
# Uncertainty Reasoner for RDF Graphs based on Numismatic Use Cases

**Master Thesis**

Presented by
**Jens Weigel**
Submitted to
**Dr. Karsten Tolle**
For the degree
**Master of Science Informatik**

Fachbereich 12 Mathematik und Informatik
Johann Wolfgang Goethe-Universität Frankfurt am Main

15.07.2024

**Zusammenfassung**

RDF Graphen haben sich als nützliches Werkzeug in dem Feld der Numismatik etabliert, da RDF es erlaubt heterogene Daten von verteilten Münzsammlungen miteinander zu verbinden [7]. Ein Problem der Numismatik bleibt aber: Die Unsicherheit [41]. Diese Unsicherheit entsteht in der Numismatik z.B. durch korrodierte Münzen, widersprüchlichen Expertenmeinungen, oder vagen Zeitperioden. Vorhergehende Forschung hat sich schon angeschaut, wie man Unsicherheit in RDF graphen modelliert [30] und wie man diese berücksichtigt wenn man implizites Wissen aus den Graphgen abfragt, so genanntes Reasoning [25, 19, 42, 3].

In dieser Masterarbeit werden die Anwendungsfälle aus meinem Forschungsprojekt [45] nochmal betrachtet, es wird reevaluiert, ob die Methoden zum Lösen dieses Anwendungsfalls immer noch Sinn ergeben, und dann wird ein Reasoner implementiert, um diese Anwendungsfälle zu lösen. Dieser Reasoner soll leicht verständliche Methoden für das Reasoning anwenden und mit einer intuitiven API implementiert sein, sodass auch Menschen ohne viel IT-Wissen mit diesem Reasoner umgehen können. Dieser Reasoner wird am Ende auch nochmal mit realen Daten und synthetischen Daten auf die Laufzeit getestet.

# Contents

# Abbreviations

**AFE** Antike Fundmünzen in Europa

**AMT** Academic Meta Tool

**OWL** Web Ontology Language

**RDF** Resource Description Framework

**RDFS** Resource Description Framework Schema

**W3C** World Wide Web Consortium

# 1. Introduction

The Semantic Web aims to extend the World Wide Web by providing machine-interpretable metadata about data, enabling computers to make meaningful interpretations of the processed data [46]. Semantic Web technologies and concepts such as the Resource Description Framework (RDF) graphs and ontologies have proved to be valuable tools to unify distributed coin databases for the numismatic [7]. One of the problems that still persists, is the uncertainty when, for example, handling ancient coinage [41]. This uncertainty may stem from corroded coins, conflicting expert classifications, or vaguely defined time periods. Recent research already evaluated different approaches to model uncertainty within RDF graphs [30] and how to deal with uncertainty during reasoning (inferring implicit knowledge which is encoded inside a graph) [25, 19, 42, 3].

In my research paper [45], I already defined three possible numismatic use cases that could be solved using current uncertain reasoning approaches. This thesis will define uncertainty and vagueness, evaluate those use cases again, and then implement a reasoner to solve them. Since the target user group of this reasoner includes non-it researchers, the three main requirements for this reasoner are:

- Simplicity: The approaches, used in this reasoner, should solve the problems but also be simple enough to explain to non-it researchers.
- Expandability: The reasoner should be easy to expand, enabling users to implement their own reasoning approaches and uncertainty modeling techniques.
- Easy-To-Use: The reasoner should be accessible using an intuitive API and with as few lines of code as possible. It should support a wide variety of graphs without modifying the graphs.

Firstly, subsection 2.1 will introduce the basics of the Semantic Web and RDF. Additionally, subsection 2.1.5 shows a broad overview of classical RDF reasoners, and two common approaches for modeling uncertainty are introduced in subsection 2.1.6. Then, subsection 2.2 explains four different ways to reason in uncertain RDF graphs.

The numismatic use cases are defined in section 3. Before defining the use cases, subsection 3.1 defines *uncertainty* and *vagueness*. Additionally, this subsection also explains how you can deal with vagueness and uncertainty, and the limitations of the approaches used in this thesis. Then, the three use cases from the research project are split into smaller problems and then tried to solve using the reasoning approaches from the section before.

The actual reasoner implementation is documented in section 4. This section implements the reasoning needed to solve the use cases from section 3. The reasoner is then benchmarked

for the different use cases using synthetic data in section 5. In the end, section 6 summarizes the findings of this thesis and outlines the possible future research in this domain.

# 2. Preliminaries

This section is about the semantic web basics (subsection 2.1) and the current approaches for reasoning in uncertain RDF graphs (subsection 2.2).

## 2.1. Semantic Web

This subsection is about the Semantic Web and is based on [14] until subsection 2.1.6. The Semantic Web, an idea by Tim Berners-Lee, is an extension of the World Wide Web through standards set by the World Wide Web Consortium (W3C)[1]. The main goal of the Semantic Web is to make data on the Internet machine-readable in a way that queries containing semantic statements are possible. The key concepts and technologies involved in the Semantic Web are RDF (subsection 2.1.1), RDFS (subsection 2.1.1), OWL (subsection 2.1.3) and SPARQL (subsection 2.1.4). After defining the RDF basics, subsection 2.1.5 shows different approaches and frameworks to reason in RDF graphs. Then subsection 2.1.6 describes two ways to model uncertainty in RDF graphs.

### 2.1.1. RDF

The Resource Description Framework (RDF) is a standard model for describing the meaning (semantics) of data and data interchange on the web. While the common XML format represents data in a tree structure, RDF describes the data as a directed graph. The tree structure of XML may be good for representing hierarchical structures and working with them efficiently but are too restricting when working with more general relationships between objects of interest. You cannot, for example, derive a hierarchy from the information "Ben eats Pizza" but you can represent it as a graph with Ben and Pizza being nodes, and eats being a directed edge from Ben to Pizza. The graph structure also helps when merging data. You can simply include the nodes and edges from one graph in the other to get a bigger graph. Merging tree structures requires more work and even then, it could be impossible because the tree nodes are in different locations and cannot be connected while maintaining the tree structure.

While merging data and maintaining a graph structure is possible, there is another problem to be tackled: Actually merging the knowledge inside the graphs. When you merge two graphs with a node `Ben`, you do not know whether those graphs refer to the same person. That is why RDF uses distinct Uniform Resource Identifiers (URIs) like `http://www.example.org/1999/02/22-rdf-syntax-ns#type`. Comparable to Uniform Resource Locators (URLs), which are also URIs and already used in the World Wide Web to distinguish internet documents, URIs allow

[1] `https://www.w3.org/` (last accessed 10.07.2024)

two graphs to refer to the same objects. In RDF everything, except literals and blank nodes, has a distinct URI.

Due to lengthy URIs, you often assign *namespaces* prefixes to shorten URIs. For example, the prefix `rdf` is assigned to the namespace `http://www.w3.org/1999/02/22-rdf-syntax-ns#`. This allows the usage of abbreviated URIs which are called *qualified names* (QNames). The namespaces and prefixes of this thesis can be found in Appendix A. Using `rdf:type` with the declared namespace for rdf now means `http://www.w3.org/1999/02/22-rdf-syntax-ns#type`. In Figure 2, you can see the namespace declaration using the `prefix` keyword. Since you are allowed to use prefixes like rdf, abbreviated URIs are not enclosed in angular brackets to distinguish them from full URIs.

Literals are nodes containing concrete data values such as strings, numbers, or dates. You can use literals to store information, that has no real use for a URI, like the name of a book or the weight. There are untyped literals, that are always read as strings, and typed literals where you specify the datatype. Using the prefix `xsd` for the namespace `http://www.w3.org/2001/XMLSchema#` you can specify the type of the literal `"6.0"` as a decimal by using two caret symbols: `"6.0"^^xsd:decimal`. Smaller RDF graphs can be visualized using tools like the *RDF Grapher*[2] which is also used for the RDF graphs in this thesis.

The Relationships between RDF nodes are described using triples in the form of (Subject,

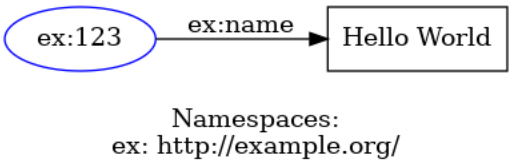


Figure 1: Example RDF graph with a URI node and a literal node.

Predicate, Object). Each RDF statement (or triple) consists of three parts:

- Subject: The resource being described.
- Predicate: The property or characteristic of the subject.
- Object: The value of the property.

In Figure 1 you can see a simple example with two nodes and one property. In the statement `(ex:123, ex:name, "Hello World") ex:123` (full URI `<http://www.example.org/123>`) is a URI node, `Hello World` is a literal node and `ex:name` is the property of the statement. This graph can be represented using Turtle[3] triples like in Figure 2.

[2] `https://www.ldf.fi/service/rdf-grapher` (last accessed 10.07.2024)
[3] `https://www.w3.org/TR/turtle/` (last accessed 10.07.2024)

```
@prefix ex: <http://example.org/> .

ex:123 ex:name "Hello World" .
```

Figure 2: Graph from Figure 1 as Turtle triples.

While XML stores the data of the document in a hierarchical way, you can use it to describe

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:ex="http://www.example.org/">
 <rdf:Description rdf:about="http://www.example.org/123">
      <ex:name>Hello World</ex:name>
    </rdf:Description>
</rdf:RDF>
```

Figure 3: Example from Figure 1 as XML/RDF.

RDF graphs using XML/RDF, since XML only provides the document structure. The example from Figure 1 can also be written in RDF/XML as you can see in Figure 3. In the `rdf` tag, you can see the namespace declarations via `xmlns`. Even though the Turtle variant looks more intuitive than the RDF/XML representation, almost every programming language offers a standard library for XML while Turtle libraries are rare. That made XML/RDF the main syntax for RDF.

RDF also supports blank nodes which are no literals but also have no URIs. Blank nodes are auxiliary nodes to represent many-valued relationships as you can see in Figure 4. Many-valued relationship means that `ex:hasPrice` represents a connection between three values: The `ex:food123` node, the money value, and the currency. Combining both value and currency into one literal like "8.0 Euro" would be impossible to query, when searching for example all items with a Euro price below 10 Euros. Since RDF only allows connection with two values (subject and object) you can just add the blank node. This helper node does not need a URI because it is just an extension of `ex:hasPrice`. Now you can type the money value as a `xsd:decimal` and keep the currency as a string. We will use blank nodes later in the Reification subsection to model uncertainty in RDF graphs.

To further specify an individual you can assign them to a class via `rdf:type`. The triple `ex:food123 rdf:type ex:Pizza` states that `ex:food123` is an instance of the class `ex:Pizza`.

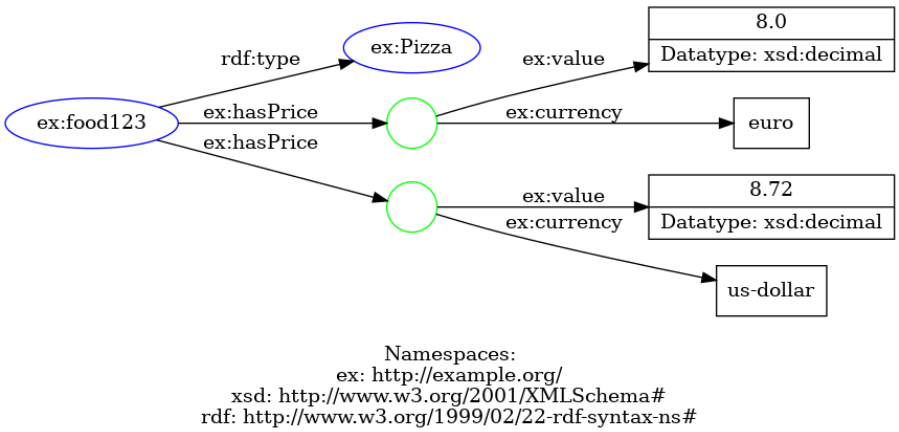


Figure 4: Example RDF graph with a blank node.

### 2.1.2. RDFS

The Resource Description Framework Schema (RDFS) extends the RDF vocabulary by providing a way to describe *schema knowledge* about the classes and properties (predicates) used in your graph. It enables the creation of simple ontologies which are a description of knowledge about a domain. The RDFS namespace `http://www.w3.org/2000/01/rdf-schema#` is usually abbreviated with the prefix `rdfs`. RDFS introduces class and property hierarchies with `rdfs:Class`, `rdfs:subClassOf`, and `rdfs:subPropertyOf`, and allows the further description of properties by defining subject (`rdfs:domain`) and object (`rdfs:range`) node classes. RDFS also provides ways to add human-readable descriptions to resources through `rdfs:label` and `rdfs:comment`.

In Figure 5 you can see a simple RDFS ontology with two classes `ex:Cat` and `ex:Bird` be-

