:

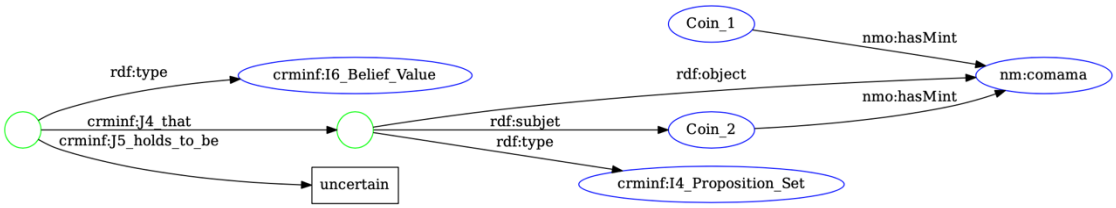


Figure 4. 9: Solution 4; using properties/classes from CRMinf to add uncertainty, based on RDF-Reification.

We tried a different way (**Figure 4. *10***) by using the same approach as in solution 2, for it requires less additional triples than solution 1, respective RDF-Reification.

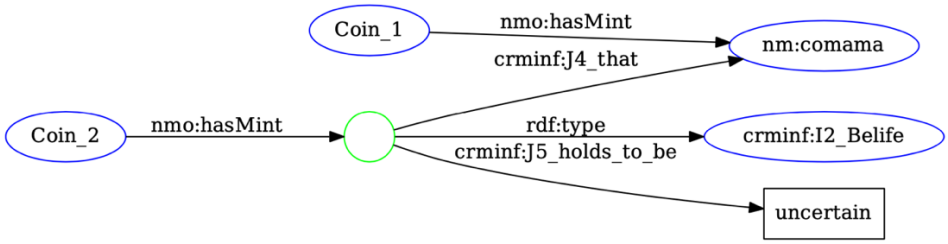


Figure 4. 10: Solution 4; using properties/classes from CRMinf to add uncertainty, based on Solution 1.

If there is more than one option for the value a property can have belief proposition "Coin_2 minted in comama, is uncertain, but it been minted in cretopolis, is more likely". We suggest using (I5 Inference Making), illustrated in **Figure 4. *11***, along with (J2 concluded that) to add the belief that "Coin_2 minted in cretopolis, is more likely" as another proposition:

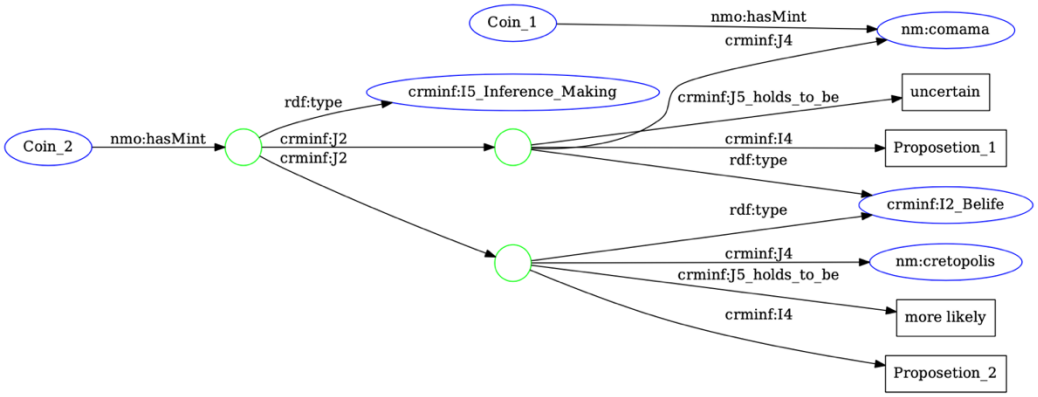


Figure 4. 11: Solution 4; using I5 Inference Making from CRMinf to add multiple beliefs about Coin_2 minted either in comama, uncertain, or in cretopolis, more likely.

Dr. Doerr's opinion about using CRMinf was that it is good when one has supporting arguments, but it works with Named Graphs.

4.1.5 Solution 5: Extending CRM with Properties of Properties

This solution, proposed in (45th joint meeting, 2009) by Dr. Doerr (Doerr M. e., 2014), suggests expanding CRM properties with the "Property Class" PC and adding a "certainty value" as a ".2" property (properties of properties). As we understood, .2 properties can only be assigned to the CRM properties that already have a .1 property (listed in **Table A4. *1*** in **Appendix 4**). So, we tried to create .2 properties, listed in **Table A4. *2*** in **Appendix 4**, to use with all the currently existed Nomisma properties. Translating this solution into RDF/XML we decided to base on the approach of ***solution 2***, by adding an additional resource directly in the property path **Figure 4. *12***, as it requires less additional triples. We also used "*amt:weight"* from the "*Academic Meta Tool Voca*bulary" (Thiery F. U., 2018), which represents a degree of vagueness and takes a value between [0;1].

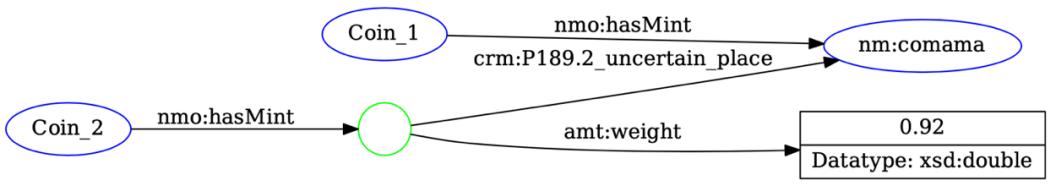


Figure 4. 12: Solution 5; creating CRM .2 properties to express uncertainty in every possible aspect, using amt:weight to represent the degree of vagueness/uncertainty.

RDF does not support properties of properties, (Doerr M. L., 2020, S. 12-13) suggests two workarounds; a) defining the .2 properties locally, for they all have range “E55 Type”, so they "correspond to subtyping of the respective property by a local vocabulary" (Doerr M. L., 2020, S. 12-13). b) using "property class PC" and defining the .2 properties as open vocabulary. At first, we resorted to solution a) and defined our customed .2 properties locally as described in (Doerr M. L., 2020), for more freedom, we decided not to specify a range. Eexample for uncertain mint:

```
<rdf:Property rdf:about="P189_uncertain_place">
    <rdfs:domain rdf:resource="E1_CRM_Entity"/>
    <rdfs:subPropertyOf rdf:resource="P189_ approximates"/>
</rdf:Property>
```

Dr. Doerr’s opinion on this solution was that we should use the "PC" property class model (FORTH-ICS, 2014) and reinterpret to our properties. And that "this model replaces each property by an RDF-Class". So, we added our .2 properties to the model as shown in this example:

```
<rdf:Property rdf:about="P189.2_uncertain_place">
    <rdfs:domain rdf:resource="PC189_approximates"/>
    <rdfs:range rdf:resource="E55_Type"/>
</rdf:Property>
```

Dr. Doerr advised not to use CRM properties with Nomisma, unless we map Nomisma to CRM:

“A first thought is that mixing CRM with the nomisma ontology is problematic. The latter is not CRM compatible, because it shortcuts the events in the history of the coin, as I understand. Anyway, Attribute Assignment, is neutral to the property it assesses, as well as CRMinf.”

He also recommended:

„If you have not too many weights, you can instead subtype your properties with "has_mint_very_likely", "has_mint_likely", "has_mint_unlikely" etc. This is what I normally recommend instead of properties of properties. It is more efficient for querying. "has_mint_very_likely", "has_mint_unlikely" may even be subproperties of "has_mint_likely", depending on the logic you would apply.“

But if we want to create such subproperties for every Nomisma property that can have uncertainty would lead to property-explosion and may not be very flexible (Thiery F. N.-W., 2021). Still, this suggestion can be applicable with creating Nomisma-IDs for such different degrees of likelihood that can be used with, for example, bmo:PX_likelihood instead of amt:weight. This also applies to the next suggestion of using "multi-valued logic".

Dr. Doerr discourages using real number uncertainty weights, as in solutions 3 and 5, for they lack empirical basis. Instead, he recommended using "multi-valued logic"; True, Very-Likely, Likely, Unlikely, False, etc., and that more than three or five degrees of uncertainty will hardly be used by researchers. This was also reported by (Tolle & Wigg-Wolf, 2014), experts would not like to weight the likelihood/uncertainty for each property. Also, if different experts entered these weights, then they would need to be adjusted accordingly as experts can grade coins differently and therefore those weights will not be very comparable.

4.2 Further Approaches for Modeling Uncertainty

Solutions 6-8 are our own attempt for modeling uncertainty. Solutions 6 and 7 are based on the use of UncertainStatement/ApproximateStament, classes of the Extended Date/Time Format (EDTF) Ontology (Congress, 2022) (Shaw, 2021), which are also sub-classes of Qualified statement a sub-class of rdf:Statement **Figure 4.** ***13***. Solution 8 is a new approach using un:hasUncertainty as a property of the item that has uncertainty.

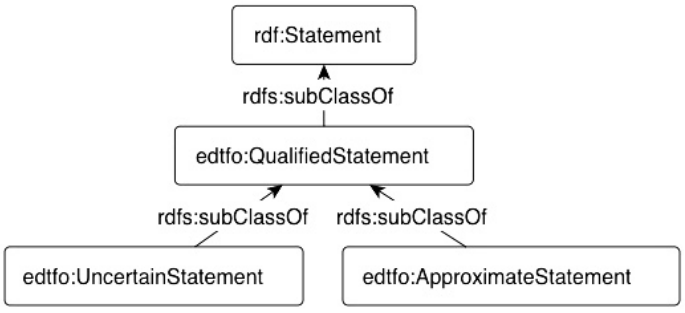


Figure 4. 13: edtfo:QualifiedStatement a sub-class of rdf:Statement, edtfo:UncertainStatement and edtfo:ApproximateStament subclasses of edtfo:QualifiedStatement.

4.2.1 Solution 6: RDF-Reification with Extended Date/Time Format Ontology (EDTFO)

The Extended Date/Time Format (EDTF), created by the Library of Congress with the participation of communities with related interests (Congress, 2022), defines features to be supported in a date/time string. And the EDTF concepts, by (Shaw, 2021), a draft ontology for expressing EDTF constructs using the Time Ontology in OWL, with the namespace prefix 'edtfo'.

For this solution we will be using RDF-Reification with an EDTFO-Class 'UncertainStatement', "a statement the source of which is considered dubious" (Shaw, 2021). With this example (**Figure 4.** ***14***) we describe the statement 'Coin_2 minted in Comama' as an uncertain statement.

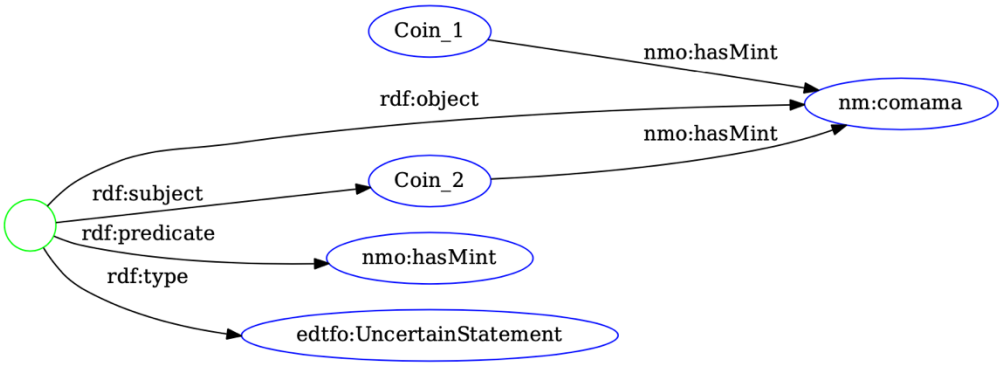


Figure 4. 14: RDF-Reification with edtfo:UncertainStatement.

In RDF/XML and turtle respectively:

```
<rdf:Description rdf:about="Coin_1">
    <nmo:hasMint rdf:resource="nm:comama"/>
</rdf:Description>

<rdf:Description rdf:about="Coin_2">
    <nmo:hasMint rdf:resource="nm:comama"/>
</rdf:Description>
<edtfo:UncertainStatement rdf:about="statement_c2">
    <rdf:subject rdf:resource="Coin_2"/>
    <rdf:predicate rdf:resource="nmo:hasMint"/>
    <rdf:object rdf:resource="nm:comama"/>
</edtfo:UncertainStatement>
```

```
:Coin_1 nmo:hasMint nm:comama .
```

```
:Coin_2 nmo:hasMint nm:comama .
:statement_c2 a edtfo:UncertainStatement ;
    rdf:object nm:comama ;
    rdf:predicate nmo:hasMint ;
    rdf:subject :Coin_2 .
```

4.2.2 Solution 7: Based on Solution 2 and Extended Date/Time Format Ontology (EDTFO)

This solution is based on solution 2, by Dr. Tolle and Dr. Wigg-Wolf. Instead of adding the property un:hasUncertainty with value nm:uncertain_value, we used edtfo:ApproximateStament **Figure 4.** ***15*** (edtfo:UncertainStatement is also an option). The ApproximateStatement is a statement that is possibly correct, or close to being correct.

It requires 4-triples to model the following example:1-triple for Coin_1 hasMint=comama (certain), and 3-triples for Coin_2 hasMint=comama (uncertain but possibly correct). Whereas solution 6 use 6 triples for the same example.

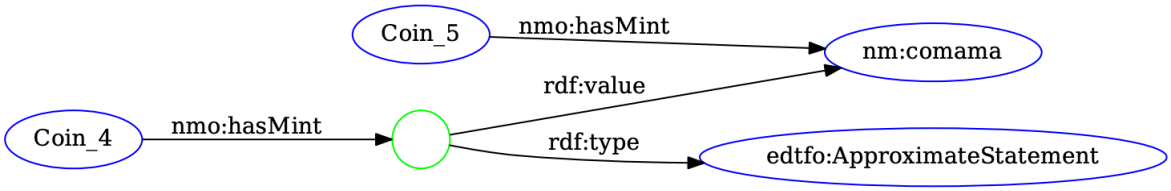


Figure 4. 15: Using EDTFO: ApproximateStatement, added directly in the property path, based on solution 2.

In RDF/XML and turtle respectively:

```
<rdf:Description rdf:about="Coin_1">
    <nmo:hasMint rdf:resource="nm:comama"/>
</rdf:Description>
<rdf:Description rdf:about="Coin_2">
    <nmo:hasMint>
        <edtfo:ApproximateStatement>
            <rdf:value rdf:resource="nm:comama"/>
        </edtfo:ApproximateStatement>
    </nmo:hasMint>
</rdf:Description>
```

```
:Coin_1 nmo:hasMint <nm:comama> .
:Coin_2 nmo:hasMint [ a edtfo:ApproximateStatement ;
```

```
            rdf:value <nm:comama> ] .
```

4.2.3 Solution 8: A New Approach Based on Using un:hasUncertainty

This new approach is based on using un:hasUncertainty as a property of a coin that has uncertainty in one or multiple features. It requires 2 tiples to express one uncertain property. We provided 3 options for this solution: simple with one uncertain property **Figure 4.** ***16***, complex with 2 certain- and 3 uncertain properties **Figure 4.** ***17***, and a more complicated one with 2 certain- 4 uncertain properties most of which with extra features **Figure 4.** ***18***.

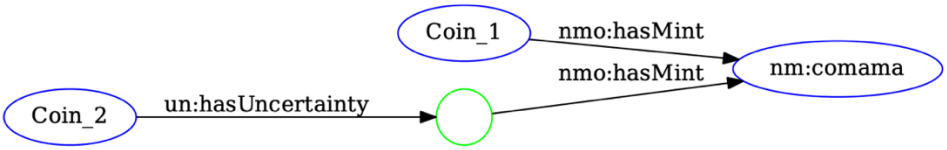


Figure 4. 16: A simple version of S8, Coin_2 has uncertainty which is the pace of mint being Comama.

In RDF/XML syntax:

```
<rdf:Description rdf:about="Coin_1">
    <nmo:hasMint rdf:resource="nm:comama"/>
</rdf:Description>
<rdf:Description rdf:about="Coin_2">
    <un:hasUncertainty>
        <rdf:Description>
            <nmo:hasMint rdf:resource="nm:comama"/>
        </rdf:Description>
    </un:hasUncertainty>
</rdf:Description>
```

This is the simplest case of Coin_2 having just one uncertain feature which is the place of mint. Note, that Coin_2 having certain features is decarded here but mentioned in the next two examples.

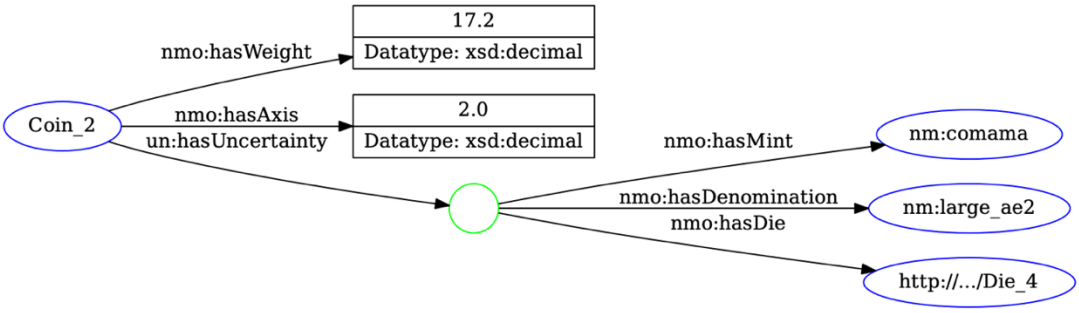


Figure 4. 17: A slight complex version of S8, Coin_2 has 2 certain and 3 uncertain features.