```
@prefix ex: <http://example.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

ex:Animal rdf:type rdfs:Class .
ex:Cat rdfs:subClassOf ex:Animal .
ex:Bird rdfs:subClassOf ex:Animal .
ex:eat rdf:type rdf:Property .
ex:eat rdfs:domain ex:Animal .
```

Figure 5: Example RDFS ontology as Turtle triples.

ing a subclass of the more general `ex:Animal` class. There is also the property `ex:eat` which should only use `ex:Animal` classes as subjects. Since both `ex:Cat` and `ex:Bird` are subclasses of `ex:Animal` they can be used as well.

Additionally, RDFS offers various ways to represent lists. It supports ordered lists, unordered lists, and a set of alternatives via `rdf:Seq`, `rdf:Bag`, and `rdf:Alt`.

### 2.1.3. OWL

The Web Ontology Language (OWL) further extends the RDF and RDFS vocabulary, allowing the creation of more complex ontologies. OWL uses the prefix `owl` for the namespace `http://www.w3.org/2002/07/owl#`. There are three sublanguages, also called *species*, of OWL with different expressiveness and computational complexity: OWL Full, OWL DL and OWL Lite. OWL Full is the most expressive, containing both OWL DL and OWL Lite, but also undecidable when inferring implicit knowledge from the graph. That is why the RDF reasoner from subsection 2.1.5 usually only supports OWL DL and OWL Lite.

Like RDF and RDFS, OWL has classes and individuals (instances of classes) but with support for more complex class definitions. OWL provides an extended vocabulary for specifying cardinality constraints on properties and describing properties as, for example, symmetric, transitive, or inverse relationships. With OWL you can, for example, model the predicate `ex:knows` as a symmetric and transitive property. With the triples `ex:Ben ex:knows ex:Simon` and `ex:Ben ex:knows ex:Clara` you can now infer new triples such as `ex:Simon ex:knows ex:Ben` (symmetric) and `ex:Simon ex:knows ex:Clara` (transitive).

OWL can also express equivalence between classes, properties or individuals. This helps when joining RDF graphs from different ontologies. If one ontology uses the class name "Place" and the other one "Location" you can declare them equivalent using `owl:equivalentClass`.

OWL's more expressive vocabulary allows users to gain more implicit knowledge from RDF graphs using the reasoning tools from subsection 2.1.5.

### 2.1.4. SPARQL

SPARQL Protocol and RDF Query Language (SPARQL) is the main query language for accessing and manipulating data from RDF graphs that are stored in triple stores like Apache Jena[4]. SPARQL is inspired by SQL but also fundamentally different since SPARQL operates on a graph stored in triples.

A simple SPARQL select query like Figure 6 first defines the namespace prefixes with the PREFIX keyword similar to the Turtle annotation. Then you can select certain variables from the WHERE statement for the result (?coin and ?cointype) with SELECT. You can also use * to select all variables. The WHERE clause defines the actual statement of the query using a *simple graph pattern*. In the WHERE part, you can use URIs, QNames, and *variable identifiers*

[4] `https://jena.apache.org/` (last accessed 10.07.2024)

like ?coin to define Turtle triples that represent the desired pattern. The example query searches coins, minted in Rome, and their coin type.

The example query states that the coins in the result must have a coin type. If you want to

```
PREFIX ex: <http://example.org/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?coin ?cointype
WHERE {
  ?coin rdf:type ex:Coin .
  ?coin ex:hasMint ex:rome .
  ?coin ex:hasCointType ?cointype
}
```

Figure 6: Example SPARQL query.

include coins without a coin type you can use the OPTIONAL keyword like `OPTIONAL{?coin ex:hasCointType ?cointype}`. You can also utilize the `UNION` keyword to join two patterns. If you replace `?coin ex:hasMint ex:rome` in the example query with `{?coin ex:hasMint ex:rome} UNION {?coin ex:hasMint ex:gargara}` the results contain coins minted in Rome or Gargara.

Referencing blank nodes is also possible by addressing them with "_:x" and x being a positive

```
PREFIX ex: <http://example.org/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?pizza ?value
WHERE {
  ?pizza rdf:type ex:Pizza .
  ?pizza ex:hasPrice _:1 .
  _:1 ex:currency "euro" .
  _:1 ex:value ?value .
  FILTER(?value < 10.0) .
}
```

Figure 7: Example SPARQL query for a graph like Figure 4.

number. In Figure 7 you can see an example query for a graph like Figure 4. The query searches all Pizzas that cost below 10 Euros. Using `?pizza ex:hasPrice _:1` you access the blank node with the currency and value. The triple `_:1 ex:currency "euro"` now uses the same blank node to search for Euro prices the `FILTER` keyword filters for values below 10.

### 2.1.5. RDF Reasoner

Reasoning in the context of RDF graphs is inferring new triples from a given graph using, for example, rule-based reasoning. One simple rule could be: "If Mary is the wife of Ben, then Ben is the husband of Mary". This means, that given the triple `ex:Ben ex:hasWife ex:Mary`, we can infer the triple `ex:Mary ex:hasHusband ex:Ben`.

In [6], the authors benchmarked five reasoners from 3 different groups. The first group is the tableaux-based OWL DL reasoners like Racer [11] and Pellet [28]. The second group uses a disjunctive datalog engine like the commercial reasoner KAON2[5]. The final class is rule-based reasoners like Sesame[6] and OWLIM[7] [18]. It highlighted the selection of the reasoning approach according to the reasoning problem.

In the newer paper [31], you can find a comparison and benchmark of rule-based reasoning approaches like the Jena Inference Engine [33], the Euler YAP Engine[43], and BaseVISor[22]. In Figure 8 you can see the feature comparison of the approaches.

The popular Open-Source ontology editor Protégé[8] also already includes two reasoners: ELK

| | Jena | EYE | BaseVISor |
|---|---|---|---|
| **Reasoning Strategies** | | | |
| Forward Chaining | ✓ | ✓ | ✓ |
| Backward Chaining | ✓ | ✓ | - |
| **Reasoning Algorithms** | RETE | Euler Path Detection, YAP Prolog engine | RETE |
| **RDFS/OWL Reasoning** | RDFS, OWL Lite | selected predicates of RDFS and OWL | OWL2-RL |
| **Rule Languages** | Own format | Notation3, RIF-BLD | Own format |
| **Production Operations** | | | |
| Assertion | ✓ | ✓ | ✓ |
| Retraction | ✓ | - | ✓ |
| **Built-in Functions** | ✓ | ✓ | ✓ |
| **User-defined Functions** | ✓ | ✓ | ✓ |
| **Proof Explanation** | ✓ | ✓ | - |
| **Software License** | Apache License 2.0 | MIT License | Free for academic/ research use |

Figure 8: Rule-based RDF reasoner comparison. From [31].

[17] and HermiT [36]. The Protégé documentation[9] provides a nice example to test these reasoners on an example pizza ontology. Opening the ontology with Protégé allows you to infer

[5]Seemingly discontinued as the URL to their website is dead
[6]Now RDF4J [35]
[7]Now GraphDB [26]
[8]`https://protege.stanford.edu/` (last accessed 10.07.2024)
[9]`http://protegeproject.github.io/protege/getting-started/` (last accessed 10.07.2024)

new triples by selecting Reasoner/HermiT (or ELK respectively) and starting the reasoning. The tool then highlights inferred knowledge with a yellow background. You can also display the whole new inferred class hierarchy.

In terms of expressiveness, ELK supports a subset of OWL EL[10] and aims for full OWL EL support in the future. However, there is currently no up-to-date list of currently supported features for ELK 0.5.0. Only the features of the older version 0.4.0 are available[11]. HermiT allows the whole set of OWL 2 [10] features and also the creation of custom DL-safe rules [9].

### 2.1.6. Modelling Uncertainty in RDF Graphs

There are several different ways to model uncertainty in RDF Graphs and in [30], the author benchmarked the most common approaches. One usual approach is called reification and uses a blank node. In Figure 9 on the left, you can see the triple `ex:coin1 nmo:hasMint nm:rome` modeled with the certainty value (`ex:certaintyValue`) 0.8 and the source (`ex:accordingTo`) `ex:model1` using reification. You need 6 or 5 triples with this approach depending on excluding the `rdf:type` triple or not.

The benchmarks showed the best runtime results on another approach which is called RDF-

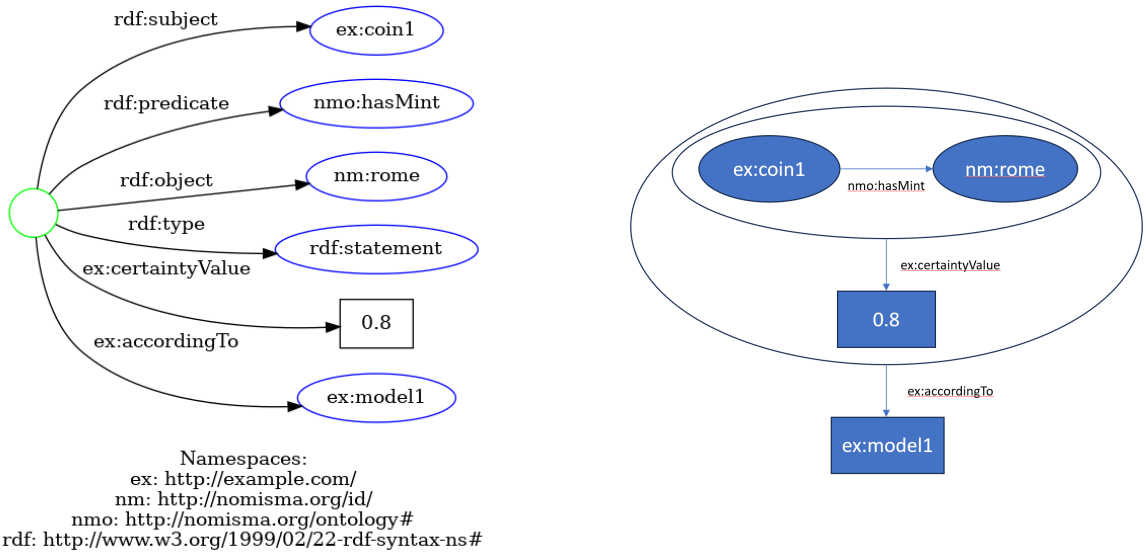


Figure 9: Different approaches for modeling uncertainty. A reification example from [45] on the left and the same example in RDF-star on the right.

star. RDF-star is an extension of RDF which allows the creation of statements about statements [32]. It is currently only a draft and not part of the W3C standards but already been implemented in Open-Source software like Apache Jena [13]. In Figure 9 on the right, you can see the

[10] `https://www.w3.org/TR/owl2-profiles/#OWL_2_EL` (last accessed 10.07.2024)
[11] `https://github.com/liveontologies/elk-reasoner/wiki/OwlFeatures` (last accessed 10.07.2024)

same example in RDF-star. This graph can also be written in Turtle-star triples as you can see in Figure 10. The triple `« ex:coin1 nmo:hasMint nm:rome » ex:certaintyValue "0.8"` further describes the triple `ex:coin1 nmo:hasMint nm:rome` and `ex:accordingTo ex:model1` describes the certainty value. Therefore RDF-star only needs 3 triples to model the given use case: The statement itself and two statements about this statement. In my research project [45], the Turtle-star example is wrong because both `ex:certaintyValue` and `ex:accordingTo` describe the triple `ex:coin1 nmo:hasMint nm:rome`. Therefore, you do not actually know which certainty value is from which model.

```
@prefix ex: <http://example.com/> .
@prefix nm: <http://nomisma.org/id/> .
@prefix nmo: <http://nomisma.org/ontology#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

<< << ex:coin1 nmo:hasMint nm:rome >> ex:certaintyValue "0.8" >> ex:accordingTo
    ex:model1 .
```

Figure 10: Example from Figure 9 as Turtle-star.

## 2.2. Reasoning in Uncertain RDF Graphs

In this subsection, we will look at current approaches and tools for reasoning in uncertain RDF graphs. Firstly, there is an approximative query-time reasoning tool presented in subsection 2.2.1. Then there is fuzzy reasoning from subsection 2.2.2 which utilizes fuzzy logic and a subset of OWL for reasoning. The context-aware reasoning in subsection 2.2.3 describes a way to reason in RDF graphs for assisted living sensors. The last approach is the Academic Meta Tool, which uses property chain rules to infer new triples, from subsection 2.2.4.

### 2.2.1. Query-Time Reasoning in Uncertain RDF Knowledge Bases with Soft and Hard Rules

In the paper [25], the authors present their framework URDF to resolve inconsistencies and derive new RDF triples from existing ones using deduction rules at query-time. These inconsistencies may stem from automated information extraction which is used to fill large semantic knowledge bases like DBpedia [5] or YAGO [38]. This automated extraction is often probabilistic or heuristic and thus prone to adding noise and incorrect triples and additionally, the data may still be incomplete.

Consistency constraints and keeping the triple with the highest confidence have been shown to be a good way to resolve conflicts. However, the challenge is to balance the need for high-confidence data while avoiding to only use high-confidence triples because some triples could receiver lower confidence when dealing with incomplete data. Deduction rules, which are often also uncertain themselves, can be used for reasoning over uncertain facts to produce likely or speculative results when critical knowledge is missing. The paper introduces the URDF framework to address several requirements for query-time reasoning over uncertain RDF triples like providing answers to complex SPARQL queries, overcoming incompleteness using deduction rules, considering consistency constraints for noise reduction and achieving high efficiency for interactive response times.

To achieve this, URDF uses a set of facts (the triples in the graph), a set of Horn clauses with confidence assignments (soft rules), and a collection of consistency constraints (hard rules). An example soft rule could state: "If Alex and Luke live in Frankfurt and Alex and Luke both play football, then Alex and Luke know each other with a confidence of 0.2". Additionally, also the non-derived base facts in the knowledge base are assigned a weight which can be modeled using a soft rule like "Alex plays football with a confidence of 0.9". To enforce consistency, a rule such as "no person can have multiple birth or death dates" might be implemented. After defining the soft and hard rules, the goal is to maximize the confidence of the soft rules by marking the facts as either true or false. Therefore, the facts, soft and hard rules are given to a modified weighted MaxSAT algorithm. This algorithm is inspired by the original weighted MaxSAT which did not include consistency constraints [16]. For fast response times the algorithm provides only an approximation rather than an exact solution since finding the exact maximum is a NP-hard problem [15].

To benchmark URDF, the authors run queries on the YAGO knowledge base. They then compare the resulting query times and weights to other current state-of-the-art solutions based on Markov Chain Monte Carlo, such as MC-SAT [29] and MAP inference [37]. In those benchmarks, URDF showed significant performance improvements while achieving similar weights.

### 2.2.2. Fuzzy Reasoning over RDF Data using OWL Vocabulary

Another approach is fuzzy reasoning from [19]. For reasoning, the authors use a subset of the OWL vocabulary, called pD* semantics, to extend the RDFS semantics, which was introduced by [39]. The pD* fragment provides a full set of entailment rules while being able to be calculated in polynomial time. In this paper, the authors use the fuzzy logic introduced in [12] to adapt the pD* semantics to vague statements and thus present a new way of reasoning in uncertain RDF graphs.

Firstly, fuzzy RDF triples in the form of (s, p, o)[n] are introduced as RDF triples with a fuzzy

$$
\begin{array}{ll}
I(\phi \wedge \psi) = I(\phi) \otimes I(\psi) & I(\phi \vee \psi) = I(\phi) \oplus I(\psi) \\
I(\phi \rightarrow \psi) = I(\phi) \Rightarrow I(\psi) & I(\neg\phi) = \ominus I(\phi) \\
I(\exists x.\phi(x)) = \sup_{c \in \Delta^I} I(\phi(c)) & I(\forall x.\psi(x)) = \inf_{c \in \Delta^I} I(\phi(c))
\end{array}
$$

Figure 11: Fuzzy interpretation extension. From [19].

degree (degree of truth) n assigned to them. This means, the triple (s, p, o) is true to a degree of at least n. An example fuzzy triple could be `(Alex, plays, football)[0.9]`. Classical logic only supports statements that are either true or false. Therefore, they use fuzzy logic [12] to deal with statements assigned a fuzzy degree between 0 and 1. In Figure 11 you can see the fuzzy interpretation of classical logic statements. The $I$ function maps the statement to its fuzzy degree and the symbols $\oplus$, $\otimes$ and $\ominus$ are combination functions like, for example, triangular norms. Some of the important combination functions can be seen in Figure 12.

Using the fuzzy logic the fuzzy entailment rules can be defined as seen in Figure 13. Every

| | Lukasiewicz Logic | Gödel Logic | Product Logic |
|---|---|---|---|
| $a \otimes b$ | $\max(a+b-1, 0)$ | $\min(a, b)$ | $a \cdot b$ |
| $a \oplus b$ | $\min(a+b, 1)$ | $\max(a, b)$ | $a+b-a \cdot b$ |
| $a \Rightarrow b$ | $\min(1-a+b, 1)$ | $\begin{cases} 1 & \text{if } a \leq b \\ b & \text{otherwise} \end{cases}$ | $\min(1, b/a)$ |
| $\ominus a$ | $1-a$ | $\begin{cases} 1 & \text{if } a = 0 \\ 0 & \text{otherwise} \end{cases}$ | $\begin{cases} 1 & \text{if } a = 0 \\ 0 & \text{otherwise} \end{cases}$ |

Figure 12: Example combination functions of various fuzzy logics. From [19].

entailment rule has a condition, a conclusion, and sometimes an additional constraint. The sets $U$ and $B$ represent the sets of URI references (U) and Blank nodes (B). The entailment rules can be utilized to derive already existing and new fuzzy RDF triples from a fuzzy graph (a graph with fuzzy RDF triples). In this context, the Best Degree Bound (BDB) of a triple (s, p, o) defines the highest fuzzy degree n which can be derived from the fuzzy graph using the entailment rules. The authors showed that a partial fuzzy pD* closure, containing only BDB triples, always exists and can be computed in polynomial time. A prototype of a framework using fuzzy reasoning and MapReduce can be seen in a later paper [20].

### 2.2.3. Uncertainty Handling in Semantic Reasoning for Accurate Context Understanding

The authors of the next paper [3] reason in RDF graphs with contextual sensor information. The paper aims to improve ambient assisted living systems to tackle the increasing number of dependent patients with chronic and age-related diseases. Ambient assisted living systems use

| | Condition | Constraint | Conclusion |
|---|---|---|---|
| f-rdf1 | $(v, p, w)[n]$ | | $(p, \texttt{type}, \texttt{Property})[1]$ |
| f-rdfs2 | $(p, \texttt{domain}, u)[n]$ $(v, p, w)[m]$ | | $(v, \texttt{type}, u)[n \otimes m]$ |
| f-rdfp4 | $(p, \texttt{type}, \texttt{TransitiveProperty})[n]$ | | |
| | $(u, p, v)[m]$ $(v, p, w)[l]$ | | $(u, p, w)[n \otimes m \otimes l]$ |
| f-rdfp5a | $(v, p, w)[n]$ | | $(v, \texttt{sameAs}, v)[1]$ |
| f-rdfp5b | $(v, p, w)[n]$ | | $(w, \texttt{sameAs}, w)[1]$ |
| f-rdfp6 | $(v, \texttt{sameAs}, w)[n]$ | $w \in U \cup B$ | $(w, \texttt{sameAs}, v)[n]$ |
| f-rdfp7 | $(u, \texttt{sameAs}, v)[n]$ $(v, \texttt{sameAs}, w)[m]$ | | $(u, \texttt{sameAs}, w)[n \otimes m]$ |
| f-rdfp11 | $(u, p, v)[n]$ $(u, \texttt{sameAs}, u')[m]$ $(v, \texttt{sameAs}, v')[l]$ | $u' \in U \cup B$ | $(u', p, v')[n \otimes m \otimes l]$ |

Figure 13: Part of the fuzzy entailment rules. From [19].

technology like sensors, which are prone to hardware failure, to help dependent persons live in their homes [4]. In previous research [2], the authors analysed the ambient assisted living system decision against caregiver observations for given contexts. They concluded that in 29% of cases, the observation differed from the system decision, and it was mostly because of the sensors having low battery or communication issues.

To include the uncertainty from the sensor's hardware issues, the authors define an uncertainty

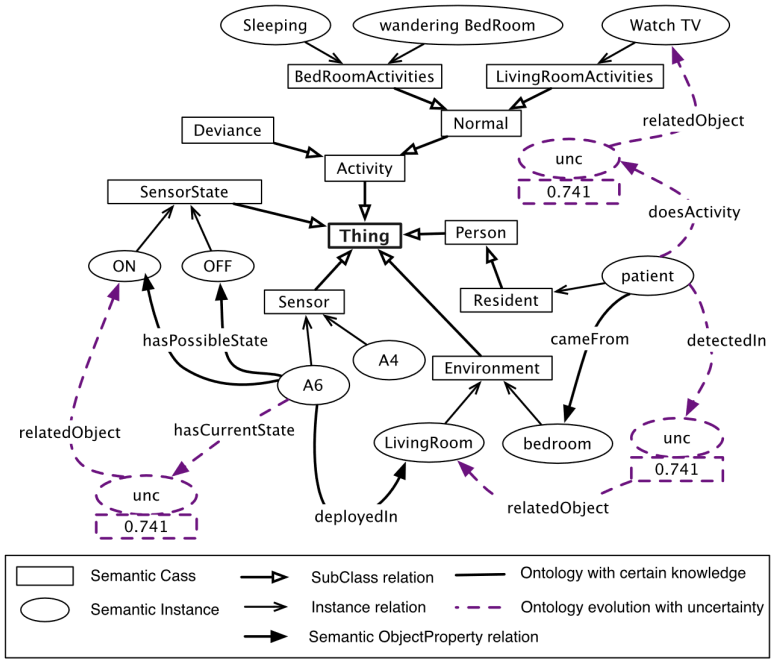


Figure 14: Context aware ontology. From [3].

measurement that is based on hardware characteristics, like battery level, and use-case-related operational parameters. The uncertainty measurement is then used in the RDF graph of Figure 14 to include the confidence in the sensor that creates an event. The sensor `A6`, which is located in the living room, has a confidence level of 0.741, calculated by the uncertainty measurement, and is currently firing events. Therefore, the patient is in the living room with

a certainty of 0.741.

To resolve conflicts among sensor events, the authors use the Dempster-Shafer Theory [34] to combine the evidence from multiple sources. In the Dempster-Shafer Theory, you have a set of atomic mutually-exclusive outcomes or states, the frame of discernment, and can assign mass values to sets (hypotheses) of its power set. In this case, the subsets contain the contexts that can be detected by the sensors and the mass values are the calculated uncertainty measures. Each sensor represents one mass function with the mass value being the uncertainty measurement based on hardware characteristics. To combine multiple observations from different sensors, you can use Equation 1 from [3] to combine their mass functions. With the combined mass function you can then calculate the plausibility, which represents the degree to which the hypothesis is possible, using the Equation 2 from [3].

$$m_c(H) = \frac{\sum_{\forall X,Y:X\cap Y=H} m_1(X).m_2(Y)}{1 - \sum_{\forall X,Y:X\cap Y=\emptyset} m_1(X).m_2(Y)} \tag{1}$$

$$\text{Plausibility}(H) = \sum_{H'|H\cap H'\neq\emptyset} m_c(H') \tag{2}$$

In Figure 15, you can see an example for a sensor conflict. If subsets contain more than one

| Context/doesActivity | A4 mass function | A6 mass function | P1 mass function |
|---|---|---|---|
| sleeping | 0.0 | 0.0 | 0.741 |
| showering | 0.0 | 0.0 | 0.0 |
| wanderingBedroom | 0.0 | 0.0 | 0.0 |
| wanderingLivingroom | 0.0 | 0.0 | 0.0 |
| watchTV | 0.0 | 0.0 | 0 |
| sleeping, wanderingBedroom | 0.923 | 0.0 | 0.0 |
| watchTV, wanderingLivingroom | 0.0 | 0.743 | 0.0 |
| ignorance | 0.077 | 0.257 | 0.259 |

Figure 15: Mass functions from the sensors for activity recognition. From [3].