In Turtle syntax:

```
:Coin_2 nmo:hasAxis "2.0"^^<xsd:decimal> ;
    nmo:hasWeight "17.2"^^<xsd:decimal> ;
    un:hasUncertainty [ nmo:hasDenomination nm:large_ae2 ;
            nmo:hasDie <http://.../Die_4> ;
            nmo:hasMint nm:comama ] .
```

We omitted Coin_1 as is modelled the same in all solutions. This model has 6 triples in total, 2 for 2 certain properties, and 4 for 3 uncertain properties.

A more coplicated model we created that has 16 triples in total, with 2 certain properties and 4 uncertain properties, most of which with features:

- It is uncertain if Coin_2 was produced using Die_2 (the URI to Die_4 is provided). It is uncertain if Die_4 was in use during the time (170-162 B.C.).
- It is uncertain which of these three is the issuer of Coin_2: issuer_A, issuer_B, or issuer_C.

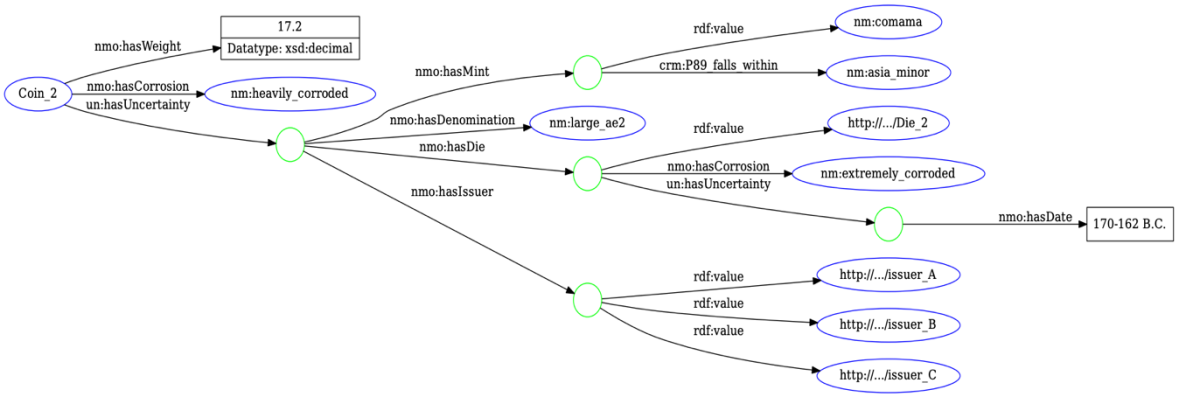


Figure 4. 18: A more complex version of S8, Coin_2 has 2 certain and 4 uncertain properties, most of which with more additional features.

In RDF/XML and turtle syntaxes respectively:

```
1.  <rdf:Description rdf:about="Coin_2">
2.      <nmo:hasWeight rdf:datatype="xsd:decimal">17.2</nmo:hasWeight>
3.      <nmo:hasCorrosion rdf:resource="nm:heavily_corroded"/>
4.      <un:hasUncertainty>
5.         <rdf:Description>
6.              <nmo:hasMint>
7.                 <rdf:Description>
8.                     <rdf:value rdf:resource="nm:comama"/>
9.                     <crm:P89_falls_within rdf:resource="nm:asia_minor"/>
```

```
                </rdf:Description>
            </nmo:hasMint>
            <nmo:hasDenomination rdf:resource="nm:large_ae2"/>
            <nmo:hasDie>
                <rdf:Description>
                    <rdf:value rdf:resource="http://.../Die_2"/>
                    <nmo:hasCorrosion rdf:resource="nm:extremely_corroded"/>
                    <un:hasUncertainty>
                        <rdf:Description>
                            <nmo:hasDate>170–162 B.C.</nmo:hasDate>
                        </rdf:Description>
                    </un:hasUncertainty>
                </rdf:Description>
            </nmo:hasDie>
            <nmo:hasIssuer>
                <rdf:Description>
                    <rdf:value rdf:resource="http://.../issuer_A"/>
                    <rdf:value rdf:resource="http://.../issuer_B"/>
                    <rdf:value rdf:resource="http://.../issuer_C"/>
                </rdf:Description>
            </nmo:hasIssuer>
        </rdf:Description>
    </un:hasUncertainty>
</rdf:Description>
```

```
:Coin_2 nmo:hasCorrosion nm:heavily_corroded ;
        nmo:hasWeight "17.2"^^<xsd:decimal> ;
        un:hasUncertainty [ nmo:hasDenomination nm:large_ae2 ;
            nmo:hasDie [ nmo:hasCorrosion <nm:extremely_corroded> ;
                    un:hasUncertainty [ nmo:hasDate "170–162 B.C." ] ;
                    rdf:value <http://.../Die_2> ] ;
            nmo:hasIssuer [ rdf:value <http://.../issuer_A>,
                        <http://.../issuer_B>,
                        <http://.../issuer_C> ] ;
            nmo:hasMint [ crm:P89_falls_within nm:asia_minor ;
                    rdf:value <nm:comama> ] ] .
```

4.3 RDF-Star

RDF-star is an extension of the conceptual data model and concrete syntaxes of RDF. It provides an alternative to standard RDF reification by allowing the creation of short triples (referred to as RDF star triples) that contain other triples in their subject and/or object position (Arndt, et al., 2022). There is no RDF/XML extension supporting RDF-Star yet, but there is one for Turtle, and other serialization syntaxes like N-Triples, N-Quads, TriG and a query language SPARQL-Star. In this section we will only be using Turtle-Star to propose a solution for modeling uncertainty.

With RDF-Star, a triple can be used as an asserted triple, a quoted triple, or both. An asserted triple is what we been using throughout this work, a tuple of three components: subject, predicate, and object. A quoted triple is a tuple that can have another quoted triple as its subject or object component. Example of both triple types in Turtle-Star:

- asserted triple: `Coin_1 nmo:hasMint nm:comama.`
- quoted triple: `<<:Coin_1 nmo:hasMint nm:comama>> a rdf:Statement`

To represent an uncertain statement like 'Coin_2 minted in Comama, is uncertain' in Turtle-Star we can use quoted-triples as follows:

```
<<:Coin_2 nmo:hasMint nm:comama>> un:hasUncertainty nm:uncertain_value .
```

The triple that holds the statement 'Coin_2 minted in comama' is in the subject position of the triple that completes with 'has uncertainty, uncertain value'. **Figure 4.** ***19*** shows the graph representation of this example:

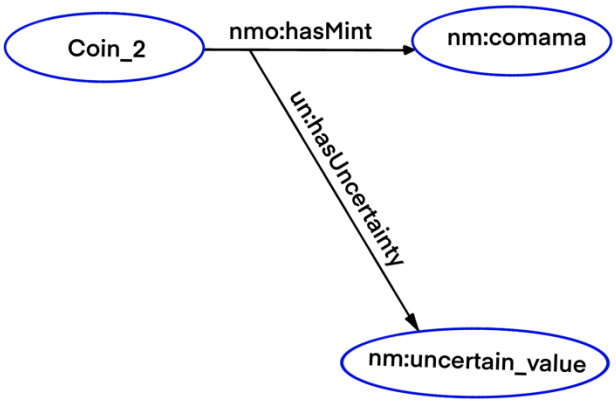


Figure 4. 19: Turtle-Star in graph representation.

In comparison to the Turtle-format and graph representation of RDF-Reification and of solution 2, both for the same example, shows the renunciation of using empty nodes with RDF-Star:

RDF-Reification:

```
:Coin_2 nmo:hasMint nm:comama .

[] a rdf:Statement ;
    un:hasUncertainty nm:uncertain_value ;
    rdf:object nm:comama ;
    rdf:predicate nmo:hasMint ;
    rdf:subject :Coin_2  .
```

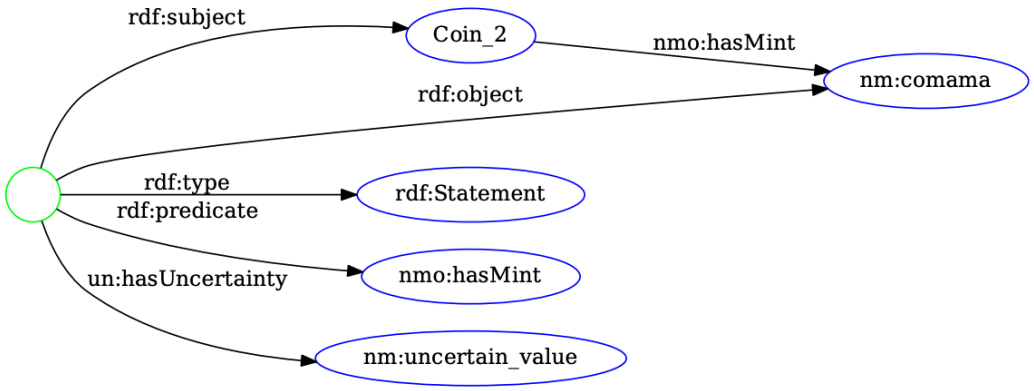


Figure 4. 20: Graph representation of RDF Reification.

Solution 2:

```
:Coin_2 nmo:hasMint [ un:hasUncertainty nm:uncertain_value ;
            rdf:value nm:comama ] .
```

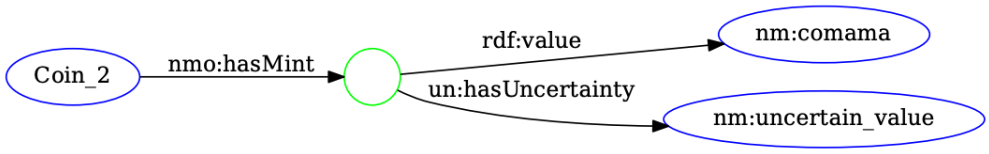


Figure 4. 21: Graph representation of Solution 2.

To give one more example of Coin_2 showing a portrait of 'Titus' is certain, but it is uncertain if this coin minted in 'Comama':

```
:Coin_2 nmo:hasPortraie nm:titus.
```

```
<< :Coin_2 nmo:hasMint nm:comama>> a edtfo:UncertainStatement .
```

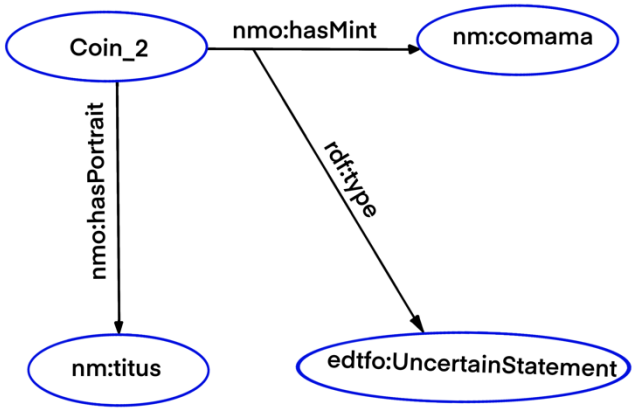


Figure 4. 22: Using asserted triple for certain features, and quoted triple to refer to a statement being uncertain.

4.4 Summary

Information about coins as archaeological objects is complex and, in many cases, involve uncertainty. But even uncertain information in this matter can be of great value, and here emerges the need of a way to model this information and distinguish it from reliable ones. In this chapter we introduced eight approaches to modeling uncertainty in the field of numismatics. The first two approaches (S1 and S2) were given as RDF graphs, so we had no difficulty in writing them in RDF/XML format. The next three solutions (S3-S5) were discussions suggesting using this property and that class. It was not easy translating these into graphs and into RDF/XML. We contacted Dr. Doerr (FOURTH Greece, Chair, CIDOC-CRM Special Interest Group) to ask for his opinion of our interpretation of solutions (S3-S5), which we brought up while presenting the solutions.

We believe that RDF-Star can offer a good and compact solution to modeling uncertainties. It offers the possibility to express uncertainty with much less triples compared to RDF-Reification and with the dispense of using blank-nodes. Since there exists not an RDF/XML extension that supports RDF-Star, we must end our short journey with it at this point.

In order to test these eight solutions on large datasets with 100K- to 1M-coins, we created an application that automatically generates an RDF/XML file from a given Excel/CSV file and a selected one of the eight solutions, which we implemented within the application. The solutions will then be tested for query -runtime and -memory-consumption in retrieving certain and uncertain data. The next chapter handles creating the application explaining its most important components for automating the generation of the RDF/XML file based on the solutions for uncertainty discussed in this chapter.

The Excel/CSV file contains the coins as subjects, each with property, property value and a field indicating whether the property is uncertain.

5. Implementation of the Uncertainty Solutions as Web Application

In the former chapter we presented eight modeling solutions for dealing with uncertainties in determining features of ancient coins. To explain these solutions, we used an example of two coins, one of which minted in Comama is certain and for the other this is uncertain. Such example with only two entries can easily be written manually, yet it is far away from reality where thousands (or maybe hundreds of thousands) of coins are dealt with that may contain uncertainties. Therefore, we decided to create a program that generates an RDF/XML file from the input data (in Excel/CSV) according to one of the eight proposed solutions.

As first approach, we created a CSV-file of 500-coins with few Nomisam-properties and fake randomly generated property-values, in **Figure 5. *1***. For the uncertainty we created a separate array that includes an uncertainty-level $i, for\ [0 < i \leq 1]$ to 50% of the property-values of few properties. We then implemented one solution statically depending on our example.

subject	hasAxis	hasLegend	hasMint
Coin_0	12	Legend119	Mint42
Coin_1	9	Legend154	Mint58
Coin_2	15	Legend62	Mint26
Coin_3	15	Legend69	Mint34
Coin_4	12	Legend190	Mint11
Coin_5	12	Legend37	Mint56
Coin_6	14	Legend66	Mint14
Coin_7	13	Legend75	Mint2
Coin_8	11	Legend116	Mint45
Coin_9	9	Legend118	Mint12
Coin_10	11	Legend146	Mint36

Figure 5. 1: The generated CSV-file of 500-coins with few properties and random property-values, first approach.

At this point we had no knowledge of the method used by experts to document such data of coins in tables, nor of how to express uncertainty in a particular property. After we inquired about the method used, we changed the CSV-file accordingly and modified the implementation of the solutions to be dynamic.

In this chapter we introduce SAUN, the web application created to convert the input CSV/or Excel file of coins to an RDF/XML file, based on the eight uncertainty modeling solutions. We will explain the purpose for the used technologies, the form of the input file, the dynamic implementation of the solutions (based on an example), and the steps for generating the output RDF/XML file.

5.1 Requirements

SAUN stands for 'Serialization Application for Uncertainty in Numismatics'. It is based on the Model View Controller (MVC) architecture with a graphical user interface that receives an 'excel' or a 'csv' file and generates, based on a selected solution of modeling uncertainties, an RDF/XML file —later also included the Turtle format — that is compatible to the W3C standards, illustrated in **Figure 5.** *2*. With the help of the Graphviz tool, SAUN can also visualize the RDF/XML output file as RDF graph. It has a Fuseki server embedded with extended memory for an automatic data transfer and queries running.

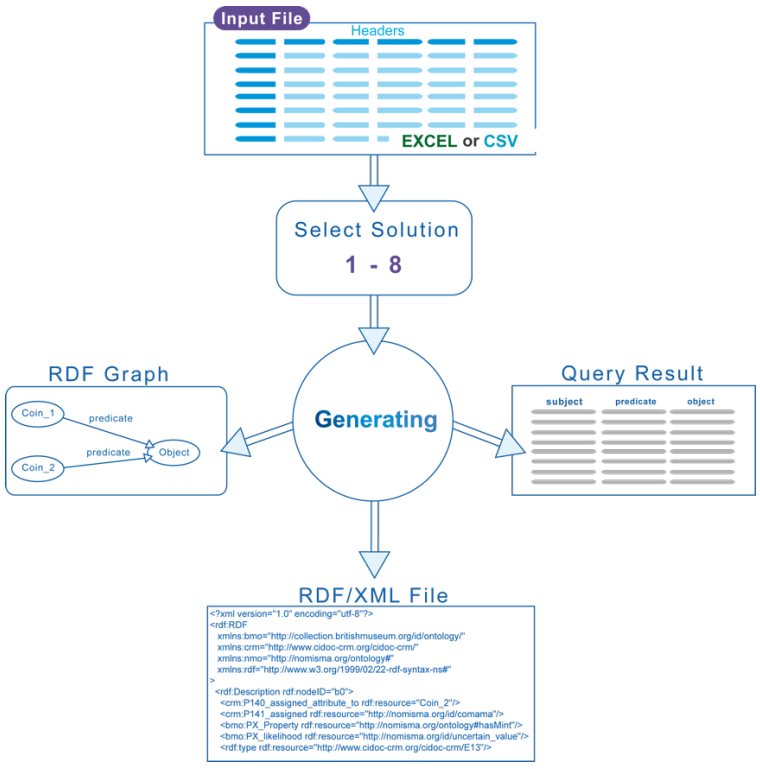
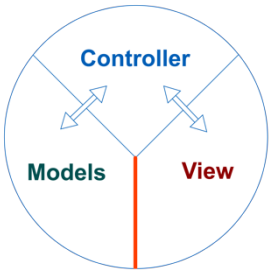


Figure 5. 2: A basic illustration of the main goal of SAUN, from an input of excel/csv data, generate an output of RDF/XML file based on the uncertainty modeling approaches.

5.1.1 MVC Architecture



For the implementation of SAUN, we used the MVC architecture. The Model layer should have all the functional component and the data of the application, and it must have no connection to the View layer. In our implementation the Model layer is equivalent to the python classes and database. We only used HTML pages, and the View layer is for displaying the components of those pages. The pages are separated in the template directory, and the Python classes are responsible for processing the data.

The controller takes the role of controlling the communication between the View and the Model. In our implementation we created a class app.py, which is the only class that uses the flask libraries, and any data transportation (between Model and View) must be within this class. The user has no direct access to the model and data transferring is only possible through the controller using the methods GET and POST.