element like {sleeping, wanderingBedroom}, then there is evidence for both activities and the patient is doing one of those things. The activities are the subsets, and each sensor assigns a mass value to each subset. The ignorance is the set of all possible states or outcomes (the set of discernment), and will be explained later. Using Equation 1 the combined mass function can be calculated which is shown in Figure 16. Then, the plausibility values can be derived using Equation 2 which is shown in Figure 17. In this example, the authors conclude that the patient is sleeping with a confidence of 94.5%.

To evaluate their approach the authors tested it using a Monte Carlo simulation and in a real setting. During these tests they used highly-coupled sensors, that detect one activity, and low-coupled sensors (for example sensor P1 in Figure 15), that can detect multiple possible activities without knowing which exactly (for example sensors A4 and A6 in Figure 15). The simulation showed that using not enough high coupled sensors lowers the chance of detecting

| Context/doesActivity | A4, A6, P1 mass function |
|---|---|
| wanderingBedroom,sleeping | 0.226 |
| wanderingLivingroom,watchTV | 0.054 |
| sleeping | 0.7 |
| ignorance | 0.018 |

Figure 16: Combined mass functions from the sensors for activity recognition. From [3].

the exact activity the person is doing but using the right amount of low-coupled sensors together with high-coupled ones can improve the performance. They also conclude that using the right hardware characteristics is crucial, so faulty sensors are detected and get lower mass values when the evidence is combined.

| Context/doesActivity | A4, A6, P1 plausibility |
|---|---|
| wanderingBedroom | 0.244 |
| sleeping | 0.945 |
| wanderingLivingroom | 0.073 |
| watchTV | 0.073 |
| showering | 0.018 |

Figure 17: Plausibility for the example from Figure 16. From [3].

In [34], the authors further explain the meaning of the mass functions and the plausibility. The mass function also called *basic probability assignment* (bpa), does not assign a classical probability to the subsets. The mass value given to a set A does not include any information about the subsets of A. These subsets would need an additional mass value assigned to them. There is also another measure besides the mass function and the plausibility which is called the belief. The belief can be calculated from the mass function like in Equation 3 from [34]. The belief and the plausibility are closely related as you can see in Equation 4 from [34]. $\overline{H}$ is the complement of $H$.

$$\text{Belief}(H) = \sum_{H' | H' \subseteq H} m_c(H') \tag{3}$$

$$\text{Plausibility}(H) = 1 - \text{Belief}(\overline{H}) \tag{4}$$

The plausibility and the belief are non-additive and are therefore not required to add up to one. The precise probability of a subset is somewhere between their belief (lower bound) and their plausibility (upper bound). This means, that if the belief and the plausibility of a subset are equal, you have the precise probability of this subset.

### 2.2.4. Academic Meta Tool

The final approach discussed is the Academic Meta Tool (AMT) from [42]. In their paper, the authors define vagueness as a measure of the precision of statements where the statements are true to a certain degree. Uncertainty, on the other hand, means that the statement is either true or false. In AMT the RDF statements are assigned a vagueness from 1 to 0 which are used in context-dependent rule-based reasoning to infer new statements that also have vagueness attached to them.

Firstly, you define node categories, which are called concepts, similar to `owl:Class`. Then the

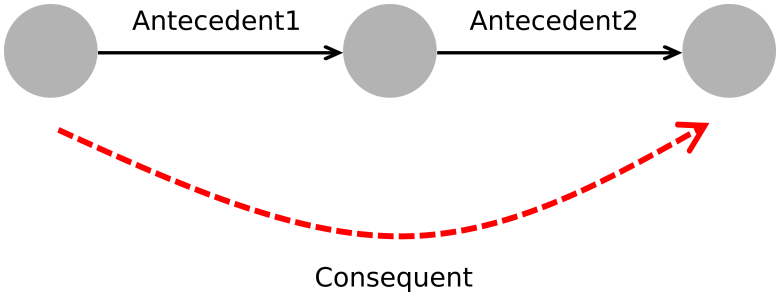


Figure 18: Chain rule axiom illustration. From [42].

statement categories, called roles, which are like owl:ObjectProperty. Every role needs a source (like rdfs:domain) and target (like rdfs:range) concept. After defining your ontology, you can define the axioms of your use case which define the rules for the reasoning. There are two types of axioms in AMT: The chain rule axiom and the inverse axiom. The chain rule axiom uses two roles between three concepts to derive a new role as you can see in Figure 18. It is similar to owl:ObjectPropertyChain but the AMT chain rule also defines the resulting role. Additionally. you need to choose Goedel, Lukasiewicz, or product logic as the reasoning logic [21] for the chain rule. The inverse axiom defines inverse roles between concepts, for example `isParent` and `isChild`, as you can see in Figure 19. There can also be constraints like the disjoint and self-disjoint axioms.

Before you can use the AMT on your own data, you need to download the code from the

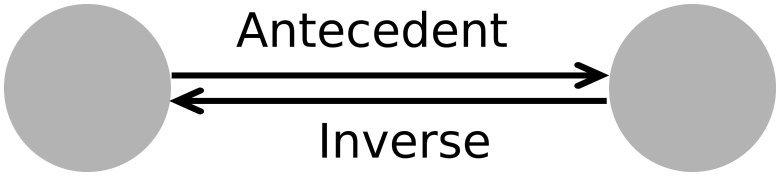


Figure 19: Inverse axiom illustration. From [42].

author's Github[12] and modify the amt.js JavaScript file from, for example, the "Samian" folder

[12] `https://github.com/mainzed/academicmetatool-viewer` (last accessed 10.07.2024)

so it uses your triple store like Apache Jena Fuseki[13]. Additionally, you may need to add certain prefixes to the queries in the code because you get an error otherwise. Then you need to modify the *index.html* to work with your use case or create your own HTML file and call the functions from amt.js. After this, you define the concepts, roles, and axioms as RDF triples like the example Turtle files in their Github. You load those triples into your triple store and next, you can use the AMT web interface to add data (if you implemented this) or add the data directly to the endpoint again. You can now use the AMT reasoning to infer new triples. Therefore, using the AMT requires a lot of work to modify the code right now. This thesis will implement the AMT chain rule reasoning with an easy-to-use API that does not require modifying the code.

[13] `https://jena.apache.org/` (last accessed 10.07.2024)

# 3. Numismatic Use Cases

In this section, we will look at numismatic use cases we will solve with the reasoner. We will use modified versions of the use cases defined in [45] and split them into smaller problems. A small summary of all use cases is shown in Table 1. The use cases are sorted by their complexity and some use cases extend other use cases. Subsection 3.1 starts with defining uncertainty and vagueness for this thesis and how to use the AMT chain rule reasoning for uncertain statements. The first use case is introduced in subsection 3.2 where similarity values from multiple models are aggregated. Sometimes there are no certainty values in the data but only possible alternatives. Therefore, the next use case from subsection 3.3 assigns each of the possible alternatives a certainty value. The third use case in subsection 3.4 describes using the Dempster-Shafer Theory for aggregating certainty values from multiple models. The fourth use case from subsection 3.5 extends the second use case and uses the Dempster-Shafer Theory in a highly use-case-dependent setting. The fifth use case from subsection 3.6 explores the usage of AMT chain rules for the numismatic. It utilizes a modified set of axioms to allow the usage of uncertain statements and not only vague ones like the AMT. The last use case, introduced in subsection 3.7, tries utilizing the Dempster-Shafer Theory to determine the most probable minting intervals of coins.

| Use Case | Short Description | Extends Use Cases | Complexity |
|---|---|---|---|
| 1 | Aggregate similarity values from multiple models. | - | 1 |
| 2 | Assign certainties to AFE issuer and issuing for alternatives. | - | 1 |
| 3 | Aggregate coin classification certainties from multiple models. | - | 2 |
| 4 | Combine AFE issuer selection evidence with domain knowledge evidence. | 2 | 2+ |
| 5 | Coin and coin type similarity chain rule reasoning. | 1,3 | 3 |
| 6 | Infer most probable minting time interval of a coin. | - | undefined |

Table 1: Table of all use cases. Complexity ranges from 1 to 3 with a plus indicating a slightly higher complexity.

## 3.1. Uncertainty and Vagueness

The authors of the AMT tool defined vagueness in their paper as: *"Vagueness is a measure of precision of a statement. A vague statement is only true to a certain degree."*[42]. Their

example of vagueness is the statement "It will rain tomorrow". They argue that the statement does not exactly tell you how much rain is expected but there will be rain tomorrow. You could further specify the statement by adding how many hours of rain are expected. However, the statement "There will be rain tomorrow" could already be a complete statement by itself without adding the amount of rain or the rain duration.

The AMT definition of vagueness is based on the *fuzzy logic* [12]. In [19] from subsection 2.2.2, the authors defined *fuzzy statements*, which are also vague, as statements in the form of $\phi[n]$ where $\phi$ is the statement and the statement $\phi$ is true to a degree of at least n. This n is called the *fuzzy degree* of the statement and is between 0 and 1. Similarly, *fuzzy triples* are in the form (s, p, o)[n], meaning the triple (s, p, o) is true to a degree of at least n. The fuzzy logic from Figure 11 is then used to interpret boolean operations with fuzzy statements.

A better example of vagueness is in [21] which is also referenced in the AMT paper: *"A standard example that points out the difference between degrees of uncertainty and degrees of truth is that of a bottle [8]. In terms of binary truth values, a bottle is viewed as full or empty. But if one accounts for the quantity of liquid in the bottle, one may, e.g. say that the bottle is 'half-full'. Under this way of speaking, 'full' becomes a fuzzy predicate [47] and the degree of truth of 'the bottle is full' reflects the amount of liquid in the bottle. The situation is quite different when expressing our ignorance about whether the bottle is either full or empty (given that we know that only one of the two situations is the true one). Saying that the probability that the bottle is full is 0.5 does not mean that the bottle is half full."*[21]. This is a rather simple example because the boundaries for 0 (the bottle is empty) and 1 (the bottle is completely full) are easy to define. Additionally, the degree of truth of the statement is easy to get by measuring the liquid in the bottle and dividing it by the total possible liquid in the bottle.

The AMT definition of uncertainty is: *"In the case of uncertainty, it is completely unknown whether the statement made is true at all."*[42]. Their example statement "There will be rain tomorrow with a 75% probability" is an uncertain statement because the core statement "there will be rain tomorrow" is either true or false with a 75% probability of being true. Therefore, the authors define vagueness and uncertainty as different things and their article only uses vague statements for the use cases.

In a W3C report [44], they classified uncertainty into five different subtypes with vagueness being one of those subtypes of uncertainty:

- **Ambiguity:** The entities in the statement are not clearly specified (for example the statement "Tobi eats pizza" when there are multiple persons with the name Tobi and not all eat pizza).
- **Empirical:** It is currently not possible to determine if the statement is true or not and

additional knowledge (like experiments) is required.

  - **Randomness:** There is statistical randomness determining whether the statement is true or not (like the statement "There will be rain tomorrow with a 75% probability").

- **Vagueness:** The statement is not precise enough (for example the statement "It will rain tomorrow" does not state how much rain is expected).

- **Inconsistency:** There is conflicting information (for example a coin that is classified as two different coin types by two experts or models while there can only be one coin type per coin).

- **Incompleteness:** The uncertainty stems from missing information (for example a corroded coin that can not be clearly classified).

However, there are some problems with this interpretation of uncertainty and vagueness. Usually, there are multiple types of uncertainty in each use case. Incompleteness can lead to Inconsistency because the missing information leads to conflicting results when working with the information. For example, a coin with high corrosion is hard to classify (Incompleteness) that could lead to conflicting classifications from multiple experts (Inconsistencies). This means there are often multiple sources of uncertainty involved and it is therefore hard to classify the uncertainty. Additionally, vagueness and uncertainty work differently and therefore it makes no sense to define vagueness as a subclass of uncertainty. There are statements you can assign with a probability of being true and some, where it makes no sense. For example, the statement "Coin A looks a bit similar to coin 2" does not mean that there is a small chance of coins A and B looking completely the same. It means, that those coins look similar to a certain degree.

Therefore, this thesis acknowledges the differences between uncertainty and vagueness. *Uncertainty* stems from statements that are either true or false and it acts like a probability of the statement's truthfulness. The *certainty* of an uncertain statement is the probability of the statement's truthfulness. *Vagueness*, on the other hand, is a measure of precision of a statement. A higher vagueness means that the statement is less precise and vice versa. This thesis defines the *precision degree* (fuzzy degree) as the precision and thus the intensity of the statement on a scale from 0 to 1. The precision degree $n$ means, that the statement is true to a degree of at least n and therefore represents the lower bound of the statement's truthfulness. A precision degree of zero means that the statement is false and a precision degree of one that the statement is true. You could, for example, assign the statement "Coin A looks a bit similar to coin 2" with a precision degree of 0.2 for "a bit similar" because the core statement "Coin A looks similar to coin 2" is only true to a lesser degree (at least 0.2) and is therefore less precise.

In this example, a precision degree of one would define two coins as totally similar. The actual meaning of "A is totally similar to B" depends on the use case and context.

In the bottle example from [21], you could also define a new predicate "half-full" to describe the bottle as half-full. However, doing so would create an endless amount of triples, because everyone creates a new predicate to describe their bottles: "one-third-full", "one-sixth-full" etc. Additionally, you may not want to measure the exact amount each time. Therefore, you define the statement "The bottle is full" as a fuzzy statement and assign a precision degree (fuzzy degree) of 0.5. This way, you can keep using the predicate "full" and handle vagueness stemming from inexact measuring by using a fuzzy predicate. Thus, defining a predicate as a fuzzy predicate changes the meaning of the predicate. In this example, the meaning of the predicate "full" shifts to "at least full to a degree of n" with n being the precision degree 0.5. Nonetheless, you also lose information when using fuzzy statements. The reasoning logic used in this thesis does not allow to also define an upper bound to the statement's truthfulness. This way, "exactly half-full" and "half-full to two-thirds-full" would both get the same fuzzy degree of 0.5. Future research could focus on reasoning logic that can also handle upper boundaries instead of only lower boundaries.

The question is how you map statements to precision degrees. The example with the bottle was an easy example but the coin similarity example is trickier because it is subjective how a person defines something like "a bit similar". If two persons are asked to measure "a bit similar" on a 0 to 1 scale, person A might use 0.25 and person B might use 0.2. The persons may also use different metrics to measure coin similarity. Person A focuses on the depicted person while person B concentrates on the coin legend. So it is crucial to define more objective measures, that depend on the use case, and how the scale should be used. In the case of similarity, this can be quite challenging since there is no objective measure. There are machine learning models that may produce similarity values from 0 to 1 depending on, for example, visual features, but it is unclear what exactly leads to these values.

While the authors of the AMT only used vague use cases in their paper, this thesis will also solve use cases that contain uncertainty and vagueness using the AMT chain rule reasoning. However, this thesis still only supports statements that are either vague or uncertain and not both. Future research could explore reasoning with statements that are both uncertain and vague. Three cases need to be considered when allowing uncertainty and vagueness for the chain rule reasoning: vague-vague, uncertain-uncertain, and vague-uncertain. The first one is when both antecedent statements are vague like in [42]. The authors of the AMT already defined how to deal with this type of chain rule. The second one is where both antecedent statements are uncertain. The concept of chain rules where both antecedents are uncertain statements does make sense when using the product rule as the reasoning logic and as long as

both antecedent statements are independent of each other:

$$P(\text{Consequent}) = P(\text{Antecedent1} \cap \text{Antecedent2}) = P(\text{Antecedent1}) \cdot P(\text{Antecedent2})$$

This means the independence of the uncertainty of Antecedent1 and Antecedent2 is crucial when combining two uncertain statements otherwise you would need to subtract the joint probability which complicates chain rule reasoning. Additionally, it is unknown whether the usage of the Lukasiewicz or Goedel reasoning logic makes sense when handling two uncertain statements in the chain rule. Therefore, this thesis will only use the product rule for this type of chain rule. Nevertheless, the reasoner will not enforce any restrictions regarding the usage of another reasoning logic. It is up to the user to decide which logic is applicable to their use case.

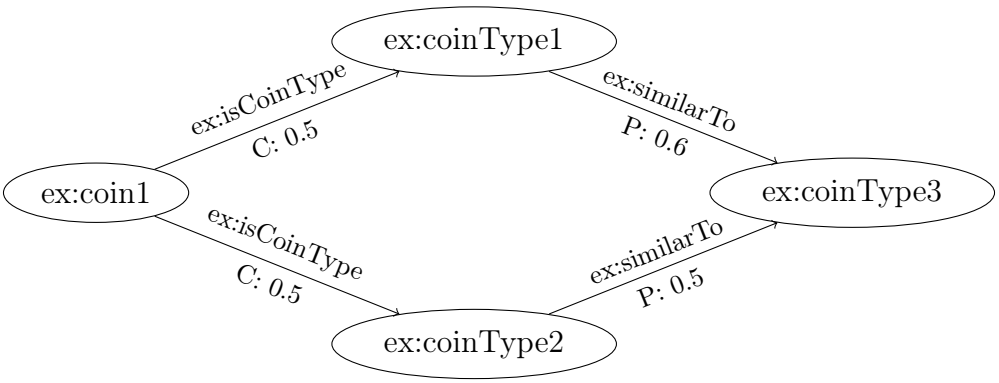


Figure 20: Mixing uncertain and vague statements. `ex:similarTo` is a vague statement and ex:`isCoinType` is an uncertain one. C stands for certainty and P for precision degree.

Combining uncertain and vague statements is a bit trickier since you combine a probability with a precision degree. The consequent triple of such a chain rule can be interpreted as an expected value for a precision degree if you modify the chain rule reasoning and stick to the product rule reasoning. For example, there is a chain rule that states "If `A ex:isCoinType B` and `B ex:similarTo C` then `A ex:similarTo C`" with `ex:isCoinType` being an uncertain predicate and `ex:similarTo` a vague predicate. The predicate `ex:similarTo` means that coins and coin types look similar like in the examples before. In Figure 20, you can see the example graph with certainty and precision degree values. The standard AMT chain rule reasoning would only consider the maximum certainty value for `ex:coin1 ex:similarTo ex:coinType3` that can be derived using the chain rule: $0.5 \cdot 0.6 = 0.3$. However, for the expected value you need to sum both viable paths and thus get the precision degree $0.5 \cdot 0.6 + 0.5 \cdot 0.5 = 0.55$.

In Figure 21, you can see the updated graph with the consequent triple. The Lukasiewicz or Goedel reasoning logic does not make sense for these chain rules since we calculate an expected value and thus need the product of the precision degree and the probability (certainty). Whether

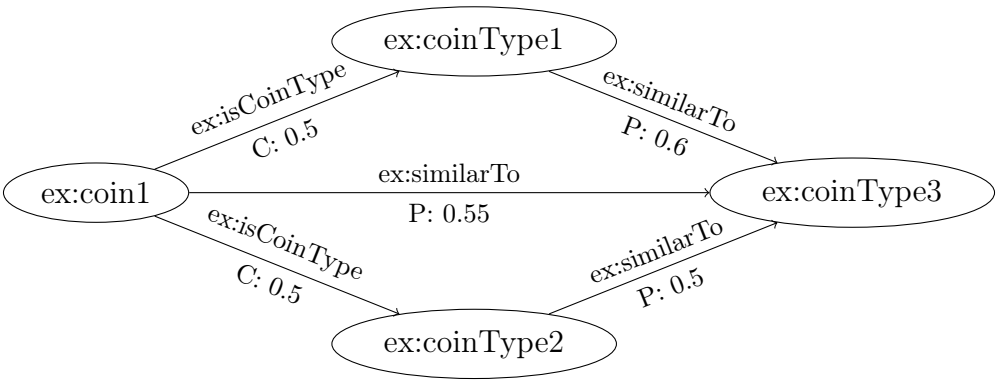


Figure 21: Example from Figure 20 with the inferred consequent triple. `ex:similarTo` is a vague statement and ex:`isCoinType` is an uncertain one. C stands for certainty and P for precision degree.

the antecedent statements are vague-uncertain or uncertain-vague makes no difference for this use case since you sum up the products of all viable paths and the product rule is commutative ($A \cdot B = B \cdot A$).

However, one problem that arises is shown in Figure 22. The defined chain rule would derive `ex:coin1 ex:similarTo excoinType2` with a vagueness of only $0.5 \cdot 0.8 = 0.4$ even though there is a 50% chance that ex:coin1 is ex:coinType2. To tackle this, you can define a rule to filter those during post-processing or even add a constraint to the chain rule reasoning. The rule could state "If there is ex:isCoinType present then there should be no ex:similarTo".

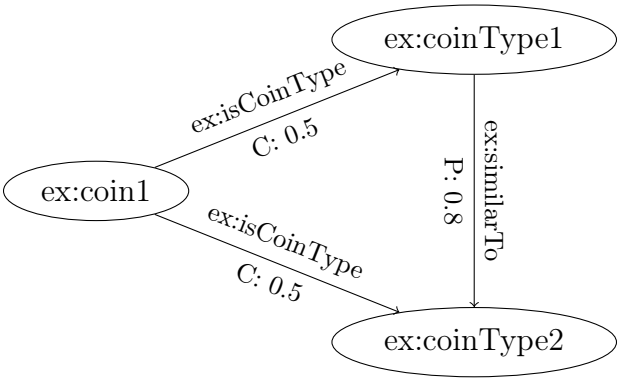


Figure 22: Problematic example for chain rule reasoning. `ex:similarTo` is a vague statement and ex:`isCoinType` is an uncertain one. C stands for certainty and P for precision degree.

One major problem of dealing with uncertainty and vagueness is the context. In the use case from subsection 3.6, the first intuition is to define `ex:similarTo` as a transitive property using the product rule as the reasoning logic. However, this is just a simplification of the problem since this similarity chain rule highly depends on the context. For example, Coin1 is similar to

Coin2 because of some similar text on the coin. Coin2 and Coin3 are similar because of a similar picture of the coin. Does this mean Coin1 and Coin3 are similar? It depends on the features that make up the similarity between the coins. In this case, the preferred reasoning logic would be Lukasiewicz rather than the Product logic. Lukasiewicz logic for two precision degrees $a, b$ is defined as $\max(a+b-1, 0)$. For example, Coin1 is similar to Coin2 with a precision degree of 0.9 and Coin2 is similar to Coin3 with a precision degree of 0.9. Using the Lukasiewicz reasoning logic, we can infer that Coin1 is similar to Coin3 of at least $\max(0.9 + 0.8 - 1, 0) = 0.7$. This would be the lower bound of the resulting similarity if the 0.1 that differentiates Coin1 and Coin2 is part of the similarity between Coin2 and Coin3.