5.1.2 Input-File

After we informed about the form of the input file, how the properties are included and the uncertainties are expressed, we created a simple Excel file of two coins as shown in **Figure 5. *3*** below. The headers include the full URI of the properties, each property could have a second column referring to the uncertainty in this particular property, which contains of the property name with the expression 'hasUncertainty' separated by an asterisk.

Subject	http://nomisma.org/ontology#hasMint	hasMint*hasUncertainty
Coin_1	http://nomisma.org/id/comama	
Coin_2	http://nomisma.org/id/comama	1

Figure 5. 3: The generated Excel-file of 2-coins with one property and a column for representing the uncertainty in that property for each coin, second approach.

The header in the uncertainty must be unique to each property, which is the reason for using the property name here. The asterisk serves the same purpose and for easier identification

when reading and separating the headers. The '1' under 'hasMint*hasUncertainty' is an indicator for being the property 'hasMint' with the property-value 'http://nomisma.org/is/comama'—or as we used to refer to it with short form 'nm:comama'—is uncertain for 'Coin_2'. For 'Coin_1' this is certain because the field opposite is empty. We then replaced the '1' in the uncertainty column with a random number $i, where\ [0 < i \leq 1]$, to be the uncertainty/confidence level, used e.g., in solution 4.

5.1.3 Technologies Used

The following technologies are used in creating and running SAUN.

Python, PyCharm and Java-14

SAUN is running on python 3.10, which can be downloaded from python.org. However, it is also runnable on python 3.6 or higher. After running the installation, it is important to add python path in the Environment variable. We used Python because it has many useful libraries and good documentation. It is simple to work with and code changes can be easily made.

After we successfully installed Python, we need to write our code in an optimal environment. There are many to consider, we recommend using Visual Studio Code or PyCharm.

To be able to run the Fuseki server we need to install java. The server requires Java 14.02, which can be downloaded from oracle.com under Java SE 14 Archive-Java SE Development Kit 14.0.2 Windows x64 Installer. After running the installation, we need to check with the cmd command java –version, that java 14.0.2 is in the system, because a different version of Java may cause issues. In case Java is not recognized, we ensure it is added to the environment variables path.

Next is to install the latest version of the following libraries: `Flask, JSON, os, time, csv, pandas, rdflib`, and `NumPy`. But before that, we run the command `pip install --upgrade pip`. In case of a permission error, we add the keyword `--user` at the end. This will allow the command to run the installation and avoid the user permission. During the code implementation, we wanted to use a tool to easily connect the code to the

UI. Flask offers a very easy way to create a web application and has very good documentation that describes everything we need for our implementation.

Now that all libraries for the program are installed, we ensure that everything is working correctly by running the class `app.py`. The Flask server should be started on the internal address 127.0.0.1:5000. When clicking on the address SAUN's index-page should appear.

Apache Jena Fuseki Server

In our implementation, we appended the Fuseki server to the project file, to be able to upload data and run queries automatically.

Further requirements

To use SAUN, and for further development the following requirements are a necessity:

1. Installing Python 3.10 or 3.9
2. MySQL, MySQL Workbench.
3. Windows Machine with:
    a. CPU (intel core i3 – i9) up 2016 or AMD Ryzen 3 (G).
    b. 16GB Ram 2400mhz or higher
    c. SSD 128 or higher
4. Java Version 14.02
5. Fuseki Server
6. Graphviz

5.2 Deploying SAUN

In this section we describe the process of preparing the input data, implementing the solutions within SAUN and generating the output file.

5.2.1 Preparing the Input data

Preparing the data occurs within the class 'readAndSeparate'. First is to read the data file, which is either an Excel- or a CSV- file. Any other file type will lead to an error, and a reload of the HTML-index page. Reading the data file can be done in many ways, for an easier approach, we use `csv`-library for reading the csv-files and `pandas` for reading the Excel-files and writing both CSV- and Excel-files. The inner classes and functions only work with CSV-format, therefore, after selecting the input data file and the modeling solution to be generated, the format of the input will be checked. If the input file is in Excel-format, it will be converted to CSV and rewritten with the same name in the *csv* directory.

Next step is to read the headers within the CSV-file and separate the ones that contain the actual properties from the uncertain headers, which contain the propertyName*hasUncertainty. Each group will then be stored with their data in a separate file, as illustrated in **Figure 5.** ***4*** below.

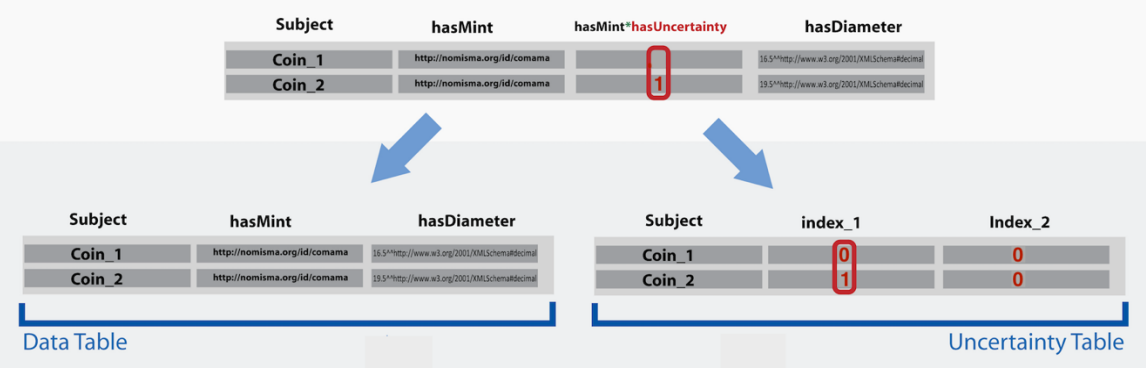


Figure 5. 4: Separating the input data into two groups, one contains the actual properties/property-values, and the other contains the columns with the headers with propertyName*hasUncertainty.

Note, that the uncertainty headers must contain the symbol '*' with the property name, e.g., '*hasMint' or uncertain*hasMint. In case of a misspelling or the name of the uncertain header

dose not match the actual name of the property, for example: the uncertainty header is ‘hasProductionPlace*hasUncertainty’ and the actual property name is ‘hasMint’, or ‘hasClosingDate*hasUncertainty’, while the actual property name is ‘hasEndDate’. In this case, the property will be falsely interpreted as certain.

This process of separating the input data is executed one time for each uploaded data file, no matter how many solutions the user decides to generate. This outcomes in reducing the total runtime of the generation process of further solutions.

5.2.2 Implementing the Uncertainty Modeling Solutions

First thing is to define the namespaces being used in the solutions. We found no other choice here but to statically code this process. What happens here is that the application identifies each class/property used in creating a solution through its URI. It inspects the URI for specific keywords. For instance, to identify that the Nomisma-Ontology was used it looks for the keywords 'nomisma' and 'ontology' and use the letters ‘nmo’ as prefix. It will then store the URI within an array containing of the: Namespace-name, Namespace-prefix, and the property-/class-name. In case, for example, the use of the Nomisma-ontology was determined, the array will contain the following:

```
['http://nomisma.org/ontology#', nmo, 'property-name']
```

We only define a namespace needed for a specific solution in the code section of that solution. After defining the namespace, we append them to the RDF-Graph of the solution. When retrieving the selected solution, we loop the input data that has been separated into two tables data table and uncertainty table. First, we loop every subject *i* of *n*, which always has index 0, and every subject has a row where its property values are stored, and we need to iterate each of these values in a row. So, we defined a second for-loop and set its range to the length of the columns *m*.

Subjects	hasProperty_1	hasProperty_2	hasProperty_3	…	hasProperty_m
subject_1	property_value	property_value	property_value	.	property_value
subject _2	.	.	.	.	.
	.	.	.	.	.
subject _n	property_value	property_value	property_value	.	property_value

For each inner iteration:

1. Check if a property value in the subject(row) is non or equal empty string. If so, means that the subject in this column do not have any value. We continue the loop without adding anything to the graph.

2. The Program will check the value of the respective cell in the uncertain array, if it is set to 0, means that the input value of the data is certain. Otherwise, it is uncertain.

In either way, whether certain or uncertain, three cases must be considered in order to attach the entry (subject, predicate, object) to the graph in the correct way:

Case 1: If the cell includes the substring http or https and not XMLSchema then the entry must be added as a URI-Reference to the graph. The property value that contains an *XMLSchema* will be skipped here, because it represents a datatype and not part of the actual property value.

Subjects	http://nomisma.org/ontology#hasMint	http://nomisma.org/ontology#hasWidth
subject_1	http://nomisma.org/id/comama	12.3^^ http://www.w3.org/2001/XMLSchema#decimal

Case 2: if the cell includes the substring XMLSchema means it includes a property value with its datatype and it should be included as a literal not as URIRef.

The difference here to Case 1, is when the property-value was a literal (string, numeric or date), with datatype (e.g., `2.3^^http://www.w3.org/2000/01/rdf-schema#decimal`). In this case, we return a list with first element "property value" and second element "datatype" as follows:

```
[2.3, 'http://www.w3.org/2000/01/rdf-schema#decimal']
```

Case 3: If the cell does not contain a URI or a specific data type (plain text, number, date with characters), it is inserted as a literal.

The above three cases were to build the graph with the certain entries (where the position in the uncertainty table contains '*nan*'). To add the uncertain entries (those with position in the uncertainty table is not empty), we still need these three cases but with some modifications, which is, acording to solution 2, using blank-node (*BNode*) directly in the property path. This

means, that the blank-node will be in the object position (subject-property-`BNode`) in line 1, then in the subject position in lines 2 & 3 where it has the property '`hasUncertainty`' with property value '*`uncertain_value`*' in line 2. And the property '*`value`*' with property-value retrieved from the data array.

Case 1_Uncertain Entries:

```
1. G.add((URIRef(con[0]), eval(n_node[inncon]), BNode('b' +
   str(nodeCounter))))
2. G.add((BNode('b' + str(nodeCounter)), UN['hasUncertainty'],
   NM['uncertain_value']))
3. G.add((BNode('b' + str(nodeCounter)), RDF['value'],
   URIRef(con[inncon])))
4. nodeCounter += 1
```

To make each blank-node unique, we define a counter and initialize it with 0 (`nodeCounter` in line 4). Then we append the counter value as string to the blank-node's name `'b'`. This returns us `'b0'` for the first uncertain triple, `'b1'` for the second and so on untel the last uncertain triple is added.

Cases 2 and 3: Aplaying same proceger as in *Case 1_Uncertain Entries* (lines 1, 2 & 4). In Case 2 line 3, we must replace the *`URIRef()`* in the object position with *`Literal()`* and use the result of the function *`valueLinkSeperator()`*. And in Case 3 line 3, we just add the value of the cell as it is.

Case 2_ Uncertain Entries:

```
3. G.add((BNode('b' + str(nodeCounter)), RDF['value'],
       Literal(valueLinkSeperator(con[inncon])[0],
          datatype=valueLinkSeperator(con[inncon])[1])))
```

Case 3_ Uncertain Entries:

```
3. G.add((BNode('b' + str(nodeCounter)), RDF['value'],
   Literal(con[inncon])))
```

Finally, we define a new directory name '*`rdf`*', and use it to save the solution model (solution 2 in this case) under name *'`modelGraphDynamic_G2`'*.

Generating the Serialization Syntaxes RDF/XML and Turtle

Based on the selected solution we first define the required namespaces as explained in the former section. Then the input data is split into two tables; 'Data Table' and 'Uncertainty

Table' as described above. During generation, the corresponding cell in the 'Uncertainty Table' is checked for emptiness for each cell in the 'Data Table'. If empty, the cell in the 'Data Table' holds certain data, otherwise uncertain and will then be modelled according to the selected uncertainty solution. After iterating all cells, the process is complete, and the generated file will download automatically.

When Prof. Heath suggested that the Turtle syntax might be easier to read than the RDF/XML, regarding the Nomisma-Cookbook. We decided to also add Turtle here as another serialization option.

RDF-Graph Visualizer

Under Graph, the recent generated solutions can be displayed in their RDF graph representation. For now, this tool can only display the generated RDF/XML solutions in RDF graphs. It can though be further modified to be capable of displaying any RDF/XML or Turtle snippets.

SPARQL-Query and Further Features

SAUN also has a SPARQL request page. This page allows the user to view a list of all triples of the generated solution and search them for specific keywords, like search for specific coin ID, property, or property value.

It also has a Fuseki Control page where the user can change the ‘dataset name’ that holds the RDF file to be queried, the address, the port, and the memory size of the server.

Challenges

First challenge was to determine the correct technologies, that make it possible to convert out ideas to a functional application.

It was not possible to use rdflib to deploy every cell of the input file to a rdf/xml file. Therefore, we used the function eval() in python to build the code as string and then push it as a code in python. This however do not work with any URI that comes in the subject, object, or datatype position, e.g., when using the Nomisma-IDs, or XML-Schema. Because the abbreviated approach of using namespace-prefixes is not meant to be used in those positions,

and to use it, the namespaces must be defined using XML-Base or XML-Entities. We were only able find a way to insert a namespace as 'XML-Base' (Rörtgen, 2020), but we did not apply it since our coins are fake and not part of any dataset. We could not define any namespace using xml-entities, therefore in our implementation, the RDF/XML output file do not include, for example, the namespace-string of the Nomisma-IDs, which comes in the object position as a property-value, nor that of XML-Schema, that used to define the object-datatype.

Also, analysing, for example, a property URI and find an approach to making it dynamically recognizable by the application was challenging.

Vision

We can think of SAUN being expanded in the future to include more uncertainty modeling solution, for this topic is yet unclosed and there may arise new solution suggestions to be compared and tested. While creating implementing those eight solutions we worked with limited number of ontologies and the only static part of the implementation was to define the prefixes to be used for each of those ontologies. To make this process of identifying the used ontologies dynamic we can imagine applying a deep clustering algorithm to cluster the input headers and content. The algorithm would then cluster URIs without the term name (i.e., without the property- or class- name), and generate a prefix to each cluster.

5.3 Summary

In this chapter we presented a web application, Serializing Application for Uncertainty in Numismatic (SAUN). This application allows the user to model their input data —which is of the format Excel or CSV—according to one of the eight modeling solutions for uncertainty. SAUN automatically converts the input file into one of a selected serialization syntax (RDF/XML or Turtle), view the graph representation of the converted data, display- and search through all triples of the converted data.

To compare the eight-uncertainty modeling solutions we are going to run a test measuring the runtime and memory consumption querying data from those solutions. For this purpose, we will be using SAUN to generate four Excel-files of different number of entries. The test procedure and results are presented in the next chapter.

6. Evaluation of the Uncertainty Solution

In this chapter we will perform two types of tests on the eight uncertainty modeling solutions. One test is for measuring the time running various SPARQL queries on each model. We performed an automated test that runs each query five times on each solution and stores the results in an Excel-file. The other test is for measuring the memory usage by each query. For this, we run a manual test and documented both query runtime and memory usage.

6.1 Runtime and Memory-Usage Measurement

To perform the test, we generated four Excel-files with: 2 entries, 100,000 entries, 500,000 entries and 1 million entries. Each file includes one coin minted in Comama is certain, whereas for the rest of the coins this is uncertain, example in **Figure 6. *1***. The values under 'hasMint*hasUncertainty' indicating the certainty (value = 0), or uncertainty (0 < value ≤ 1), only coin_0 is certain. The values for uncertainty are randomly generated and meant to indicate the uncertainty/confidence level.

	A	B	C
1	**Subject**	**http://nomisma.org/ontology#hasMint**	**hasMint*hasUncertainty**
2	coin_0	http://nomisma.org/id/comama	0
3	coin_1	http://nomisma.org/id/comama	0.62
4	coin_2	http://nomisma.org/id/comama	0.82
5	coin_3	http://nomisma.org/id/comama	0.66
6	coin_4	http://nomisma.org/id/comama	0.88
7	coin_5	http://nomisma.org/id/comama	0.73
8	coin_6	http://nomisma.org/id/comama	0.67
9	coin_7	http://nomisma.org/id/comama	0.81
10	coin_8	http://nomisma.org/id/comama	0.54
11	coin_9	http://nomisma.org/id/comama	0.68
12	coin_10	http://nomisma.org/id/comama	0.55

Figure 6. 1: Screenshot an Excel file generated for the test.

We also wrote five SPARQL queries for each solution, the queries are as follows:

Q1. *Returns coins with certain mint only and the value it holds. (Coin_0, nm:comama)*

Q2. *Returns coins with mint whether certain or uncertain. (All coins)*

Q3. *Returns coins with certain or uncertain Mint and the value it holds. (All coins, each with nm:comama)*

Q4. *Returns coins with certain Mint only, without the value it holds. (Coin_0)*

Q5. *Returns coins with uncertain properties, the value they hold, and the uncertainty/confidence level (for S3, S5). (All coins with uncertain mint, each with the property 'nmo:haMint',property-value 'nm:comama' and a numeric value l, $l \in [0,1]$, for the uncertainty/confidence level)*

Q5. *Returns coins with uncertain properties and the value they hold (for S1, S2, S4, S6-S8, since they do not have uncertainty/confidence level). (All coins with certain mint, each with the property 'nmo:haMint' and property-value 'nm:comama')*

Some queries are identical for different solutions like, Q1 in (S2, S4, S5 & S7) and Q2 in (S1, S3 & S6), which is the following:

```
SELECT ?Coin WHERE {?Coin nmo:hasMint nm:comama. }
```

All queries can be found in **Appendix 3**.

We listed the queries, each as string, inside a class called 'testQueries' and created a variable called 'PREFIX' that contain, also as string, all prefixes required for all queries. This variable is appended to each query at run.

In the automated test the queries are pushed to the fuseki server, and the result retrieved in JSON format. Each query is executed five times for every solution, the time by each execution is measured and retrieved individually, then the average and median of all five execution iterations are calculated. **Table 6. *1*** shows the test results of the runtime of the five queries (0 - 4) on solution 1.