This means that defining AMT chain rules with statements, that are assigned with uncertainty, need to be done with caution. In my research paper [45], I defined the chain rule "If A is coin type B and B similar to C then A is coin type C" and therefore inferred an uncertain statement using a vague and an uncertain statement. In hindsight, it makes no sense to infer an uncertain statement by combining a vague with an uncertain antecedent statement. The correct inferred statement is "A similar to C" and you need to sum the values of all the viable paths for this use case.

Since the mass values of the Dempster-Shafer Theory are not classical probabilities, we need to define how to deal with them. There are two types of mass functions in this thesis: One-element mass functions and time interval mass functions. The one-element mass functions only contain subsets with one element or with the whole set of discernment (the *ignorance*). There will be another paragraph to describe why we need this ignorance subset. The belief of a one-element subset A in these mass functions is their mass values because you can not derive another subset $A' \subseteq A$ except $\emptyset$ which will not be used in the mass functions. The plausibility of A is the sum of the mass values $m(A)$ and $m$(ignorance). The actual probability for A is between the belief and the plausibility. Therefore, in this case, you can use the mass value of A as a lower-bound estimate for the probability (certainty) of A like in Equation 5. The ignorance subset is a special case. It contains all the singular elements and thus has a belief of 1.0. However, setting the probability of the ignorance subset to one minus the probability of all the one-element subsets will result in the mass value of the ignorance as you can see in Equation 6. Thus, the probability of the ignorance subset represents the maximum error that the actual probabilities may differ from their lower bound estimates. An ignorance mass value of 0.11 as you can see in Table 3 means, that the total sum of error between the joint mass values of A and B, and their actual probabilities is equal to 0.11.

$$p(H) \approx \text{Belief}(H) = m(H) \tag{5}$$

$$p(\text{ignorance}) = 1 - \sum_{H|H \neq \text{ignorance}} p(H) \approx 1 - \sum_{H|H \neq \text{ignorance}} m(H) = m(\text{ignorance}) \tag{6}$$

The time interval mass functions use subsets that contain a continuous interval with natural numbers. In this case, you can not use the mass value of a subset A as the lower-bound estimate of the probability because the subsets $A' \subseteq A$ may also have mass values. This increases the belief and thus the lower bound of the probability of A.

In my research project [45], I already found potential use cases for the Dempster-Shafer Theory but also some flaws. For example, if the mass functions only contain disjoint subsets like in Table 2, then the combined mass value for each subset is zero. To solve this, you can

| Subset | Mass Function 1 | Mass Function 2 | Joint Mass | Plausibility |
|---|---|---|---|---|
| {A} | 1.0 | 0 | 0 | 0 |
| {B} | 0 | 1.0 | 0 | 0 |

Table 2: Problematic example with disjoint subsets for the Dempster-Shafer Theory.

assign a mass value to the whole frame of discernment like the authors of [3] did in Figure 15 with their `ignorance` subset. This ignorance subset represents the amount of trust you have in the model, that made this selection. More trust means less ignorance and vice versa. In Table 3 you can see the updated example.

| Subset | Mass Function 1 | Mass Function 2 | Joint Mass | Belief | Plausibility |
|---|---|---|---|---|---|
| {A} | 0.8 | 0 | 0.44 | 0.44 | 0.56 |
| {B} | 0 | 0.8 | 0.44 | 0.44 | 0.56 |
| {A, B} | 0.2 | 0.2 | 0.11 | 1.0 | 1.0 |

Table 3: Using a set of all elements of the frame of discernment to tackle problematic example with disjoint subsets for the Dempster-Shafer Theory.

## 3.2. Use Case 1: Similarity Values from Multiple Models

The first use case is to handle the vagueness from similarity detection using multiple models. In the numismatic, one task is to categorize coins into coin types. However, some of those coin types are almost the same as shown in Figure 23 with two coins from OCRE[14]. It can be helpful to show experts similar coin types to those they currently selected. In this use case, there are coin types that have similarity relationships detected by machine learning models. Therefore,

[14] `http://numismatics.org/ocre/` (last accessed 10.07.2024)

each model assigns the similarity predicate a precision degree which indicates how similar the coin types are according to the model. These precision degrees can be calculated by various tools and methods [40] and you may want to store them separately in the RDF Graph. The task of this use case is to aggregate those precision degrees from multiple models into one value per coin type similarity triple. Then, this value can be used to show similar coin types to the selected ones.

For example, the only distinction between the two coin types in Figure 23 is the missing "V"[15]

Figure 23: Example for similar coin types with OCRE coins. `ric.1(2).aug.4A` at the top (picture from [23]) and `ric.1(2).aug.5` (picture from [24]) at the bottom. The legend at the top left is "IMP CAESAR AVGVST" and at the bottom left "IMP CAESAR AVGVSTVS". From [45].

in the legend. That is why it might be helpful to calculate similarity values between coin types and offer similar coin types to users that classify coins. The expert selects one coin type of those two example coin types and the other one is shown as a suggestion. The expert can then make a more informed decision about whether the current coin type is the right one or whether one of the suggested coin types is more likely.

The objective is now to take these similarity values from different models and aggregate them into one value. This can be done using simple aggregation functions like mean and median. Due to similarity being a vague statement and thus being true to a certain degree, the similarity values from incoming or outgoing similarity triples do not need to add up to 1. ex:coinType1 can be similar to ex:coinType2 with a precision degree of 0.9 while also being similar to ex:coinType3 with a precision degree of 0.8. Thus, you can not use the Dempster-Shafer Theory from subsection 2.2.3 for this use case.

[15] `http://numismatics.org/ocre/id/ric.1(2).aug.5` (last accessed 10.07.2024) states that the observe legend of ric.1(2).aug.5 is "IMP CAESAR AVGVSTV" instead of "IMP CAESAR AVGVSTVS" like in Figure 23 at the bottom. This could be a typo or the extra "S" is not required for the classification of this coin type.

## 3.3. Use Case 2: Translating AFE Issuer and Depicted Person Relationships into Certainty Values

In [41], the authors present their relational database *Antike Fundmünzen in Europa* (AFE) [1] to handle uncertainty when classifying coins. In Figure 24, you can see the web mask for an example coin. This use case will focus on the fields "Issuing for" (the depicted person on the coin) and issuer (who issued the coin). To model the uncertainty of classifying corroded coins AFE allows the selection of multiple issuers and depicted persons through the alternative fields. Additionally, the users can use the checkbox "uncertain" to state that they have low confidence in their alternatives containing the real issuer or depicted person. Therefore, AFE selection of

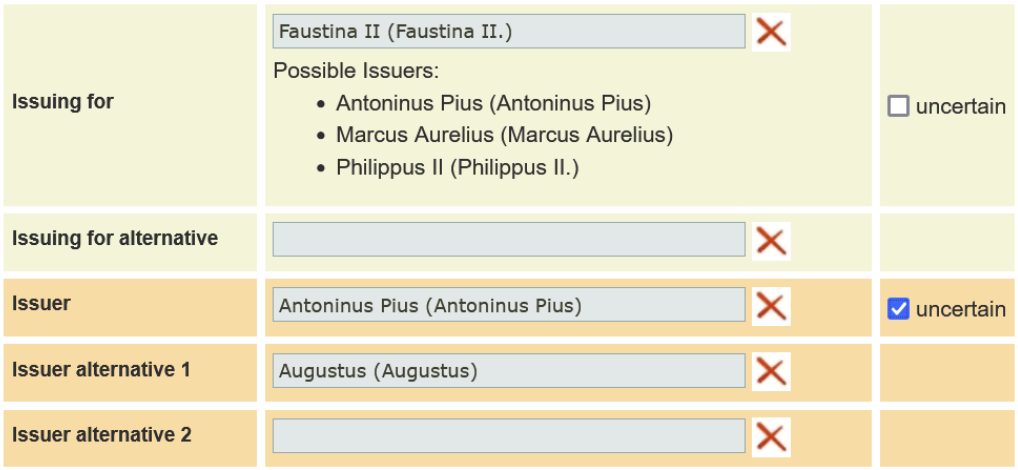


Figure 24: Uncertainty representation in AFE.

*Issuing for* and *Issuer* allows up to five states:

- Selecting one while not selecting the uncertain checkbox: The user has high confidence that this is the true issuer or depicted person.
- Selecting one while selecting the uncertain checkbox: The user has moderate confidence that this is the true issuer or depicted person but it could be someone else.
- Selecting alternatives while not selecting the uncertain checkbox: The user has high confidence that the true issuer or depicted person is part of the alternatives.
- Selecting alternatives while selecting the uncertain checkbox: The user has moderate confidence that the true issuer or depicted person is part of the alternatives but it could be someone else.
- Nothing is selected: The user is unable to identify the issuer or depicted person (sometimes there is no depicted person) or to significantly reduce the possibilities.

The first step of this use case is to translate the relational database model of AFE into an RDF graph. You need to represent the issuer alternatives in a way that enables the RDF reasoner to work with the data. You could, for example, add as many `ex:issuer` and `ex:issuing_for` triples as there are alternatives for the respective field. Additionally, you need to add a triple like `ex:coin1 ex:issuer ex:uncertain` for coins where the uncertain checkbox for issuer (and depicted person respectively) is selected.

Then the reasoner will assign certainty values to the edges from the AFE data in the graph. The certainty value is evenly distributed among the alternatives and checking the uncertain checkbox decreases the certainty value by a user-defined amount like 0.2. This decreased amount will be assigned to the statements containing the uncertainty object from the uncertain checkbox. This means the coin from Figure 24 has two issuer alternative triples that get a certainty value of 0.4 each and one issuer triple to the uncertainty object with a certainty of 0.2. The depicted person triple to Faustina II gets the certainty value 1.0 because there are no alternatives selected and the uncertain checkbox is also unchecked.

## 3.4. Use Case 3: Classification Values from Multiple Models

Due to corrosion, it can be difficult to identify the coin type of a coin as you can see in Figure 25. Therefore, you may get conflicting information from different experts or machine learning models. There are also coin types that are notoriously hard to classify even if the coin is well-preserved.

For example, there is a coin to be classified by two models. One model classifies the coin as

Figure 25: Corroded coin. From [41].

either ex:coinType1 or ex:coinType2 while the other model classifies it as ex:coinType3. The models in this example could also be real persons like researchers. One expert focuses on the hair of the depicted person while the other makes the selection mostly based on the legend (the text) on the coin leading to conflicting selections. Contrary to the previous use case from subsection 3.2, the coin type selection uses an uncertain predicate and not a vague one. The coin is only one of the selected coin types (regarding one coin type catalogue) and therefore the statements used are assigned a certainty. The task of this use case is to aggregate the certainty values from multiple models or experts. Since this use case has uncertain statements, the cer-

tainty values outgoing statements add up to one, enabling the usage of the Dempster-Shafer Theory from subsection 2.2.3.

In subsection 2.2.3, the authors use the Dempster-Shafer Theory to solve conflicts among

| Coin Type | Model 1 | Model 2 | Weighted Mean |
|---|---|---|---|
| {ex:coinType1} | 0.5 | 0 | 0.2 |
| {ex:coinType2} | 0.5 | 0 | 0.2 |
| {ex:coinType3} | 0 | 1 | 0.6 |

Table 4: Solving use case 3 using the weighted mean as the aggregation function. Model 1 has a weight of 0.4 and Model 2 0.6.

sensors. This use case can be solved by using a simple aggregation like in subsection 3.2 or utilizing the Dempster-Shafer Theory to combine the evidence. You also only need a simple version of the Dempster-Shafer Theory with subsets containing one element and one subset containing the whole set of discernment as you can see in Table 5. The mass value of the set containing all values (the *ignorance*) could be determined by model performance metrics like the F-score or a fixed value like 0.2. A model with an F-score of 0.9 would get an ignorance value of 0.1. The ignorance of a model or experts determines the amount of trust you have in their selection. More ignorance means less trust and vice versa. It also prevents having disjoint subsets and thus no real solution for the joint mass function like in Table 2.

For example, there are two models that classify a coin again. Model 1 showed a lower perfor-

| Subset | Model 1 | Model 2 | Joint Mass | Plausibility |
|---|---|---|---|---|
| {ex:coinType1} | 0.3 | 0 | 0.065 | 0.152 |
| {ex:coinType2} | 0.3 | 0 | 0.065 | 0.152 |
| {ex:coinType3} | 0 | 0.9 | 0.783 | 0.87 |
| ignorance | 0.4 | 0.1 | 0.087 | 1.0 |

Table 5: Example for using Dempster-Shafer Theory instead of aggregating using the weighted mean.

mance than model 2 during tests and is therefore assigned a lower confidence than model 2. The F-score of model 1 was 0.6 and the score of model 2 was 0.9. In Table 4 you can see an example that is resolved using the weighted mean of those selections. ex:coinType1, ex:coinType2 and ex:coinType3 are the possible coin types. The selection of model 1 is weighted with $\frac{0.6}{0.6+0.9} = 0.4$ and the selection of model 2 is weighted with $\frac{0.9}{0.6+0.9} = 0.6$. However, using the weighted mean simplifies the problem. In Table 5 the example is solved using the Dempster-Shafer Theory. The ignorance of each model is $1-f(m)$ with $f(m)$ being the F-score of the model $m$. The ignorance represents the set of all possible coin types (set of discernment) {ex:coinType1, ex:coinType2,

ex:coinType3} and thus the distrust in the selection. The weighted mean ignores the evidence for coin type ex:coinType3 from Model 1 which is included in the ignorance from Model 1. Additionally, the Dempster-Shafer Theory also outputs an ignorance value which can be interpreted as an amount of distrust in this aggregation of evidence. Accordingly, this thesis will solve this use case using the Dempster-Shafer Theory.

## 3.5. Use Case 4: Adding Supporting Evidence to AFE Relationships

In Figure 24, you can also see that AFE suggests issuers to the users based on the depicted person on the coin (issuing for). The task of this use case is to also process supporting evidence after assigning the issuer and depicted person triples with certainty values. The supporting evidence in this use case stems from domain knowledge and you can see it in Figure 24 as "Possible issuers:". The list is created by domain experts based on current knowledge. Those issuers are known to have minted coins with that depicted person but AFE also allows the selection of other issuers that are not in the list.

You can use the Dempster-Shafer Theory to combine the certainty values of the current issuer

| Subset | User Mass Func | Domain Knowledge Mass Func | Joint Mas |
|---|---|---|---|
| {Antoninus Pius} | 0.3 | 0.267 | 0.411 |
| {Marcus Aurelius} | 0 | 0.267 | 0.178 |
| {Philippus II} | 0 | 0.267 | 0.178 |
| {Augustus} | 0.3 | 0 | 0.100 |
| ignorance | 0.4 | 0.2 | 0.133 |

Table 6: AFE example with supporting evidence from domain knowledge. Antoninus Pius and Augustus are selected by the user and the uncertain checkbox is selected. The ignorance is the whole set of discernment.

selection with the domain knowledge. You start by translating the issuer triples into a mass function. Then each depicted person alternative has a selection of domain knowledge issuers that are also translated into mass functions. You can now combine all the mass functions using the equation 1.

In Table 6, you can see the example from Figure 24 where the user classified a coin and selected `Antoninus Pius`, `Augustus`, and the uncertain checkbox but the domain knowledge suggests `Antoninus Pius`, `Marcus Aurelius`, or `Philippus II`. Every mass function has a default ignorance of 0.2 represented by the whole set of discernment. The uncertain checkbox increases the ignorance of the user mass function from 0.2 to 0.4. Therefore the rest 0.6 is split evenly among the issuers that the user selected. The domain knowledge mass function

uses the default ignorance value and also splits the rest evenly among its elements. Using the Dempster-Shafer Theory you can now combine the mass function and use the joint mass values as certainty values for the resulting triples from coin to issuer. However, this is just one of five possible cases.

The first case is that no depicted person is selected. This means the coin is skipped during processing because no supporting evidence can be added. In the second case, only one depicted person is selected like in Table 6. The next one is one depicted person but the uncertain checkbox for depicted person is now selected. You can handle this by increasing the ignorance of the domain knowledge mass functions from 0.2 to 0.4. The question is how to deal with multiple depicted person alternatives. In the end, selecting multiple depicted person alternatives is just adding more evidence and thus mass function to the use case. That means instead of one domain knowledge mass function there are now as many as there are depicted person alternatives selected (the web interface only allows up to two). In Table 7, you can see the

| Subset | User Mass Func | Domain Knowledge Mass Func 1 | Domain Knowledge Mass Func 2 | Joint Mass |
|---|---|---|---|---|
| {Antoninus Pius} | 0.3 | 0.267 | 0 | 0.237 |
| {Marcus Aurelius} | 0 | 0.267 | 0 | 0.103 |
| {Philippus II} | 0 | 0.267 | 0 | 0.103 |
| {Augustus} | 0.3 | 0 | 0.4 | 0.327 |
| {Tiberius} | 0 | 0 | 0.4 | 0.154 |
| ignorance | 0.4 | 0.2 | 0.2 | 0.077 |

Table 7: AFE example from Table 6 but with "Tiberius" as a depicted person alternative.

example from Table 6 but with "Tiberius" selected as a depicted person alternative. The domain knowledge list suggests the issuers Augustus and Tiberius for Tiberius as the depicted person. The last case is multiple depicted person alternatives and the uncertain checkbox for the depicted person is selected. This would increase the ignorance of both domain knowledge mass functions from Table 7 0.2 to 0.4.

## 3.6. Use Case 5: Working with Similar Coins

In this use case, you have coins with coin types, that are assigned with coin types using the `ex:isCoinType` predicate, and coin types that have similarity triples with other coin types using the `ex:similarTo` predicate. Both relationships can have multiple certainty values (`ex:isCoinType`) and precision degrees (`ex:similarTo`) from different models. Your first step

is to aggregate the precision degrees from the similarity triples like in subsection 3.2. Then you use the Dempster-Shafer Theory from subsection 3.4 to aggregate the certainty values from the coin type selection. The last step is to use the chain rule reasoning from subsection 2.2.4 to infer new triples from the graph.

One chain rule could state if `A ex:isCoinType B` and `C ex:isCoinType B` then `A ex:similarCoinTypeAs C`. This sounds like an obvious rule but since everything is assigned a certainty, it can be helpful to calculate the certainty for this consequent triple. With uncertainty that means if ex:coin1 is ex:coinType1 with a certainty of 0.9 and ex:coin2 is ex:coinType1 with a certainty of 0.7 then ex:coin1 and ex:coin2 are of the same coin type with a certainty of $0.9 \cdot 0.7 = 0.63$ (using the product rule). Additionally, you could state if `A ex:isCoinType B` and `B ex:similarTo C` then `A ex:similarTo C`. For example, if ex:coin1 is ex:coinType1 with a precision degree of 0.9 and ex:coinType1 is similar to ex:coinType2 with a precision degree of 0.8 then ex:coin1 is similar to ex:coinType2 with a precision degree of $0.9 \cdot 0.8 = 0.71$. This example had only one viable path from ex:coin1 to ex:coinType2 during reasoning you need to sum the precision degrees of all viable paths like it is shown in Figure 21. The last chain rule states if `A ex:similarTo B` and `B ex:similarTo C` then `A ex:similarTo C`. In this case, you would need to use the Lukasiewicz logic as discussed in subsection 3.1.

To achieve this you can use AMT chain rule reasoning. In this use case are the classes *ex:Coin* and *ex:CoinType*, and the predicates *ex:similarTo*, *ex:isCoinTypeOf*, `ex:similarCoinTypeAs`, and *ex:isCoinType*. The predicate *ex:similarTo* is a vague predicate because coins look similar to each other to certain degree. The rest of the predicates are uncertain because they are either true or false. This needs to be considered while defining the following axioms:

1. RoleChainAxiom: ex:similarTo ex:similarTo ex:similarTo (LukasiewiczLogic)
2. RoleChainAxiom: ex:isCoinType ex:similarTo ex:similarTo (ProductLogic, Summarize)
3. RoleChainAxiom: ex:isCoinType ex:isCoinTypeOf ex:similarCoinTypeAs (ProductLogic)
4. DisjointAxiom: ex:isCoinType ex:similarTo (Remove ex:similarTo)
5. DisjointAxiom: ex:isCoinTypeOf ex:similarTo (Remove ex:similarTo)
6. SelfDisjointAxiom: ex:similarTo (Remove ex:similarTo)
7. InverseAxiom: ex:similarTo ex:similarTo
8. InverseAxiom: ex:similarCoinTypeAs ex:similarCoinTypeAs
9. InverseAxiom: ex:isCoinType ex:isCoinTypeOf

10. InverseAxiom: ex:isCoinTypeOf ex:isCoinType

Axiom 1 defines the property `similarTo` as a transitive statement. Since the similarity highly depends on the context, you need to use the Lukasiewicz logic (see subsection 3.1). Axiom 2 states that coins are similar to the coin types that are similar to their coin type. In this case, you combine a precision degree with a certainty and thus need to summarize all the viable paths instead of just choosing the best path like the AMT. Axiom 3 represents coins of the same coin type being of the same coin type. In this axiom, you combine two certainty values to get a probability of two coins actually being of the same coin type. Axiom 4 is a constraint that removes unwanted triples as in Figure 22. Contrary to the AMT, this constraint axiom should not throw an error but remove the affected `ex:similarTo` triples. Axioms 5 to 8 define the inverse relationships.