Table 6. 1: The test results of the function dynamicTest() measuring the runtime of the five queries (0 - 4) on solution 1 on the dataset of 100K entries.

Queries	Time per Run for S1	Average Time	Medien
0	[1.23, 0.39, 0.36, 0.34, 0.39]	0.54	0.39
1	[0.42, 0.36, 0.36, 0.35, 0.37]	0.37	0.36
2	[0.64, 0.63, 0.59, 0.64, 0.60]	0.62	0.63
3	[0.43, 0.38, 0.36, 0.38, 0.36]	0.38	0.38
4	[1.41, 1.38, 1.40, 1.41, 1.35]	1.39	1.40

Before we present the test results of all solutions, we would like first to give a reminder of the eight solutions.

Solutions:

S1. *RDF-Reification with CIDOC-CRM E13, P140, P141, and BMO Likelihood.*

S2. *Adding uncertainty directly to the property path with UN hasUncertainty.*

S3. *Assigning reliability instead of uncertainty.*

S4. *Using CRMinf to assign belief values.*

S5. *Expanding the CRM properties with .2 properties.*

S6. *RDF-Reification with EDTFO class UncertainStatement.*

S7. *Based on S2, with EDTFO class ApproximateStatement instead of UN hasUncertainty.*

S8. *Adding all uncertain properties within UN UN hasUncertainty.*

We run the automatic test on all solutions generated with the dataset of 100K entries, that has one coin with Mint=comama (certain) and the rest are coins with Mint=comama (uncertain). The test result of each solution is stored in a table like **Table 6.** ***1***. We collected the average of each query, for each solution and listed them in **Table 6.** ***2***. Reading the test result we can observe the following:

- Retrieving coins with certain values with Q1 is a lot faster by solutions S2, S4, S5, S7 of (*0.01s each*) and S8 *(0.03s)*. This remains fairly stable with Q4 which is similar to Q1, except it do not return the property value ‘nm:comama’. The slowest retrieval of coins with certain values here compared to the rest of the solutions is S6 & S3 of *(0.54s & 0.59s)* in Q1 and of *(0.56s & 0.57s)* in Q4. The result of S1 was *(0.39s & 0.38s)* with Q1 & Q4 respectively.
- In retrieving coins with certain or uncertain mint Q2 by S1 *(0.36s)*, S6 *(0.38s)* & S3 *(0.40s)*, and that with the value they hold Q3 with S1 *(0.63s)*, S3 *(0.66s)* & S6 *(0.67s)*, was the fastest compared to the rest of the solutions. Next in line comes S5 *(0.48s)*, S2 *(0.59s)* S4 *(0.66s)* & S7 *(0.68s)* with Q2, and S7 *(0.78s)*, S5 *(0.83s)* S4 *(1.04s)* & S2 *(1.35s)* with Q3. The slowest with these two queries compared to the other solutions was S8 *(0.91s), (1.56s)* with Q2, Q3 respectively.

- Retrieving uncertain properties with the value they hold and if exists a confidence/uncertainty level with Q5, was fastest by solutions S8 *(0.80s)*. With S2 S7 was *(0.82s each)* were slightly better compared to solutions S5 *(0.95s)* & S4 *(1.02s)*. With S1 & S6 it was *(1.40s & 1.43s)*, and the slowest retrieval was with S3 *(1.95s)*.

With S1, S3 & S6 retrieving uncertain data was faster, while querying certain data is significantly faster with S2, S4, S5, S7 & S8. ***Table 6. 3*** lists queries that are exactly the same by some solutions, and ***Table 6. 4*** shows which solutions have the same query Qi.

Aside from the queries, S8 has the smallest number of triples of less that 200k for the 100k entries. Followed by S2 (S7, *which is based on S2*) & S5 with each of slightly less than 300k triples. S3 has the largest number of triples of about 900k.

Table 6. 2: The average/median runtime of running the five queries automatically on all solution 1-8.

Solutions + (Number of triples)	**Q1** certain + value	**Q2** certain & uncertain	**Q3** certain & uncertain + value	**Q4** certain	**Q5** uncertain + property, value & confidence level
S1 (599,995)	0.54 / 0.39	0.37 / 0.36	0.62 / 0.63	0.38 / 0.38	1.39 / 1.40
S2 (299,998)	0.01 / 0.01	0.67 / 0.59	1.45 / 1.35	0.06 / 0.06	0.82 / 0.82
S3 (899,996)	0.75 / 0.59	0.39 / 0.40	0.65 / 0.66	0.62 / 0.57	1.98 / 1.95
S4 (699,994)	0.01 / 0.01	0.81 / 0.66	1.04 / 1.04	0.08 / 0.06	1.00 / 1.02
S5 (299,999)	0.01 / 0.01	0.56 / 0.48	0.89 / 0.83	0.06 / 0.05	0.97 / 0.95
S6 (499,996)	0.67 / 0.54	0.40 / 0.38	0.66 / 0.67	0.58 / 0.56	1.43 / 1.43
S7 (299,998)	0.01 / 0.01	0.66 / 0.68	0.80 / 0.78	0.05 / 0.05	0.80 / 0.82
S8 (199,999)	0.05 / 0.03	0.98 / 0.91	1.55 / 1.56	0.05 / 0.04	0.80 / 0.80

Table 6. 3: Queries that are equal in some solutions.

Q1	**Q2**
S2, S4, S5, S7	S1, S3, S6

Table 6. 4: Different solutions that share the same queries.

S2, S4, S5, S7	**S1, S3, S6**
Q1, Q4	Q2, Q3

With the automatic test we did not skip the cold start (Stamer, 2020), which occurs on the first run of the query after the data has been uploaded to the server. Also, we could not include the memory consumption to the automatic test, therefore, we performed a manual test. In the manual test we run each query Q1-Q5 for all solutions S1-S8 and on each dataset 2-coins – 1M-coins were manually about 10 times and the result taken is the most frequently occurring. The result of this test is displayed in ***Table 6. 5***.

Example of the Test-Process:

1. Generate S1-S8 for dataset X.
2. Turn Fuseki off/on and upload a solution Si.
3. Run query Qij ten-times and take the most frequent result (run-time & memory usage).
4. Repeat step 3 for all queries.
5. Go to step 2.

The runtime for the queries Q1-Q5 was measured in seconds as following:

$$Run\ Time\ Q_i = Time_{endQ_i} - Time_{startQ_i}$$

Memory usage (cache) for each query was measured via Python library ***memory_profiler***.

Table 6. 5: Test results of the runtime and memory consumption of the five queries on each solution (1 - 8). Each solution was generated for four data sizes; with 2 coins, 100K coins, 500K coins and 1M coins, where each contains 1 coin with certain mint in Comama, and for the rest this is uncertain.

	Number of Entries	Number of Triple	Q1		Q2		Q3		Q4		Q5	
			Run-Time/ Sec.	Cache MiB	Run-Time/ Sec.	Cache MiB	Run-Time/ Sec.	Cache MiB	Run-Time/ Sec.	Cache MiB	Run-Time/ Sec.	Cache MiB
S1	2	7	0.14	59.9	0.14	59.8	0.14	59.7	0.14	59.6	0.15	59.6
	100,000	599,995	0.42	59.6	0.40	122.9	0.68	159.8	0.41	59.6	1.55	199.7
	500,000	2,999,995	2.05	59.6	2.13	373.0	3.68	556.8	2.14	59.5	7.50	748.9
	1,000,000	5,999,995	4.58	59.7	4.12	682.8	6.85	1052.5	4.36	59.7	15.01	1437.4
S2	2	4	0.14	59.6	0.14	59.7	0.14	59.9	0.15	71.2	0.14	59.6
	100,000	299,998	0.14	59.8	0.72	122.8	1.38	159.2	0.20	71.2	0.96	159.9
	500,000	1,499,998	0.15	59.7	3.27	372.7	6.78	556.7	0.47	70.8	4.73	556.8
	1,000,000	2,999,998	**0.14**	59.7	6.17	682.8	12.96	1052.8	0.74	71.0	**8.77**	1054.1
S3	2	11	0.14	59.6	0.14	59.8	0.14	59.8	0.14	59.6	0.14	59.9
	100,000	899,992	0.67	59.5	0.48	123.1	0.74	160.2	0.71	59.7	1.91	167.0
	500,000	4,499,992	3.14	59.6	2.16	373.0	3.53	556.4	3.45	59.7	10.06	593.8
	1,000,000	8,999,992	6.75	59.7	4.05	683.2	6.96	1052.4	7.22	59.7	20.79	1127.8
S4	2	8	0.14	59.7	0.14	59.7	0.14	59.8	0.14	59.6	0.14	59.6
	100,000	699,994	0.14	59.9	0.78	122.9	1.14	159.6	0.19	59.7	1.12	160.5
	500,000	3,499,994	0.14	59.7	3.52	371.4	5.53	556.3	0.39	59.6	5.48	556.4
	1,000,000	6,999,994	**0.19**	59.6	7.06	683.3	10.66	1053.2	0.72	59.8	10.44	1054.2
S5	2	4	0.15	59.9	0.15	59.9	0.15	59.8	0.15	71.3	0.14	59.6
	100,000	299,998	0.14	59.8	0.62	122.8	0.97	159.6	0.21	71.3	1.08	166.6
	500,000	1,499,998	0.15	59.8	2.80	374.9	4.68	556.0	0.39	71.4	5.35	594.1
	1000,000	2,999,998	**0.14**	59.8	5.19	682.9	8.78	1052.7	0.73	71.3	10.09	1127.9
S6	2	6	0.14	59.7	0.14	59.7	0.14	59.6	0.14	59.7	0.14	59.9
	100,000	499,996	0.76	59.7	0.49	122.1	0.78	159.3	0.74	59.8	1.48	198.5
	500,000	2,499,996	3.17	59.7	2.21	372.3	3.76	556.7	3.20	59.6	7.28	748.7
	1,000,000	4,999,996	5.94	59.7	3.99	683.2	6.58	1052.4	6.20	59.8	13.68	1436.6
S7	2	4	0.14	59.6	0.14	59.7	0.15	71.3	0.15	71.5	0.14	59.6
	100,000	299,998	0.14	59.7	0.72	122.8	1.01	171.6	0.21	71.4	0.97	160.3
	500,000	1,499,998	0.15	59.7	3.22	373.9	4.85	568.3	0.49	71.3	4.65	556.2
	1,000,000	2,999,998	**0.14**	59.7	6.44	682.7	9.04	1064.7	0.65	71.3	**8.99**	1052.3
S8	2	3	0.16	71.5	0.15	59.6	0.16	59.7	0.16	71.4	0.18	59.7
	100,000	199,999	0.18	71.3	1.04	123.0	1.67	160.5	0.20	71.4	0.94	159.5
	500,000	999,999	0.27	71.3	5.69	371.5	8.87	556.6	0.35	71.2	4.81	556.3
	1,000,000	1,999,999	0.40	71.5	9.87	682.8	15.41	1052.9	0.51	71.1	**8.73**	1052.1

Observation, focusing on the dataset of 1M entries:

From ***Table 6. 5*** we can see that the size of the dataset do not affect the query runtime for retrieving certain data, with Q1 & Q4, at solutions S2, S4, S5 and S7. Retrieving certain data is significantly faster at these solutions (Q1 in green), even with large dataset of 1M entries, comparing with the rest of the solutions. While S1, S3 & S6 have lower query runtime in retrieving both certain and uncertain properties with Q2 & Q3, and S8 with the highest query runtime. Finally, the runtime querying uncertain properties with the value they hold, and if exists a confidence/uncertainty level, using Q5 was the lowest (Q5 in green) with S2, S7 & S8 at the dataset of 1M entries.

The memory consumption remained the same with each solution, increasing with larger dataset.

6.2 Evaluation

The automatic test preformed on all eight solutions generated with SAUN using the dataset of 100k coins, with only one coin minted in comama is certain and for the rest this is uncertain. According to the combined mean runtime of all five requests, S5 took first place. Followed by S7 and S4, which both based on S2 suggesting adding uncertainty directly to the property path. We also compared the solutions according to number of triples, taking the example of only two coins (Coin_1 minted in comama is certain, and for Coin_2 this is uncertain). The results of these comparisons are displayed in **Table 6. *6*** and **Table 6. *7***. We did not compare the solutions for memory consumption in querying data, for it is equal in all solutions and it depends on the size of data.

Table 6. 6: Ranking according to summation of the median runtime of all five queries, at the dataset of 100k coins.

Ranking	Solution	Total Median-Runtime in all Queries (in seconds)
1	S5	2.32
2	S7	2.34
3	S4	2.79
4	S2	2.83
5	S1	3.16
6	S8	3.34
7	S6	3.56
8	S3	4.17

Table 6. 7: Ranking according to number of triples, at the dataset of 2 coins, one of which has uncertain property.

Ranking	Solution	Number of Triples
1	S8	3
2	S2	4
3	S7	4
4	S5	4
5	S6	6
6	S1	7
7	S4	8
8	S3	11

Our Experience

S2 was easier to work with, it is not complicated, requires little number of triples and there is no need to define extra new properties for uncertain cases like S5. We used the idea presented in this solution, which is adding uncertainty directly to the property path, in modeling other solutions that all ranked better than those using RDF reification. With S2 one has multiple options to express that a piece of information, example:
A statement of (Subject Predicate Object) has uncertainty, which then can refer to:

- type of uncertainty, as defined by the W3C (Laskey, et al., 2008),
- one of the pre-defined Nomisma-IDs for uncertainty,
- specific uncertainty level, in percentage or degree of likelihood.

We made S7 based on S2, to simply refer to a statement as being uncertain or approximate, but one can still add the type and level of the uncertainty to it.

Expressing the level of uncertainty was not in the original proposal of S5, we added that using amt:weight, and still it ranked first in query-runtime. Without this extra triple, S5 would only use 2 triples to express uncertainty and might score faster query-runtime. But this solution requires creating new properties for all possible occurring of uncertainty, property explosion.
The idea of S8 was to express multiple occurring of uncertainty in a simple way, example:

- Subject has uncertainty in:
  - Property 1 with value Object 1
  - Property 2 with value Object 2
  - …
  - Property n with value Object n

S3 was the most complex solution to understand and implement, it also ranked last in both query-runtime and number of triples.

7. Summary and Conclusion

Objectives of this thesis were to create modeling use cases of the Nomisma-Ontology, provide solution suggestions for modelling layers of a hoard vessel and comparing and evaluating solutions for modeling uncertainties in the domain of numismatics.

In the theoretical part of this work, we presented the foundations of the topics covered in this thesis. Starting with an introduction to the semantic web and methods and standards used to represent information on the web in way understandable for both humans and machines. Ending with an overview of the importance of representing archaeological artifacts on the web with a focus on ancient coins and hoards.

Working on achieving the first objective, we created use cases for the different classes and properties of the Nomisma-Ontology. The procedure was first writing the use cases in RDF/XML then using an RDF visualization tool (RDF Grapher) to represent each use case as an RDF-graph. We also used an RDF validation and concerting tool (RDF-Converter) to convert the RDF/XML format of the use cases into Turtle. Browsing through the different datasets under Nomisma.org, and the dataset provided by Dr. Tolle, helped in understanding the numismatic concepts and creating the use cases. Finally, created within a wiki-page some sort of a cookbook of these use cases as a guide to using this ontology for representing numismatic objects on the web. We posted each use case individually to facilitate the process of future correction and addition by the experts in the Nomisma.org committee.

For modeling the layers of a hoard vessel, we presented two approaches. One approach handles the layers as stratigraphic units —a Nomisma-class– and uses the Nomisma-properties isAbove, isBelow and isEqual—based on Harris-Matrix– to describe the position of each layer within the vessel. The other approach uses the RDF-Collection, a closed container, within which the entries are organized by the order of inclusion.

The use or storage circumstances of ancient coins can detract their informative features what make their identification a difficult, leaving experts sometimes uncertain of their classification. Towards accomplishing the last objective of this thesis, we presented eight uncertainty modeling solutions, visualized these using an example of two coins, one is certain to be minted in comama, and for the other this is uncertain.

In order to test these eight solutions on large datasets of 100K- to 1M-coins, we created a web application, implemented the eight solutions within it and used it to generate an RDF/XML file from a given Excel/CSV file based on one of the solutions models. We tested the solutions according to query- runtime and memory consumption in retrieving certain and uncertain data.

The result of this test showed that solution 5, that is based on defining new CIDOC CRM .2 properties, was fastest in the total median runtime (the median runtime of five queries summarized). However, we modeled this solution according to the idea proposed by solution 2, which is adding the uncertainty directly to the property path. Solution 2 was easier to work with, it is not complicated, requires little number of triples and there is no need to define extra new properties for uncertain cases like S5.

Finding a uniformed modeling approach for uncertain information is a topic still open for discussion and solution suggestions. We believe that the web applications we created — which is called SAUN, a Serialization Application for Uncertainty in Numismatic —can be expanded by implementing further solution models in it, what would facilitate the comparison and testing of these models.

8. Bibliography

Yu, L. (2011). *A Developer's Guide to the Semantic Web.* Springer.

Moraitoua, E., Christodouloua, Y., & Caridakisa, G. (2021). *Semantic models and services for conservation and restoration of cultural heritage: a comprehensive survey.* Semantic web journal.

Kemmers, F., & Myrberg , N. (2011). Rethinking numismatics: the archaeology of coins. *Cambridge University Press* , 87-108.

Laskey, K. J. (2008, March 31). *Uncertainty Reasoning for the World Wide Web*. Retrieved from W3C: https://www.w3.org/2005/Incubator/urw3/XGR-urw3/

Niccolucci, F. H. (2016). *Expressing reliability with CIDOC CRM.* Springer.

Doerr, M. L. (2020). *Implementing the CIDOC Conceptual Reference Model in RDF, Version 1.1.* CIDOC CRM.

Alexiev, V. (2013, February 12). *Confluence*. Retrieved August 2021, from BMX - BM Extensions to CRM: https://confluence.ontotext.com/display/ResearchSpace/BMX

Alexiev, V., & Mahmud, J. (2012, November 15). *Confluence*. Retrieved August 2021, from BM Association Mapping: https://confluence.ontotext.com/display/ResearchSpace/BM+Association+Mapping

Ackermann, R., Codine, F., Dahmen, K., Duyrat, F., Gruber, E., Heath, S., . . . Wigg-Wolf, D. (n.d.). *Nomisma.org*. Retrieved Mai 2022, from http://nomisma.org/

W3C. (2012). *OWL 2 Web Ontology Language Primer*. Retrieved April 2022, from W3C Primer: https://www.w3.org/TR/owl2-primer/

W3C. (2015, April 10). *Ontology*. Retrieved April 2022, from W3C: https://www.w3.org/standards/semanticweb/ontology

W3C. (2013, March 21). *SPARQL 1.1 Query Language*. Retrieved April 2022, from https://www.w3.org/TR/sparql11-query/

W3C. (2013). *SPARQL 1.1*. Retrieved April 2022, from https://www.w3.org/TR/2013/REC-sparql11-overview-20130321/

W3C, Eric Prud'hommeaux, Steve Harris, Garlik, a part of Experian, Andy Seaborne, The Apache Software Foundation,. (2013, March 21). *SPARQL 1.1 Query Language*. Retrieved April 2022, from https://www.w3.org/TR/sparql11-query/

Walker, A. (2018, March 28). *Hoards and other coin finds*. Retrieved September 2021, from Coins Weekly: https://coinsweekly.com/hoards-and-other-coin-finds-part-i-hoards/

Thiery, F. U. (2018, January 19). *Academic Meta Tool*. Retrieved February 2022, from Academic Meta Tool Vocabulary: http://academic-meta-tool.xyz/vocab/

Thiery, F. N.-W. (2021, September 22). *Zenodo*. Retrieved February 2022, from Nomisma Uncertainty and AMT Vagueness Modelling Appraoches: https://zenodo.org/record/5520978#.YnKayfFByrx

Shaw, R. (2021, April). *Extended Date/Time Format (EDTF) concepts*. Retrieved March 2022, from https://periodo.github.io/edtf-ontology/

Nomisma.org. (2016, 03). Retrieved April 2022, from Llíria Hoard: http://nomisma.org/id/lliria_hoard

Miles, A., & Brickley, D. (2005, November). *W3C*. Retrieved April 2022, from SKOS Core Guide: https://www.w3.org/TR/2005/WD-swbp-skos-core-guide-20051102/

FORTH-ICS. (2014, Novwmber 19). *CIDOC CRM*. Retrieved January 2022, from CRM PC: https://www.cidoc-crm.org/sites/default/files/CRMpc_v1.1_1.rdfs

Doerr, M. e. (2014, October). *CIDOC CRM*. Retrieved February 2022, from Issue 349: Belief Values: https://cidoc-crm.org/Issue/ID-349-belief-values

Doerr, M. E. (2021). *CIDOC CRM*. Retrieved January 2022, from CIDOC CRM: http://www.cidoc-crm.org/versions-of-the-cidoc-crm

Congress, T. L. (2022, February). *The Library of Congress*. Retrieved March 2022, from Extended Date/Time Format (EDTF) Specification: https://www.loc.gov/standards/datetime/

Berners-Lee, T. (2006). *Linked Data*. Retrieved April 2022, from https://www.w3.org/DesignIssues/LinkedData.html

CIDOC-CRM. (2009). *45th joint meeting of the CIDOC CRM SIG and SO/TC46/SC4/WG9; 38th FRBR – CIDOC CRM Harmonization meeting.* Foundation of Research and Technology -Hellas Vassilika Vouton, Heraklion. CIDOC CRM.

Tolle, K., & Wigg-Wolf, D. (2014). *Uncertainty Handling for Ancient Coinage.* Big Data Lab.

Tolle, K., Wigg-Wolf, D., Sabah, R., & Sabah, Z. (2022, April). *Nomisma.org-Cookbook*. Retrieved from Hypothesis: https://nomisma.hypotheses.org/category/nomisma-org-cookbook#

Wigg-Wolf, D., Tolle, K., & Kissinger, T. (2019, March 16). *Nomisma.org:Numismatik und das Semantic Web.* Retrieved 04 2022, from Zenodo: https://zenodo.org/record/4622251#.YiPb9_HMKrw

Vitale, V., Barker, E., & Kahn, R. (n.d.). *Pelagios Network*. Retrieved 05 2022, from Pelagios Network: https://pelagios.org/

Laskey, K. J., Laskey, K. B., Costa, P. C., Kokar, M. M., Martin, T., & Lukasiewicz, T. (2008, March 31). *Uncertainty Reasoning for the World Wide Web*. Retrieved 11 2021, from W3C: https://www.w3.org/2005/Incubator/urw3/XGR-urw3/

Gandon, F., Schreibe, G., & Beckett, D. (2014, February 25). *RDF 1.1 XML Syntax*. Retrieved 03 2022, from W3C: https://www.w3.org/TR/rdf-syntax-grammar/

Earl, G., Sly, T., Chrysanthi, A., Murrieta-Flores, P., Papadopoulos, C., Romanowska, I., & Wheatley, D. (2012, March 26-30). Archaeology in the Digital Era Volume II. In G. Earl, T. Sly, A. Chrysanthi, P. Murrieta-Flores, C. Papadopoulos, I. Romanowska, & D. Wheatley (Eds.), *Computer Applications and Quantitative Methods in Archaeology (CAA)* (U. o. Press, Trans., Vol. II, pp. 487-497). Amsterdam: Amsterdam University Press, University of Chicago Press.

Cripps, Paul J. (2012). Places, People, Events and Stuff; Building Blocks for Archaeological Information Systems. In *Archaeology in the Digital Era Volume II.* Amsterdam University Press, University of Chicago Press.

Wickens, J. M. (n.d.). *The Production of Ancient Coins*. Retrieved 05 2022, from Lawrence University: http://www2.lawrence.edu/dept/art/BUERGER/ESSAYS/PRODUCTION.HTML

Stardog. (n.d.). *Lerning SPARQL*. Retrieved 01 2022, from Stardog Union at DOCS: https://docs.stardog.com/tutorials/learn-sparql

Gruber , E., Heath , S., Meadows , A., Pett , D., Tolle , K., & Wigg-Wolf, D. (2012). Semantic Web Technologies Applied to Numismatic Collections. In *Archaeology in the Digital Era Volume II* (Vol. II).

Stamer, A. (2020). *Cold Start*. Retrieved 05 2022, from Bertiebswirtschaft Lernen: https://www.betriebswirtschaft-lernen.net/en/explanation/cold-start-computer/

Oras, E. (2013). Importance of terms: What is a wealth deposit? *Papers from the Institute of Archaeology (PIA) - Student Jurnal, 22*(Volume 22), 61-82.

Structured vs. Unstructured Data: A Complete Guide. (n.d.). Retrieved 05 2022, from Talend: https://www.talend.com/resources/structured-vs-unstructured-data/#:~:text=Structured%20data%20is%20highly%20specific,employs%20schema%2Don%2Dread.

RDF Grapher. (n.d.). Retrieved 05 2022, from Living Laboratory of the Linked Data Finland: https://www.ldf.fi/service/rdf-grapher

RDF-Converter. (n.d.). Retrieved 05 2022, from isSemantic: https://issemantic.net/rdf-converter

Arndt, D., Broekstra, J., DuCharme, B., Lassila, O., Patel-Schneider, P. F., Prud'hommeaux, E., . . . Thompson, B. (2022, May 20). *RDF-Star and SPARQL-Star*. Retrieved 06 2022, from W3C: https://w3c.github.io/rdf-star/cg-spec/editors_draft.html#turtle-star

Brickley, D., Guha, R., & McBride, B. (2014, February 25). *RDF Schema 1.1*. Retrieved 02 2022, from W3C: https://www.w3.org/TR/rdf-schema/

Sharless, A., & al., e. (2021, July 15). *Nomisma Ontology*. Retrieved 06 2022, from Nomisma.org: http://nomisma.org/ontology

Rörtgen, S. (2020, April 15). *RDFlib*. Retrieved 06 2022, from GitHub: https://github.com/RDFLib/rdflib/issues/1003

Python Package Index. (n.d.). Retrieved 11 2021, from Python Package Index: https://pypi.org/

Python Tutorials. (n.d.). Retrieved 11 2021, from Python Tutorials: https://pythonspot.com/

SPARQL Endpoint interface to Python. (n.d.). Retrieved 12 2021, from SPARQL Wrapper: https://sparqlwrapper.readthedocs.io/en/latest/index.html#

List of Figures

List of Tables

Used Tools:

To create the RDF-graphs we used the web service RDF- Grapher by Linked Data Finland:
RDF Grapher: https://www.ldf.fi/service/rdf-grapher

To convert the RDF/XML format into the Turtle format we used the online tool isSemantic.net/ RDF CONVERTER.
isSemantic.net: https://issemantic.net/

To create **Figure 4.** ***19*** and **Figure 4.** ***22*** we used isSemantic.net/ RDF VISUALIZER, to visualize the Turtle-Star lines as graphs, then we redraw those to keep a consistent look of all RDF-graphs.

Work Distribution:

Chapters 2, 4 & 6: Zeena & Ram
Chapters 1, 3: Zeena sabah

- Creating the Nomisma-Cookbook: Zeena & Ram

Chapters 5, 7: Ram Sabah

- Implementing the Uncertainty Solutions in SAUN: Zeena & Ram

Appendices

Appendix 1: Nomisma.org-CookBook

Modeling the Nomisma-Ontology

This appendix includes all modeling use cases of the Nomisma.org-Cookbook.

Appendix 2: RDF/XML Format of the Uncertainty-Solutions

This appendix shows the RDF/XML format of the uncertainty modeling solutions presented in chapter 4.

Namespaces for all solutions:

```
nmo="http://nomisma.org/ontology#"
nm="http://nomisma.org/id/"
bmo="http://collection.britishmuseum.org/id/ontology/"
xsd="http://www.w3.org/2001/XMLSchema#"
rdfs="http://www.w3.org/2000/01/rdf-schema#"
crm="http://www.cidoc-crm.org/cidoc-crm/"
un="http://www.owl-ontologies.com/Ontology1181490123.owl#"
rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
crminf="http://www.ics.forth.gr/isl/CRMinf/"
amt="http://academic-meta-tool.xyz/vocab#"
dcterms="http://purl.org/dc/terms/"
dcmitype="http://purl.org/dc/dcmitype/"
position="https://sws.geonames.org"
skos="http://www.w3.org/2004/02/skos/core#"
nmo="http://nomisma.org/ontology#"
meta="http://www4.wiwiss.fu-berlin.de/bizer/d2r-server/metadata#"
foaf="http://xmlns.com/foaf/0.1/"
owl="http://www.w3.org/2002/07/owl#"
db="https://data.corpus-nummorum.eu/"
```

Solution 1:

```
<rdf:Description rdf:about="Coin_1">
    <nmo:hasMint rdf:resource="nm:comama"/><!--certain-->
</rdf:Description>
<rdf:Description rdf:about="Coin_2">
    <nmo:hasMint rdf:resource="nm:comama"/><!--uncertain-->
</rdf:Description>
<rdf:Description>
    <crm:P140_assigned_attribute_to rdf:resource="Coin_2"/>
    <crm:P141_assigned rdf:resource="nm:comama"/>
    <bmo:PX_Property rdf:resource="nmo:hasMint"/>
    <rdf:type rdf:resource="crm:E13"/>
    <bmo:PX_likelihood rdf:resource="nm:uncertain_value"/>
</rdf:Description>
```

Solution 2:

```
<rdf:Description rdf:about="Coin_1">
    <nmo:hasMint rdf:resource="nm:comama"/><!--certain-->
```

```
</rdf:Description>
<rdf:Description rdf:about="Coin_2">
    <nmo:hasMint>
        <rdf:Description><!--adding uncetainty directly to the property path-->
            <rdf:value rdf:resource="nm:comama"/>
            <un:hasUncertainty rdf:resource="nm:uncertain_value"/>
        </rdf:Description>
    </nmo:hasMint>
</rdf:Description>
```

Solution 3:

```
<rdf:Description rdf:about="Coin_1">
    <nmo:hasMint rdf:resource="nm:comama"/> <!--certain-->
</rdf:Description>
<rdf:Description rdf:about="Coin_2">
    <nmo:hasMint rdf:resource="nm:comama"/> <!--uncertain-->
</rdf:Description>

<rdf:Description rdf:nodeID="b0">
    <crm:P140_assigned_attribute_to rdf:resource="Coin_2"/>
    <crm:P141_assigned rdf:resource="nm:comama"/>
    <rdf:Property rdf:resource="nmo:hasMint"/>
    <rdf:type rdf:resource="crm:E13"/>
    <crm:T2_assessd_as_reliability rdf:nodeID="c0"/>
  </rdf:Description>
  <!--The statement saing "Coin_2 minted in comama" is 92% reliable-->
  <rdf:Description rdf:nodeID="c0">
    <rdf:type rdf:resource="crm:R2_Reliability"/>
    <crm:P90_has_value rdf:datatype="xsd:double">0.92</crm:P90_has_value>
</rdf:Description>

<rdf:Description rdf:nodeID="A1">
    <crm:T1_assessd_the_reliability_of rdf:nodeID="b0"/>
    <rdf:type rdf:resource="crm:R1_Reliability_Assessment"/>
</rdf:Description>
```

Solution 4:

```
<rdf:Description rdf:about="Coin_1">
    <nmo:hasMint rdf:resource="nm:comama"/><!--certain-->
</rdf:Description>
<rdf:Description rdf:about="Coin_2"><!-believed to be uncertain-->
    <nmo:hasMint>
        <crminf:I2_Belief>
           <crminf:J4_that rdf:resource="nm:comama"/>
           <crminf:J5_holds_to_be>uncertain</crminf:J5_holds_to_be>
        </crminf:I2_Belief>
```

```
    </nmo:hasMint>
</rdf:Description>
```

Solution 5:

```
<rdf:Property rdf:about="P189.2_uncertain_place">
    <rdfs:domain rdf:resource="crm:E13"/>
    <rdfs:subPropertyOf rdf:resource="crm:P189"/>
</rdf:Property>
<rdf:Description rdf:about="Coin_1">
    <nmo:hasMint rdf:resource="nm:comama"/>
</rdf:Description>

<rdf:Description rdf:about="Coin_2">
    <nmo:hasMint>
        <rdf:Description>
            <crm:P189.2_uncertain_place rdf:resource="nm:comama"/>
            <amt:weight>0.56</amt:weight>
        </rdf:Description>
    </nmo:hasMint>
</rdf:Description>
```

Solution 6:

```
<rdf:Description rdf:about="Coin_1">
    <nmo:hasMint rdf:resource="nm:comama"/>
</rdf:Description>

<rdf:Description rdf:about="Coin_2">
    <nmo:hasMint rdf:resource="nm:comama"/>
</rdf:Description>
<edtfo:UncertainStatement rdf:about="statement_c4">
    <rdf:subject rdf:resource="Coin_2"/>
    <rdf:predicate rdf:resource="nmo:hasMint"/>
    <rdf:object rdf:resource="nm:comama"/>
</edtfo:UncertainStatement>
```

Solution 7:

```
<rdf:Description rdf:about="Coin_1">
    <nmo:hasMint rdf:resource="nm:comama"/>
</rdf:Description>
<rdf:Description rdf:about="Coin_2">
    <nmo:hasMint>
        <edtfo:ApproximateStatement>
            <rdf:value rdf:resource="nm:comama"/>
        </edtfo:ApproximateStatement>
    </nmo:hasMint>
</rdf:Description>
```

Solution 8:

```
<!--Simple-->
<rdf:Description rdf:about="Coin_1">
    <nmo:hasMint rdf:resource="nm:comama"/>
</rdf:Description>
<rdf:Description rdf:about="Coin_2">
    <un:hasUncertainty>
        <rdf:Description>
            <nmo:hasMint rdf:resource="nm:comama"/>
        </rdf:Description>
    </un:hasUncertainty>
</rdf:Description>
```

```
<!--Complex-->
<rdf:Description rdf:about="Coin_2">
    <nmo:hasWeight rdf:datatype="xsd:decimal">17.2</nmo:hasWeight>
    <nmo:hasAxis rdf:datatype="xsd:decimal">2.0</nmo:hasAxis>
    <un:hasUncertainty>
        <rdf:Description>
            <nmo:hasMint rdf:resource="nm:comama"/>
            <nmo:hasDenomination rdf:resource="nm:large_ae2"/>
            <nmo:hasDie rdf:resource="http://.../Die_4"/>
        </rdf:Description>
    </un:hasUncertainty>
</rdf:Description>
```

```
<!—More Complicated-->
<rdf:Description rdf:about="Coin_4">
    <nmo:hasWeight rdf:datatype="xsd:decimal">17.2</nmo:hasWeight>
    <nmo:hasCorrosion rdf:resource="nm:heavily_corroded"/>
    <un:hasUncertainty>
        <rdf:Description>
            <nmo:hasMint>
                <rdf:Description>
                    <rdf:value rdf:resource="nm:comama"/>
                    <crm:P89_falls_within rdf:resource="nm:asia_minor"/>
                </rdf:Description>
            </nmo:hasMint>
            <nmo:hasDenomination rdf:resource="nm:large_ae2"/>
            <nmo:hasDie>
                <rdf:Description>
                    <rdf:value rdf:resource="http://.../Die_4"/>
                    <nmo:hasCorrosion rdf:resource="nm:extremely_corroded"/>
                    <un:hasUncertainty>
                        <rdf:Description>
```

```
                        <nmo:hasDate>170-162 B.C.</nmo:hasDate>
                    </rdf:Description>
                </un:hasUncertainty>
            </rdf:Description>
        </nmo:hasDie>
        <nmo:hasIssuer>
            <rdf:Description>
                <rdf:value rdf:resource="http://.../issuer_A"/>
                <rdf:value rdf:resource="http://.../issuer_B"/>
                <rdf:value rdf:resource="http://.../issuer_C"/>
            </rdf:Description>
        </nmo:hasIssuer>
    </rdf:Description>
</un:hasUncertainty>
</rdf:Description>
```

Layers of a Hoard Vessel:

```
<nmo:Hoard rdf:about="hoard_1">
    <crm:P167i_includes>
        <nmo:StratigraphicUnit rdf:about="layer_1">
            <dcterms:tableOfContents rdf:resource="http://../content_l1"/>
            <nmo:isBelow rdf:resource="layer_2"/>
        </nmo:StratigraphicUnit>
    </crm:P167i_includes>
    <crm:P167i_includes>
        <nmo:StratigraphicUnit rdf:about="layer_2">
            <dcterms:tableOfContents rdf:resource="http://../content_l2"/>
            <nmo:isAbove rdf:resource="layer_1"/>
            <nmo:isEqual rdf:resource="layer_3"/>
        </nmo:StratigraphicUnit>
    </crm:P167i_includes>
</nmo:Hoard>
<nmo:StratigraphicUnit rdf:about="layer_1">
    <dcterms:tableOfContents>
        <dcmitype:Collection>
            <rdfs:label>Content of layer 1 of Hoard_X</rdfs:label>
            <crm:P167i_includes rdf:resource="item_1"/>
        </dcmitype:Collection>
    </dcterms:tableOfContents>
    <nmo:isBelow rdf:resource="http://../layer_2"/>
</nmo:StratigraphicUnit>
<nmo:StratigraphicUnit rdf:about="layer_2">
    <dcterms:tableOfContents>
        <dcmitype:Collection>
            <rdfs:label>Content of layer 2 of Hoard_X</rdfs:label>
```

```
            <crm:P167i_includes rdf:resource="item_2"/>
            <crm:P167i_includes rdf:resource="item_3"/>
        </dcmitype:Collection>
    </dcterms:tableOfContents>
    <nmo:isAbove rdf:resource="http://../layer_1"/>
</nmo:StratigraphicUnit>
```

Appendix 3: Queries

This appendix includes the five queries used in chapter 6 to evaluate the solutions, in addition to two counter queries.

Appendix 4: Tables

This appendix includes two tables, ***Table A4. 1*** lists the CIDOC-CRM properties that have .1 properties, and ***Table A4. 2*** lists the Nomisma.org properties and the CIDOC-CRM properties that we created to use in solution 5.

Table A4. 1: CRM .1 Properties

CRM Property	.1 Property
P3_has_note	P3_has_type
P14_carried_out_by (performed)	p14_in_the_role_of
P16_ used_specific_object (was used for)	P16_mode_of_use
P19_was_intended_use_of (was made for)	P19_mode_of_use
P62_ depicts (is depicted by)	P62_mode_of_depiction
P69_ has_association_with (is associated with)	P69_has_type
P102_has_title (is title of)	P102_has_type
P107_ has_current_or_former_member (is current or former member of)	P107_kind_of_member
P130_ shows_features_of (features are also found on)	P130_kind_of_similarty
P136_ was_based_on (supported type creation)	P136_in_the_taxonomic_role
P137_ exemplifies (is exemplified by)	P137_in_the_taxonomic_role
P138_ represents (has representation)	P138_mode_of_representation
P139_has_alternative_form	P139_has_type
P144_ joined_with (gained member by)	P144_kind_of_member
P189_ approximates (is approximated by)	P189_has_type

Table A4. 2: CRM .2 Properties that we defined and used for solution 5

Nomisma Properties	CIDOC-CRM .2 Properties
hasAxis	P3_uncertain_value
hasDepth	
hasMaxDepth	
hasMinDepth	
hasDiameter	
hasMaxDiameter	
hasMinDiameter	
hasHeight	
hasMaxHeight	

hasMinHeight	
hasWidth	
hasMaxWidth	
hasMinWidth	
hasWeight	
hasWeightStandard	
hasBearsDate	
hasEndDate	
hasStartDate	
hasDenomination	
hasCollection	P107_uncertain_member
hasTypeSeriesItem	
hasContemporaryName	P102_uncertain_name_or_ethnic
hasScholarlyName	
hasDie	P16_uncertain_technique_or_object_used_for_creation
hasProductionObject	
hasManufacture	
hasCountermark	P103_uncertain_symbole_or_features
hasMintmark	
hasSecondaryTreatment	
hasPeculiarity	
hasPeculiarityOfProduction	
hasCorrosion	
hasWear	
hasObjectType	P67_uncertain_type
representsObjectType	
hasAuthenticity	P138_uncertain_authenticity
hasAuthority	P14_uncertain_authority_or_issuer

hasIssuer	
hasMint	P189_uncertain_place
hasFindspot	
hasMaterial	P137_uncertain_material
hasContext	P136_uncertain_context_or_taxonomy
hasAppearance	P139_uncertain_form
hasShape	
hasEdge	
hasFace	P19_uncertain_mode
hasObverse	
hasReverse	
hasPortrait	P62_uncertain_depiction
hasIconography	
hasLegend	

Appendix 5: Main Classes and Functions of SAUN

In this appendix we explain in detail the main classes and function of the web application SAUN. In creating SAUN we used the following resources: (Python Tutorials), (SPARQL Endpoint interface to Python), (Python Package Index).

Link to SAUN on GitHub: https://github.com/ramsabah/saun

Class readAndSeparate

This class is responsible for data preparation. First is to read the data file, which is either an Excel- or a CSV- file. Any other file type will lead to an error, and a reload of the HTML-index page. Reading the data file can be done in many ways, for an easier approach, we use `csv`-library for reading the csv-files and `pandas` for reading Excel-files and writing both CSV- and Excel-files.

After selecting the data file and the modeling solution to be generated, the function `runDataPreparation()` will be called. This function checks if the file has the *"xlsx"* extension, which is for Excel, in the file name. If the *"xlsx"* extension exists in the file name, we call the function `convert()`, which use the `pandas` library to read the Excel file and rewrite it with the same name in the *csv* directory. In case the uploaded file is a *csv*, this step will be skipped. Very important is to set the index to `None` and headers to `True` to make sure we have a correct writing of the file, otherwise, the structure of the file will be invalid.

Function `readAndSeparate()`. This function is very important for the solution generation. In this function at first the *csv* file will be read, then we look if there are headers defined as uncertain. If the header did not contain the keyword '`http`' then it is uncertain header otherwise it is an ontology term, more specifically, a property URI.

```
            if 'http' in headers[iteration for each header]:
```

Now we need to separate the certain headers from the uncertain headers and store each group with their data in a separate file. For this operation we defined four new arrays: Certain headers, uncertain headers, certain data, and uncertain data. Then we use pandas Data frame to generate tow Excel files. One file containing the certain data and headers, and the other

file containing the uncertain data and headers. The Headers for each array should be the input in the header= value of the panda function (to Excel). We also set the index to *None* to prevent *pandas* from generating index in the Excel file.

```
return csv.DictReader(„file as csv or excel with utf-8
                      encoding“).fieldnames
```

When this function is finished, we call the function *convert()* again to convert the file to *csv* and overwrite the original csv file.

Functions *getHeader()* and *getContent()* for sending the data to the class *Dynamic*. In the function *getHeader()* uses the function *getFilename()* to read the current converted file. Since we are only interested to sending headers to the class *Dynamic*, we only return the *column-header-names*.

```
return csv.DictReader(„file as csv or excel with utf-8
                      encoding“).fieldnames
```

The function *getContent()* have the same build structure as *getHeader()*. But, instead of returning the fieldnames we return the content. This is done by using the function *Next()*, that skips the headers.

```
Next('csv or excel file as 'file''), return(file)
```

Function *seperateUncertainTable()*, this function is invoked by the function *runDataPreparation()* and has the target of separating the columns with the header of uncertainty from the columns that contains a header of a property URI. The function *seperateUncertainTable()* will first check whether the input file is an excel- or csv- file, and store the data of the input file in a variable named *'file'*. To separate the uncertain columns, we must also separate their entire data. Therefore, we define four arrays: tow arrays carry the certain headers and the certain data, and tow arrays for carrying the uncertain headers and the uncertain data. Furthermore, we need to save some data temporarily before it is written in the four arrays, therefore we define another tow arrays as data holders (helper arrays). Now we read the headers and the data of the input file and save each in a separate list using the following code lines:

```
headers = csv.DictReader(file).fieldnames
data = csv.reader(file)
```

Starting the separation by looping the length of the headers and checking each element if the keyword *'http'* is included in their string. If True, the header will be appended in the array of certain headers, and we write all the data of the index of the certain headers in the temporarily data holder array (help array). Then we append the content of the help array to the certain data array.

If the keyword *'http'* is not included in the header, the latter will be appended to the uncertain headers array, and we loop their data with the same index of the header and write it to the temporarily data holder array. Then we append the content of the help array to the uncertain data array and set the help array to empty.

Now the data is separated in both arrays. Next, we convert both arrays to *pandas.DataFrame* to use the method *excel_writer* as follow:

```
certain.to_excel(excel_writer="readyToRun/"+str(getFilename()).replace('c
sv','xlsx'), header=certain_Headers_, index=None, encoding='utf-8')
```

we run the function of the panda's library *to_exce(a, b, c, d)* on the array with the certain data with saving path in *a*, the certain headers array in *b*, setting the index to *Nun* in *c*, with the encoding *utf-8* in *d*. The same is applicable to the uncertain data and headers arrays.

By testing the performance of the program, we recognized, that the *pandas.DataFrame* and the function *to_excel* take a lot of time to finish. This can be avoided by returning the data as array to the class Dynamic instead of using *pandas.DataFrame*. However, the user must then upload the file each time they want to run a solution on the same data file. With our implementation, using *pandas.DataFrame* as descried above, the user needs to upload the file only one time to create all solutions.

The improvement we have made here is making the program recognize that the user is using the same file to generate other solutions. The class *readAndSeparate* is skipped entirely and the *Dynamic* class is run to generate the selected solution.

Class Dynamic

This class is responsible for building and generating the RDF-file based on the in the UI selected solution for modeling uncertainty. For the implementation we used the following

libraries: *JSON*, *random*, *pandas*, *rdflib(Graph*, *RDF*, *RDFS*, *Namespace*, *DCTERMS*, *URIRef*, *Literal*, *BNode)*, *re* *and* *NumPy*. Also, we need to import the class *readAndSeperate.py* to be able to access the method for receiving the headers, content, and uncertain array.

Namespaces(), this function is responsible for reading and generating a list of property-names, **Figure A5. *1*** shows the processing steps of this function. These property-names are written within the property URI at the end after a hashtag '#' or in some cases after a backslash '/'. For reading and saving the property-names we iterate the property URI in reversed way, and we use both symbols (# and /) to detect when the object property ends. We save the property-name with the rest of the URI in an array and use the rest of the latter to get the ontology of it using the function *getOntology()* with the rest of the URI as input. The function *Namespaces()* will return an array containing arrays of three elements that holds the property-names, namespace-prefix of the corresponding ontology, and the rest of the property-URI. For example, the property 'hasMint' from the Nomisma-Ontology:

```
['http://nomisma.org/ontology#', 'nmo', hasMint]
```

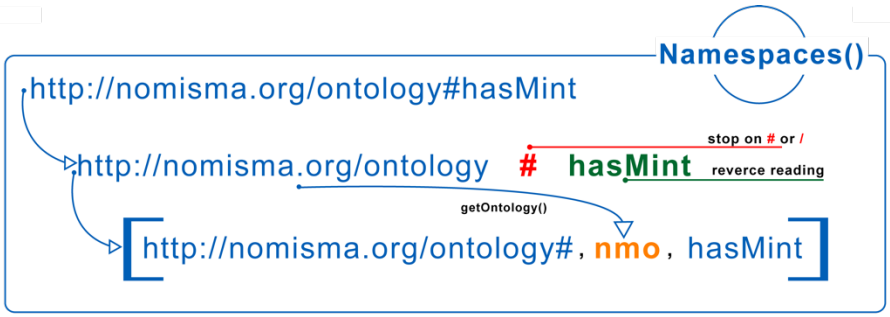


Figure A5. 1: Processing steps of the function Namespaces().

To generate the RDF-file based on the selected solution, we must generate an *rdflib.Graph*. Bute before we start building the graph, we must prepare the data to be passed in RDF format. In following we describe the functions we build to generate the necessary data for building the graph.

getOntology(), this function takes only one parameter in form of a URI, as the example below, identifies the hash singe *'#'* and the name of the ontology 'e.g., nomisma', and

returns the property (e.g., `hasMint` comes after the #) together with the namespace prefix for the ontology `'nmo'`.

```
http://nomisma.org/ontology#hasMint
```

The URI starts with the keyword for Hyper Text Transfer Protocol (Secure) 'http' or ('https'). With this keyword we know that it is a URI of a domain specific ontology. On the right side of the keyword 'http' is a colon ':' followed by the double slash '//', and the ontology name 'nomisma.org', as in the example above. For nomisma.org we have two namespace prefixes: nmo for `nomisma.org/ontology`, and nm for nomisma.org/id. In other words, we must exactly assign the prefix identification by typing the content of the link and assign them to a specific prefix as shown in the following example:

```
(["nomisma","ontology"],'nmo')
```

This may not be an efficient way to identify the prefixes since each namespace prefix must be specified manually, but we could not find a better approach to solve this dynamically.

To check if all keywords (e.g., `"nomisma","ontology"`) exists in the URI, we need to define a validator in the function *`getOntology()`* and set it to *`False`*. Then we loop the list of ontologies that we previously defined (outer loop) and loop each element at the 0 index of each list (inner loop). If the inner loop ends without breaking means all the keywords are included in the link, and the second element of the list which is the prefix (`nmo` in this example) is returned. If a keyword is not found in the link, then the inner loop will break and the outer loop will jump to the next element of the list of ontologies.

Note that, for this method to work error-free, the keywords for identifying the prefixes must be unique for each namespace name and in the correct order. In some cases, the namespace string may include keywords shared by other namespace strings (URI to ontology). For example, to differentiate between the two namespace strings below, we can choose the keywords as following: [rdfs, void] for the first namespace string, and [rdfs, sioc] for the second.

[rdfs, void] to identify `'http://rdfs.org/ns/void#'`, and

[rdfs, sioc] to identify `'http://rdfs.org/sioc/services#'`

valueLinkSeparator(), with this function we separate the property-value from the data type. For instance, when the property-value has a defined data type, e.g., integer, decimal, double, string, etc., then it is entered in a csv/excel table as follows:

```
1^^http://www.w3.org/2001/XMLSchema#integer
```

The *valueLinkSeperator()* function receives the namespace URI and detects whether it begins with the keywords http or https. When one of these keywords is included, we add each character after it in a substring until a space is reached, the end of the namespace URI. Anything that is not part of the namespace URI should be the data value (e.g., numbers or dates). We strip the rest of the namespace URI of all other characters and put it in an array with the data value, as follows:

```
[5.34, 'http://www.w3.org/2001/XMLSchema#decimal']
```

If the data value were a string, and the word *string* is included in the namespace URI, then the function *valueLinkSeparator()* will run the following code:
return *input.split('^^')*.
This will split the input by the '^^' and return a list with two elements, the string, and the namespace URI.

```
['example', 'http://www.w3.org/2001/XMLSchema#string']
```

And this is easily accessible with *index 0* for the data value and *index 1* for the namespace URI.
The function *uncertainArray_()* is responsible for creating, completing and correcting the uncertain array. First, we check if the uncertain array, generated in the *readAndSeperat* class is empty. In this case we generate a 2D array that matches the length of the headers and the depth of the data. We fill the array with zeros 0 to indicate that all data is certain. This step reduces using a conditional statement in the function *G()*, to check for uncertainty by each solution.

In case the uncertain array is not empty, then we read it using the *pandas.DataFrame*. We also read the headers of the data and the data itself after the separation. Then, for each data header we check if there exists an uncertain header in the uncertain array. If there is a match, then we append the uncertain header to a help array. Then

we loop the content of the header of the uncertain array and add them to the help array. If any field in the uncertain array contains a *nan*, it means the value here is certain. At the end, the data array and the uncertain array must have the same dimensions.

To identify an uncertain header, we used the (*) character, e.g., *hasMint*hasUncertainty*. However, this can be replaced with any character or string of characters and modify the code accordingly. It is also possible to allow a dynamic prefix by creating an input method in the UI that accepts a character or string as an identifier for uncertain headers.

`G()` This function is for generating the RDF/XML file of the uploaded data file (Excel or CSV). The number of the selected solution is stored in a variable called *selectedG*. Based on the selected solution we define the required namespaces. Some Namespaces are already embedded in the RDF library, like rdf, rdfs, and dcterms, which can be directly imported for use. Any other namespace (like: `nmo, nm, crm, edtfo, amt`, etc.,) must be added manually, otherwise it may lead to an invalid W3C standard document.

First we retrieve the result of the function `Namespaces()`. This function will return all the namespaces used in the headers of the input data. Then we define a variable for the new namespace, `NMO` in this example for the Nomisma.org-Ontology:

```
NMO = Namespace('http://nomisma.org/ontology#')
```

Note that, the namespaces exist in the headers of the data file will be inserted in the RDF file automatically. With this code line above we define a namespace we need for building the uncertainty solution. Therefore, for optimization purposes, we only define the namespaces within the code section of the solution where they are used. For example, we defined the CRMinf namespace in the code section of Solution 4. This is done using the `Namespace()` of `rdflib`. After defining the namespaces, we append them to the RDF-Graph as follows:

First, we define a variable G and initialize it as new Graph: G = Graph(). Then we append the namespaces to the Graph using the function bind:

```
G.bind('nmo', 'http://nomisma.org/ontology#')
```

To avoid Duplicates in the namespaces we define a help array called `n_names(namespaces Names)` and with a for-loop we check every iteration if the

namespace is included in `n_names`. If not, we add the namespace to `n_names`. Finally, we append the `n_names` to the Graph G and store the uncertain array from the function `uncertainityArray()` in a separate variable. Now we are ready to build the solution, however, for this part we will focus on building solution two and only explain the main differences to building the other solutions.

After retrieving the selected solution, we start looping the data retrieved from the class *`readAndSeperat`*. And for iterating the uncertain array, we define tow variables *`X_Achse`* and *`Y_Achse`*. First, we loop every subject *i* of *n*, which always has index 0, and every subject has a row where its property values are stored, and we need to iterate each of these values in a row. So, we defined a second for-loop and set its range to the length of the columns *m*.

Subjects	hasProperty_1	hasProperty_2	hasProperty_3	…	hasProperty_m
subject_1	property_value	property_value	property_value	.	property_value
subject _2	.	.	.	.	.
	.	.	.	.	.
subject _n	property_value	property_value	property_value	.	property_value

For each inner iteration:

1. Check if a property value in the subject(row) is non or equal empty string. If so, means that the subject in this column do not have any value. We continue the loop without adding anything to the graph.

2. The Program will check the value of the respective cell in the uncertain array, if it is set to 0, means that the input value of the data is certain. Otherwise, it is uncertain.

In either way, whether certain or uncertain, three cases must be considered in order to attach the entry (subject, predicate, object) to the graph in the correct way:

Case 1: If the cell includes the substring http or https and not XMLSchema then the entry must be added as a URI-Reference to the graph.

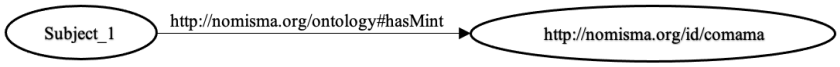
```
G.add((URIRef(con[0]), eval(n_node[inncon]), URIRef(con[inncon])))
```

Where G is the Graph, and the function add is to append the entry (subject, predicate, object) to the graph. With the function URIRef we add the subject `con[0]` and the property value

`con[inncon]` as URI references, for example (*subject_1*, *http://nomisma.org/id/comama*). The property value that contains an *XMLSchema* will be skipped here, because it represents a datatype and not part of the actual property value.

Subjects	http://nomisma.org/ontology#hasMint	http://nomisma.org/ontology#hasWidth
subject_1	http://nomisma.org/id/comama	12.3^^ http://www.w3.org/2001/XMLSchema#decimal

The property URI is added using the function eval with n_node[inncon], where inncon is a variable for iterating each cell of the array.



Case 2: if the cell includes the substring XMLSchema means it includes a property value with its datatype and it should be included as a literal not as URIRef.

```
G.add((URIRef(con[0]), eval(n_node[inncon]),
Literal(valueLinkSeperator(con[inncon])[0],
datatype=valueLinkSeperator(con[inncon[1])))
```

Similar to Case 1, the property is added using the function `eval()`, the subject and the property value are added using the function URIRef(). However, the difference here is in case the property value was a literal (string, numeric or date), with datatype (e.g., `2.3^^http://www.w3.org/2000/01/rdf-schema#decimal`). In this case, we wrap the property-value with the function `valueLinkSeperator()`, that returns a list with first element "property value" and second element "datatype" as follows:

```
[2.3, 'http://www.w3.org/2000/01/rdf-schema#decimal']
```

Case 3: If the cell does not contain a URI or a specific data type (plain text, number, date with characters), it is inserted as a literal in the object entry of the add() function.

```
G.add((URIRef(con[0]), eval(n_node[inncon]), Literal(con[inncon])))
```

The above three cases were to build the graph with the certain entries (where the position in the uncertainty table contains '*nan*'). To add the uncertain entries (those with position in the uncertainty table is not empty), we still need these three cases but with some modifications, which is, acording to solution 2, using blank-node (*BNode*) directly in the property path. This means, that the blank-node will be in the object position (subject-property-`BNode`) in line 1,

then in the subject position in lines 2 & 3 where it has the property 'hasUncertainty' with property value *'uncertain_value'* in line 2. And the property *'value'* with property-value retrieved from the data array.

Case 1_Uncertain Entries:

```
1. G.add((URIRef(con[0]), eval(n_node[inncon]), BNode('b' +
   str(nodeCounter))))
2. G.add((BNode('b' + str(nodeCounter)), UN['hasUncertainty'],
   NM['uncertain_value']))
3. G.add((BNode('b' + str(nodeCounter)), RDF['value'],
   URIRef(con[inncon])))
4. nodeCounter += 1
```

To make each blank-node unique, we define a counter and initialize it with 0 (nodeCounter in line 4). Then we append the counter value as string to the blank-node's name 'b'. This returns us 'b0' for the first uncertain triple, 'b1' for the second and so on untel the last uncertain triple is added.

Cases 2 and 3: Aplaying same proceger as in *Case 1_Uncertain Entries* (lines 1, 2 & 4). In Case 2 line 3, we must replace the *URIRef()* in the object position with *Literal()* and use the result of the function *valueLinkSeperator()*. And in Case 3 line 3, we just add the value of the cell as it is.

Case 2_ Uncertain Entries:

```
3. G.add((BNode('b' + str(nodeCounter)), RDF['value'],
       Literal(valueLinkSeperator(con[inncon])[0],
          datatype=valueLinkSeperator(con[inncon])[1])))
```

Case 3_ Uncertain Entries:

```
3. G.add((BNode('b' + str(nodeCounter)), RDF['value'],
   Literal(con[inncon])))
```

After each inner itteration 'inncon' we increase the *X_Achse* by 1. And after each outer itteration *'con'* (the itteration that loops the subjects), we increase the *Y_Achse* by 1, and reset the *X_Achse* to 0.

Finally, we define a new directory name '*rdf*', define a variable *'G_writer'* and assign it to the function *open()*. We use the created directory to save the solution model under name *'modelGraphDynamic_G2'* and use the method write *'w'* with the encoding to *utf-8*. In case the encoding is missing, the Graph will not be generated. Before running the function write, the graph must be serialized to the *'xml'* format (for a different serialization like turtle, consider changing/or removing the encoding accordingly).

```
G_writer = open('rdf/modelGraphDynamic_G2.rdf', 'w', encoding='utf-8')
G_writer.write(G.serialize(format='xml'))
```

Depending on the size of the input data, after running the write function the graph will take seconds to several minutes to finish generating.

Class app

This class is responsible for the communication between the View and the Models of the application. We created a user Interface based on a Web-application. The webpages are written in HTML and for the communication with the models we use Flask.

To install flask on python run the command pip install Flask. This command will install the flask library, which we use to import the following classes:

- From flask we import Flask, render_template, request, and send_file. We also use json library to write data in json format and os library for running system services/operations.
- Since class 'app' is responsible for the user interface and the communication with other classes, we need to import the class readAndSeperate and Dynamic from the project to access their functions.

Next, we must create two directories: the first directory must have the name templates and the second directory must have the name static. In the template directory we store all HTML files, any other HTML-file outside the template directory will not be accessible. The static directory will have images, CSS, and the JavaScript files. If any of these files is placed outside the static directory it will not be accessible from flask nor displayable on the HTML-pages.

Next, we describe the functions and commands of the class App to make it easy to understand:

Running Index: First, we must define the flask app, this is done by writing the following code:

```
app = Flask(__name__)
```

The instance app is defined as flask app. To run the Flask server, we need to run the following command:

```
if __name__ == '__main__':
    app.run()
```

With these few code lines, we start the Flask server, however, the server will return an error of 'Page Not Found'. This because we did not define the index of the web application. To avoid this, we add the following code:

```
@app.route('/')
def index():  # put application's code here
    return render_template('index.html')
```

App Routing means mapping the URL (e.g., the localhost where the server is running) to a specific function that will handle the logic for that URL. In the above example the server will run the function *`index()`* after server start, because the *`app.route`* have the input ('/'). The function *`index()`* will render the index.html temple and the page will show on the browser. To be able to control the application we have created an index page, in **Figure A5. *2***, with the following control buttons: Solutions, Queries, Graph, Fuseki Control, Ontology Setup and Gallery. In the following part we will focus on the solutions page only and describe the view and models functionality.

Solutions (Index.html): On running the app the index page will appear on the browser. In this page we have 5 menu buttons and a file selection button below. The application is design to accept Excel and CSV files only. The input file must have the header structure as previously mentioned under the function *`Namespaces()`*. The data file can be selected with 'Select Data'. When the data file is selected, the RDF file can be generated by selecting one of the generate solution buttons (Generate Solution 1, …, Generate Solution 8). The program will automatically render the file and create the RDF file version of the selected input file. The data will be saved automatically in a directory (name 'rdf').

In some cases, the selected input file is too large and need time to be generated. Therefore, we created a processing icon, that keep spinning while the application operating. The index page has, in addition to the solutions, three more columns:

- **Visualize**, that displays, when hovering, the basic graph representation of each solution as a thumbnail,
- **Description**, with few lines describing each solution, and
- **Triples**, that gives the number of triples of each solution.

Note, the basic graph and the number of triples is of the case with two coins; Coin_1 is certainly minted in Comama, and for Coin_2 this is uncertain.

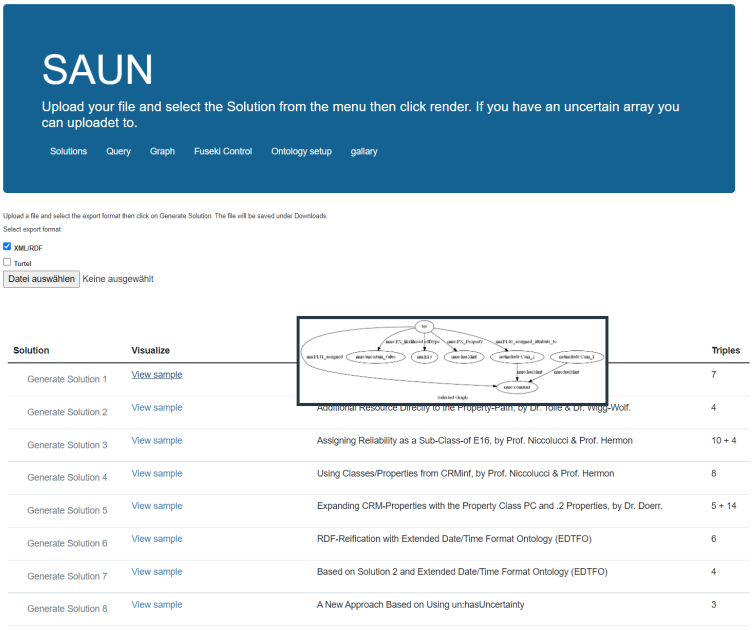


Figure A5. 2: The index page of the Application

Note, that the uncertainty headers must contain the symbol '*' with the property name, e.g., '*hasMint' or uncertain*hasMint. In case of a misspelling or the name of the uncertain header dose not match the actual name of the property, for example: uncertainty header is 'uncertain*hasProductionPlace' and the actual property name is 'hasMint', or 'uncertain*hasClosingDate', while the actual property name is 'hasEndDate'.

Function uploadFile(): After selecting the input file and clicking on one of the 8 solutions to be generated, the process starts and the function uploadFile() will be invoked from the html flask Jinja.

Form for invoking the function uploadFile():

```
<form action="{{url_for('uploadFile', file=file)}}" method="post"
enctype="multipart/form-data" id="fm">
```

Submit Button for the Form:

```
<input type="submit" class="btn btn-outline-secondary btn-outline-info-
add" name="uploadFile" value="Generate Solution 1" id="s1"
onclick="loading_()"></th>
```

In the Form we set the action to use Jinja (a method flask uses to display the data) to call the function uploadFile(). We also use the method post to send the file with http request to the controller. We also set the value 'file' equal the upload file to read it in python. To run the function uploadFile() we implement an input as button and set its value 'value' of it to '*Generate Solution 1*'. When clicking on the submit button the function uploadFile() in class App will run and check if the method request is a 'post'. If yes, we need to identify which button has been clicked, therefore we use the following code line:

```
if request.form['uploadFile'] == "Create_solution 1":
    selectedG = 1
```

In this line of code, we check if the request form with the name 'uploadFile' is equal to 'Create_Solution 1'. If equal, we set the selected value 'selectedG' to 1, to generate the first solution. We repeat this step for each 'Generate Solution' button in the solution column, eight 'if' statements in total.

After determining the selected Graph, the file must be read inside Python. To avoid reading empty input we check first if the requested file is True, then we read the name of the file and check its extension if excel or csv. Depending on the file extension we build a save path as string and use the os library to access the path on local machine, then append the filename to the save path. If the input file is neither csv nor excel the program will not run and will instead be redirected to the index page.

```
if "csv" in str(filename_):
    file.save(os.path.join("csv",file.filename))
    current_path+= "csv/"
```

Next step is to create a script that contains some information to use it the classes readAndSeperate and Dynamic. The information to store in the script are the number of the selected graph, the name of the uploaded file, the current path where the file has been saved and the export-format 'RDF/XML' or 'Turtle'. We pack this information together in a dictionary in form of key value ("Key":"value"), then use the following code to write a json file in a directory.

```
with open("script/script.json",'w') as script:
    json.dump(dictionary,script)
```

The script.json is saved to be used for running classes readAndSeperate and Dynamic to create the solution. Now simply we run class readAndSeperat, then class Dynamic. It is very important to maintain this order in running class readAndSeperate first, because this class is responsible for preparing the Data to generate the solution. When this step is finished, we open the solution as file and store it in variable f_.

```
f_ = open("rdf/modelGraphDynamic_G"+str(selectedG)+".rdf","r")
```

Now we need to push the file back to the view. To do this, we use the method GET, which is responsible for transferring the data from the models to the view. The following *return* will carry the return page and the processed data.

```
return render_template('index.html', image=getImageList(), file=f)
```

When the process of generating the solution is finished, we run the index page, otherwise the server will break. To do that we use with the return statement the flask function render_template and set the return template to index.html. This command will redirect the index page directly from the template directory. We also push the variable image which contain the thumbnails of the solutions graphs and the file itself to be displayed in the view.

In case the use choses to generate more than one solution with the same data, then the data preparation process in the class readAndSeperate will only run on the first solution selection. Means, class readAndSeperate will only run if there is a data file been selected, as shown in **Figure A5.** ***3***. For any further solution selection (there is no new selected data file) the data preparation process will be skipped, and the class Dynamic for generating the solution will run directly.

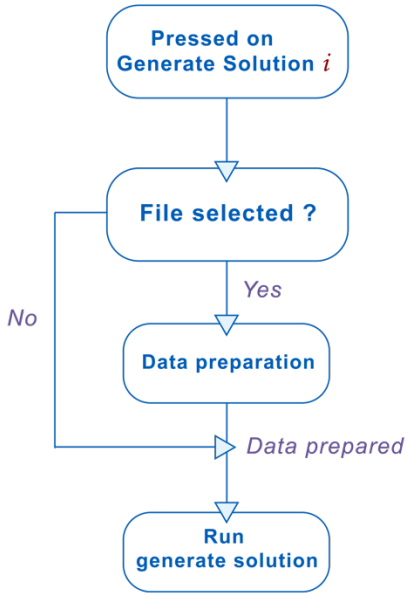


Figure A5. 3: In case there is no new data file selected, the data preparation process will be skipped, and the solution will be generated from the latest uploaded data file.

This step optimizes the run time of the application, for each additional selection can take up to half the run time as the first selection, especially with large data files. However, it requires changing the current selected graph in the json script file. Therefore, and before running the class Dynamic, we must open the script.json file and modify the current selected graph.

Figure A5. ***4*** illustrates SAUN showing all its Front- and Back-End connections with the technologies used.

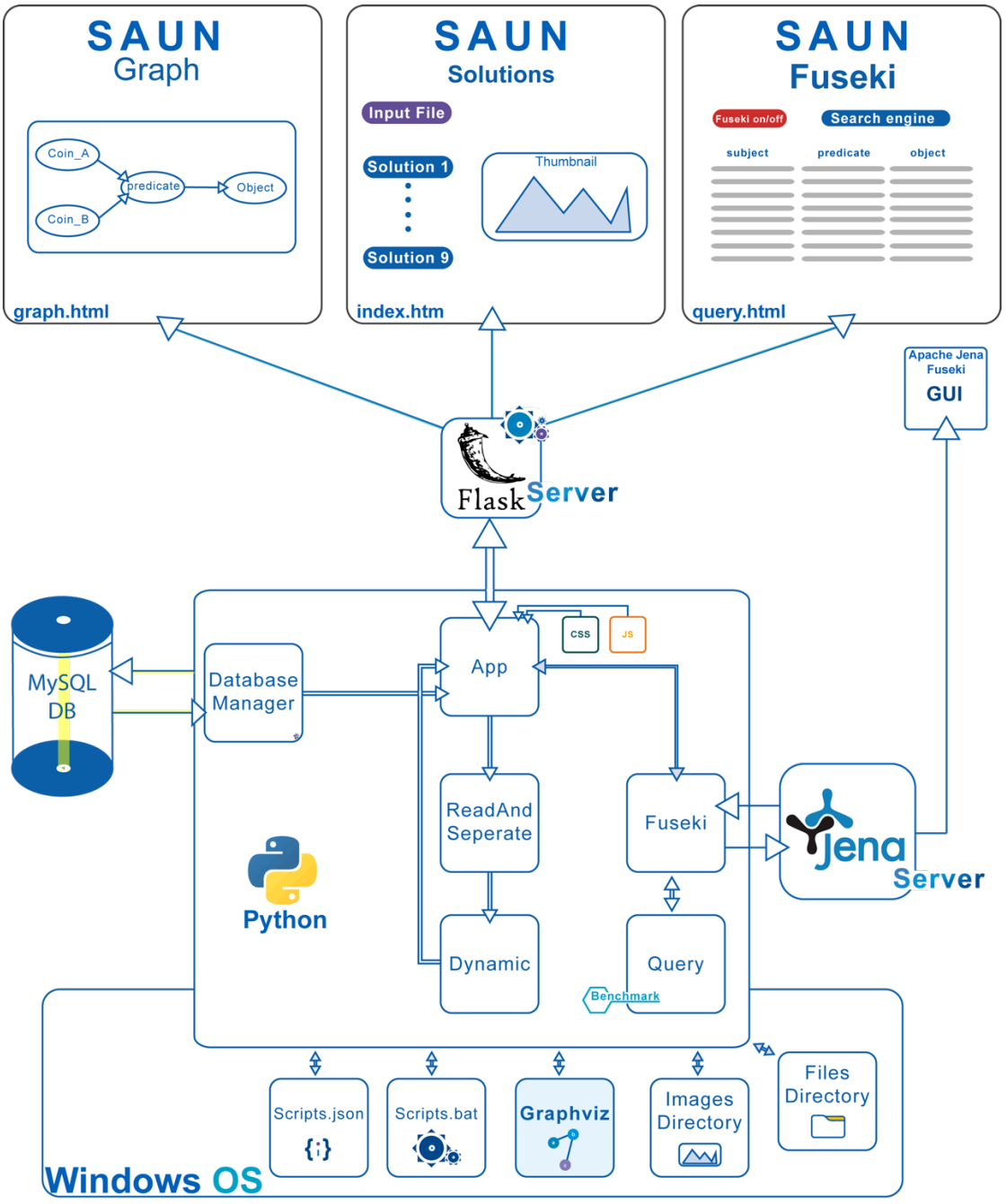


Figure A5. 4: Illustrates the web application SAUN, with its Front- and Back-End connections.

Running SPARQL and Retrieving Data

We implemented two classes (fuseki, queries) to simplify the process of running, stopping, updating, and getting data from the Fuseki server. For this purpose, we need to import some libraries like Requests, JSON, Socket, Subprocess, Psutil, PIPE and SPARQLWrapper.

Class fuseki is for managing the data and turning the server on and off. This class contains the following functions:

Fuseki_control(): This function is responsible for editing the server data. The data is simply some parameters that we already have defined and used in the class app. The data must be reachable every time the server starts, therefore, we create a dictionary file and include the data as follows:

```
dic = {"datasetName":jsonData['datasetName'], "ip": ""+ip, "port": port,
"dataName": ""+dataName}
```

Before we update the data script of the Fuseki server, we open the fuseki_data_script.json, located in the scripts directory, and read all the its data. We use the defined parameters to update the values in the dictionary. We use the function dump() of the JSON library to convert the dictionary into json file and finally we overwrite the faseki_data_script.json.

FusekiConnector(): This function is for running, terminating and uploading data to the Fuseki server. After updating the fuseki_data_script.json we now are able to run the server and upload data. We open the fuseki_data_script.json again and read the data the using function json.load(). The function FusekiConnector() has two conditions: first condition is that if the fuseki server is running and the port 3030 is open, then we turn it off. This can be done by accessing all open ports in the system using the following command:

```
Local_connections = psutil.net_connections('inet')
```

The variable local_connections will carry all the open ports in the system. Since we only are interested in getting the fuseki server port, which is by default 3030, we loop all ports in the vaiable local_connections looking for the fuseki port. When the port is found, we retrieve its

PID and call the following function of the psutil library to kill any process opened in this port:

```
psutil.Process(port).kill()
```

At this point we set the function to return false to indicate that the server is currently inactive. Second condition is that the server is not running. In this case we update the fuseki_run_script.bat with the new dataset name and run it using the function Popen():

```
subprocess.Popen('fuseki_run_script.bat')
```

Popen enable running .bat script using the cmd of the windows machine. The script includes the command of running the server and updating the data name. It is not recommended to run the server without creating dataset name, because if we need to use new data (rdf- xml- file) and run queries it will require restarting the server.

Now we have the fuseki server running on the specified port and the dataset name has been created. We need to upload the data to the server. This can be done by using the function subprocess.post(). The function post() takes several arguments, the first argument is the host, which is in our case is the http://localhost:. Next, we append the port and the dataset name to the host using the fuseki_data_script.json.

```
requests.post('http://localhost:'+str(jsonData['port']) + '/' +
str(jsonData['datasetName']), data=data.encode('utf-8'), headers=headers)
```

In the second argument we set the variable ‘data’ to the data that will be uploaded to the server. The third argument is where we set the variable ‘headers’ to equal the headers of the data. Fuseki can accept several types of headers. In our implementation we focus on ‘rdf’, therefore we use the following encoding to the headers:

```
{'Content-Type': 'application/rdf+xml;charset=utf-8'}
```

Function getData(): This function pushes a query to the fuseki server and retrieve its result. There are three important steps in this function. First step is to connect to the fuseki server using SPARQLWrapper with setting the host, port, and the dataset name in the connection path, example:

```
host = SPARQLWrapper('http://localhost:3030/dataOne')
```

In the above code line, we use the function SPARQLWrapper to connect to the localhost running on port 3030 and using the dataset name ‘dataOne’. Now we need to run the query on the fuseki server, therefore we store the connection in a variable named ‘host’ and use the following function to set the query for running:

```
host.setQuery(“Query!”)
```

Second step of the function getData() is to set the return format of the data retrieved from the server. Here we recommend using the JSON format, because it is faster at runtime than XML.

```
host.setReturnFormat(JSON)
```

In the third step we define a new variable called ‘queryResult’ and run the following command to retrieve the result in this variable:

```
queryResult = host.query().convert()
```

We also use the variable ‘host’ that contain the connection to the fuseki server and the function query(), which is responsible for running the queries on the server. Finally, we set the format of the result data to JSON using the function convert().

Now the variable ‘queryResult’ contain the header and the results of the query that we run on the server. To navigate these, we first specify, whether it is intended to only read the headers or the result.

For reading the header we use the following code:

```
for header in queryResult['head']['vars']:
```

And for the reading the result we simply replace `‘head’` and `‘vars’` with `‘result’` and `‘bindings’` respectively as follows:

```
for header in data['results']['bindings']:
```

Function dynamicTest(): the purpose of this function is to measure the runtime of each query of the eight solutions. Firs we check if the Fuseki server is in running mode, if otherwise we just call the function fusekiConnector() from the class fuseki, the server will take about 3 to 5 seconds to start. Therefore we delay the execution of the function dynamicTest(): for about 5 seconds, to make sure that the Fuseki server has started.

We created a for-loop (outer loop) that loops each of eight solutions. Inside the outer for-loop we create another for-loop (middle loop) that loops the five queries we have define in the class testQuery. Inside the middle for-loop we create another for-loop (inner loop) that repeat the process of measuring the execution time of each query five times.

To ensure an optimal state of the server and that only one file is uploaded to it, we turn off the Fuseki server at the end of the outer-loop by calling the function fusekiConnector() and delaying the function dynamicTest() by 5 seconds as well. This delay prevents the server from crashing.

Appendix 6: SKOS & OWL

SKOS: Simple Knowledge Organization System

In the previous section, we showed how to create a vocabulary (which also known by Ontology),

List 2. ***2***. We mentioned that ontologies are domain specific, meaning they are created to describe a specific area of knowledge, such as medicine, sports, cultural heritage, numismatics, etc.. There are many ontology definitions, we chose that by the W3C's Semantic Web Standard (W3C, Ontology, 2015) and (Yu, 2011):

„Ontologies define complex and formal collections of terms (concepts and relations) used to describe and represent an area of knowledge (a domain). "

The concepts are the classes and properties used to represent knowledge, and the relationships between these concepts can be represented as a hierarchical construct. Super-classes/-properties are at the top of the hierarchy (top-level concepts), and sub-classes/-properties are finer-level concepts. Therefore, ontologies offer a shared definition of main concepts and terms for creating PDF documents in a domain. As well as make the domain knowledge reusable and machine understandable.

So, how is knowledge expressed in an ontology? Knowledge is expressed by defining the terms of the ontology, i.e., defining the properties to be used and concepts, property-values can take. But the question is, how to enable the publication and use of ontologies to be linked together? for linked data being one of the main goals of the semantic web. The answer is SKOS "Simple Knowledge Organization System". What is SKOS? to answer this lets first see what KOS "Knowledge Organization System" is.

"KOS is a general term that refers, among other things, to a set of elements, often structured and controlled, that can be used to describe objects, index objects, search collections, etc." (Yu, 2011)

KOSs are often used where naming and classification are essential. Examples of KOS include, among others, taxonomies, thesauri, and classification schemes. Taxonomies refer

to the classification of things or concepts with hierarchical relations. Example, classifying living things (top-level) into humans, animals, and trees (fine-level) and further classification into even finer-levels, **Table A6. *1***.

Table A6. 1: Simple classification of living things.

Living Things									
Humans		**Animals**				**Plants**			
Male	Female	Vertebrates		Invertebrates		Non-flowering		Flowering	
Male individuals	Female individuals	Warm blood	Cold blood	Joined legs	No legs	Spore-baring	Naked seeds	1seed-leaf	2seed-leaves

Thesauri is a form of vocabulary that provides terms to give the ability to take taxonomies as described above, ordering subjects in a hierarchical way allow for further statements to be added. Examples of thesaurus-terms; BT broader term, NT narrower term, RT related term, TT top term, SN scope note, USE preferable term or label, UF inverse of USE. So, applying those terms on our taxonomy of living things will make for example, “Living things” a top-term for “Human”, “1 seed leaf” a narrower-term of “Flowering”, and “Vertebrates” has scope-note “animals with backbone”.

Taxonomies already use TT, BT, and NT to build the hierarchy, thesauri extend those terms with BT, RT, USE and UF. Therefore, a thesaurus can be understood as an extension of a taxonomy, or a taxonomy being a special thesaurus. And comparing the latter with ontologies, we can say that thesauri are used for organizing knowledge, while ontologies are used for representing it. Also, the descriptive power of thesauri is much less than that of ontologies. However, publishing KOSs to the Semantic Web is very beneficial particularly; to make the schemes be machine readable and optimally useable, to promote interoperability, and linking similar schemes to form a distributed, heterogeneous global concept scheme.

After we learned about KOSs, SKOS "Simple Knowledge Organization Systems" (Yu, 2011), with the word ‘Simple’ to state the simple yet powerful framework it offers.

“The name SKOS was chosen to emphasize the goal of providing a simple yet powerful framework for expressing knowledge organization systems in a machine-understandable way.” (Miles & Brickley, 2005)

SKOS is an RDF vocabulary for representing KOSs (taxonomies, thesauri, classification schemes, and more), and publishing them into the semantic web. SKOS is a W3C standard developed by the Semantic Web Development Working Group (SWDWG). The URI used to publish KOSs into the shared web is associated with namespace prefix skos: and is as follows:

```
http://www.w3.org/2004/02/skos/core#
```

OWL: Web Ontology Language

Web Ontology Language (OWL) a recommendation of the W3C for the purpose of defining ontologies. OWL is like RDFS and builds on top of it, except it adds new constructs for better expressiveness. This allows more complex class/property relationships to be built and stronger argumentation to be achieved. The first OWL version was released in 2004 and it has become a de facto standard for ontology development in many fields. However, due to ontology engineers and other users' point of view and requests for enhancements, OWL underwent optimization processes. The new optimized version become OWL 2 and the initial version is referred to as OWL 1. A definition of OWL 2 by the W3C's OWL 2 Web Ontology Language Primer (W3C, 2012):

"Web Ontology Language (OWL) presented by the W3C for the purpose of defining ontologies. OWL is like RDFS and builds on top of it, except it adds new constructs for better expressiveness. This allows more complex class/property relationships to be built and stronger argumentation to be achieved."

Even after the release of the updated OWL 2 version, OWL 1 is still used, also some applications can only understand ontologies created with OWL 1. And so, OWL 1 can be viewed as a subset of OWL 2, any application that understands OWL 2 can still understand ontologies built with OWL 1(meaning that, it can create a list of statements based on the ontology and express them as RDF statements). With that said, from now on when speaking about information/specifications applicable to both OWL 1 and OWL 2 we will refer to them as OWL, otherwise, we will specify the version.

To define classes and properties, OWL provide a set of pre-defined terms with the following leading URI, which is associated with namespace prefix owl:

```
http://www.w3.org/2002/07/owl#
```

Some of OWL's concepts

Axiom: An OWL axiom represents a statement, a piece of knowledge, and a given OWL ontology can be viewed as a collection of axioms.

Entity: Each axiom includes a class, a property, and an individual, also called entities. These entities can sometimes be referred to as categories (for class entity), relation (property entity), and object (for an individual entity).

Expressions: An expression is the combination of different classes- and properties entities to create new ones (new classes- and properties entities). With expressions, OWL gains more expressiveness compared to other ontology languages like RDFS.

IRI Names: OWL 2 provides the opportunity to use IRIs, which stands for Internationalized Resource Identifiers, for entities (classes, properties, and individuals) names. IRI are the same as URIs except that they can work with all Unicode characters, while URIs are limited to the ASCII characters. However, protocols like HTTP only work with UTIs, so, there is a standard way to convert IRIs to URIs and vice versa.

OWL specifications provide the possibility to first defined structural specification then choose one of various syntaxes for sharing ontologies. The Functional-Style syntax is a syntax for translating the structural specification to other syntaxes like RDF/XML-, Manchester-, and OWL/XML Syntax (Yu, 2011).

OWL's Classes and Properties

The root-class of OWL is owl:thing, which is also the base class of rdfs:Resource, and owl:class is a subclass of rdfs:class. **Figure A6. *1*** summarizes the relationship between these top classes.

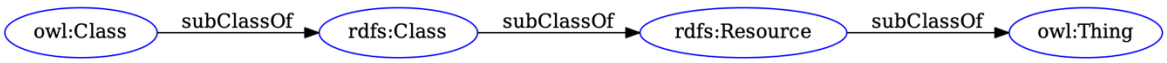


Figure A6. 1: The relationship between RDFS's and OWL's top classes.

Previously, in the ontology we created using RDFS, we have defined property_1 with domain= Class_1 and range= Class_2. We also defined Class_3 as a subclass of Class_1, meaning that it inherits property_1. Now, suppose we have defined a new class Class_4 being

a subclass of Class_2. With this, Class_4 inherits property_1 as well. This in fact will make statements like “instance of Class_1/ or Class_3 has relation property_1 to an instance of Class_2/ or Class_4” is possible. In case we would like to prevent this and perform more restrictions, allowing, as an example, for only this statement “instance of Class_3 has relation property_1 to an instance of Class_4” be valid, we can use owl:Restriction as shown in **List A6. *1***.

List A6. 1: Using owl:Class to define Class_3 being a subclass of Class_1, using owl:Restriction to specify that instances of Class_3 can only be associated to instance of Class_4 when using property_1.

```
1. <owl:Class rdf:about="&myVocab;Class_3">
2.   <rdfs:subClassOf rdf:resource="&myVocab;Class_1"/>
3.   <rdfs:subClassOf>
4.     <owl:Restriction>
5.       <owl:onProperty rdf:resource="&myVocab;property_1"/>
6.       <owl:allValuesFrom rdf:resource="&myVocab;Class_4"/>
7.     </owl:Restriction>
8.   </rdfs:subClassOf>
9. </owl:Class>
```

Lines 1-9 define Class_3 as an owl:Class, being an rdfs:subClassOf Class_1 (line 2), and has restrictions (ab line 4). In line 3 we create an empty node to add restriction to Class_3. Lines 4-7 uses owl:Restriction to describe an anonymous class, which is defined by adding restriction on some property using owl:onProperty (line 5) to specify the targeted property. To add constraints on the property itself we have two options; owl: value constraints and cardinality constraints. With value constraint we specify the range of the property (as we did in line 6, owl:allValuesFrom Class_4), the cardinality constraint restricts the number of values the property can take. **Figure A6. *2*** shows a graph representation of **List A6. *1***.

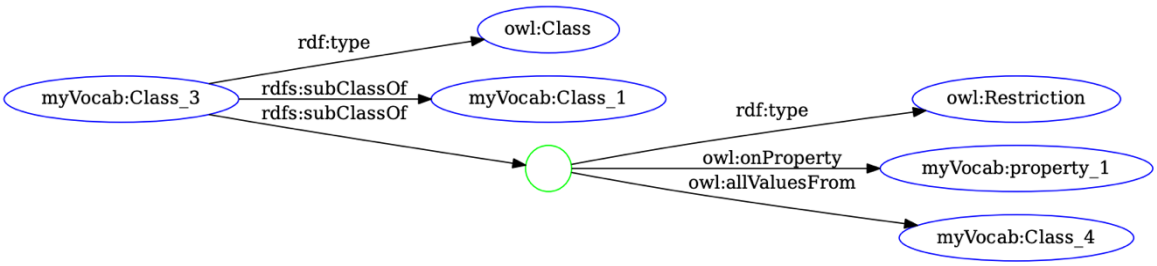


Figure A6. 2: graph representation of List (), adding restrictions allowing instances of Class_3 using property_1 to only be associated to an instance of Class_4.

Another OWL term to add value constraints owl:someValuesFrom. This, in contrast to owl:allValuesFrom, allows that property_1 to have instances from other classes as value, however, with the condition that at least one of the value an instance of the specified class, e.g., Class_4, as shown in **List A6. *2*** line 6.

List A6. 2: Restriction using value constraints with owl:someValuesFrom.

```
4.     <owl:Restriction>
5.       <owl:onProperty rdf:resource="&myVocab;property_1"/>
6.       <owl:someValuesFrom rdf:resource="&myVocab;Class_4"/>
7.     </owl:Restriction>
```

We will stop at this point with OWL, as we gave a simple idea of what it is and how it allows applying more restrictions on classes and properties. We would like to suggest this work (Yu, 2011) for any interests in learning more about OWL.