## 3.7. Use Case 6: Most Probable Minting Interval for Coins

In this use case, there are multiple coins in an RDF graph that have one or more possible coin types determined by multiple experts or models. Additionally, each coin type has a continuous time interval which represents the time this coin type was minted. The task is to assign the coins with possible minting intervals derived from the time intervals of their coin types. The resulting minting intervals should be assigned probabilities which then allow you to find the most probable minting intervals for the coins. In my research project [45], I suggested the Dempster-Shafer Theory to solve a similar use case about coin hoards which were assigned time intervals.

Each coin type represents a mass function with their intervals being the subsets. For example, the coin `ex:coin1` is either coin type `ex:coinType1`, `ex:coinType2`, or `ex:coinType3`. `ex:coinType1` was minted 72-73 BC, `ex:coinType2` was minted 70-85 BC, and `ex:coinType3` was minted 73-100 BC. Those time intervals can be represented by the subsets {-73, -72}, {-85, ..., -70}, and {-100, ..., -73} with the minus sign indicating the year being BC. Additionally, you can add a default ignorance of 0.2 again.

In Table 8, you can see an example of a coin with three coin types which are all equally possible. You can apply the equations from subsection 2.2.3 again to calculate the joint mass, belief, and plausibility values. Contrary to use case 3.4, the lower bound for the probability (the belief) is generally not equal to the mass value for this use case. This results in large possible probability intervals like between 0.68 and 1.0 for the subset (-73, -73). It means that the result of the Dempster-Shafer Theory does not provide you with possible minting intervals assigned a probability but only a probability interval. Therefore, the Dempster-Shafer theory may not be the correct choice to handle these time intervals. Future research could test different

| Subset | ex:coin-Type1 Mass | ex:coin-Type2 Mass | ex:coin-Type3 Mass | Joint Mass | Belief | Plausibility |
|---|---|---|---|---|---|---|
| (-73, -72) | 0.8 | 0 | 0 | 0.16 | 0.8 | 1.0 |
| (-85, -70) | 0 | 0.8 | 0 | 0.032 | 0.96 | 1.0 |
| (-100, -73) | 0 | 0 | 0.8 | 0.032 | 0.8 | 1.0 |
| (-73, -73) | 0 | 0 | 0 | 0.64 | 0.64 | 1.0 |
| (-85, -73) | 0 | 0 | 0 | 0.128 | 0.768 | 1.0 |
| ignorance | 0.2 | 0.2 | 0.2 | 0.008 | 1.0 | 1.0 |

Table 8: Dempster-Shafer Theory example for combining time intervals for a coin. Each subset represents the mint interval of a coin's possible coin type. (a, b) represents a set containing all the natural numbers of the continuous interval from a to b (both boundaries are inclusive).

approaches to solve this use case, but the approaches in this thesis are not applicable.

# 4. Reasoner Implementation

This section is about the implementation of the reasoner. Subsection 4.1 is about the internal format for the RDF graph representation. Subsection 4.2 then describes the SPARQL connector classes used in the reasoner. The actual reasoning process is illustrated in subsection 4.3. The code for this thesis can be found on Github[16].

## 4.1. RDF Graph and Internal Representation

In this section, we will define the graph formats that the reasoner will support by default and how the reasoner represents the graph at runtime. To simplify the internal format and reasoning process, both the certainty and the precision degree are referred to as the *weight* of the triple. Therefore, the reasoner will not enforce any constraints regarding the correctness of axioms such as checking for the right reasoning logic when combining vagueness and uncertainty in a chain rule. It is up to the user to define rules that make sense for reasoning.

### 4.1.1. Supported Graph Formats

The default SPARQL queries for the RDF graph will only work on a specific format to model weights and load the whole graph. However, you will be able to modify the select queries to limit the amount of triples that are used during reasoning. Additionally, you will also be able to quickly write your own SPARQL connectors for your own graph formats (see subsection 4.2).

By default the reasoner of this thesis supports the reification approach from Figure 9 on the left and the RDF-star representation from Figure 9 on the right. The reification approach is widely supported but rather slow because you need at least 4 triples to model uncertainty or weights (see subsection 2.1.6). The RDF-star approach is not yet part of the W3C recommendations and therefore not always supported but showed significant query runtime improvements compared to other uncertainty modelling approaches [30].

The predicate assigning a model or expert to the weight value are treated as optional. Furthermore, both approaches will also support certain and precise triples that have no weight attached to them and thus get weight value of 1 with no model assigned to them. For example, if the graph contains the triple `ex:coin1 ex:isCoinType ex:coinType1` without further weight assignment it would be interpreted as a certain triple with a weight of 1.0 by the reasoner.

[16] `https://github.com/jensw99/uncertainty-reasoner` (last accessed 10.07.2024)

### 4.1.2. Internal Representation

Since the reasoner is written in Python, it would make sense to use the Python package rdflib[17] as the backend to store the triples. However, rdflib has no official support for RDF-star yet. Therefore, for the sake of flexibility like supporting a wide variety of graph formats and reducing overhead stemming from the graph representation the reasoner will use a graph representation as a pandas[18] dataframe. Pandas is partially written in C and Cython[19], and offers a wide variety of vectorized functions that can significantly speed up you code depending on the use case [27]. Future research could further analyze the code and improve the performance.

In the tables 9 and 10, you can see the internal representation of the graph as dataframes. The

| Column | Data Type | Description |
|---|---|---|
| s | string | Subject of the triple |
| p | string | Predicate of the triple |
| o | string | Object of the triple |
| weight | float64 | Certainty or precision degree from 0 to 1. Triples without a specified weight get the default value 1. |
| model | string | Name of the model or expert this weight value stems from (optional) |

Table 9: Internal Pandas dataframe representation of the RDF graph without the class relationships. Will be referred to as *df_ triples*.

| Column | Data Type | Description |
|---|---|---|
| node | string | The node that is referenced |
| class | string | The class of the node |

Table 10: Internal Pandas dataframe representation of the RDF graph class relationships. Will be referred to as *df_ classes*.

dataframe *df_ triples* contains everything except class declarations and the dataframe *df_ classes* only the class declarations. The class declaration are in a separate dataframe to use them during reasoning without extracting them from df_triples multiple times. In this reasoner implementation, the class declarations are treated as certain/precise and therefore df_classes

[17] `https://rdflib.readthedocs.io/en/stable/` (last accessed 10.07.2024)
[18] `https://pandas.pydata.org/` (last accessed 10.07.2024)
[19] `https://cython.org/` (last accessed 10.07.2024)

has no weight column. Each row in df_triples contains a triple *(s, p, o)* with a weight value *weight*. Some of the weight values may also have a model or expert name *model* attached to it.

## 4.2. SPARQL Connector

To communicate with a SPARQL endpoint, the reasoner uses one of two possible SPARQL connectors: The *ReificationSparqlConnector* or the *SparqlStarConnector*. The type of connector determines how the weights are modeled in the SPARQL graph and acts as a layer to translate the data from the graph into the internal dataframe representation and the other way around. You can see the UML diagram for the connectors in Figure 26. Both connectors inherit from the abstract *SparqlBaseConnector* class which defines the functions that are needed for the reasoner.

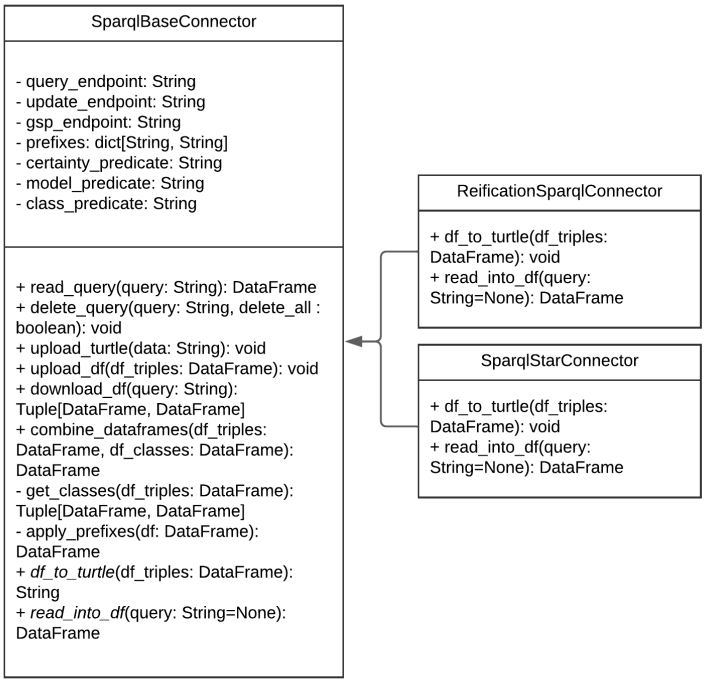


Figure 26: UML diagram of the SPARQL connector classes.

### 4.2.1. SparqlBaseConnector

The SparqlBaseConnector acts like an abstract class from Java. It defines the functions that the reasoner expects from a SPARQL connector and some helper functions that all connectors need regardless of representation in the graph. The users should not use the SparqlBaseConnector

but the classes that inherit from it like the ReificationSparqlConnector and the SparqlStarConnector. To declare the SparqlBaseConnector as abstract, it inherits from the *ABC (abstract base class)* class and the abstract methods get the decorator *@abstractmethod*. Defining SparqlBaseConnector as an abstract class makes it easier to implement new SPARQL connectors which now only need to implement the abstract methods.

The communication with the graph store endpoint is implemented in SparqlBaseConnector using HTTP requests. This communication protocol[20] was is defined by the W3C. Therefore, the base connector uses the *requests* python package for the communication with the graph store. The constructor expects three different endpoints. The query endpoint is for the select queries, the update endpoint is for altering data inside the graph like deleting triples and the *Graph Store Protocol*[21] endpoint is for bulk uploading turtle files. The Graph Store HTTP Protocol is an alternative to the standard SPARQL update protocol and you can use it to conveniently upload whole turtle files without writing a insert query. Those endpoints may depend on the type of graph store you use. In Figure 27, you can see the endpoints for an example dataset *test* in Fuseki 4.10.0.

The non-abstract functions in SparqlBaseConnector enable querying, deleting and uploading data in the graph store. The classes that inherit from SparqlBaseConnector are only for translating the queried data into the internal dataframe format and from the dataframe format back into turtle triples. You can also define additional prefixes for your namespaces and the predicates to use for weight values, model names and class assignments. The standard predicates are `ex:weight` for the weight value, `ex:accordingTo` to assign a weight value to model or expert and `rdf:type` to define the class (also called concept in subsection 2.2.4) of a node.

There are only two abstract methods to be implemented by classes that inherit from SparqlBaseConnector: *df_to_turtle* and *read_into_df*. The function *df_to_turtle* translates a given dataframe in the internal format from table 9 to turtle triples. *read_into_df* does the opposite by querying the SPARQL endpoint and translating the result into the internal dataframe format. The easy way to parse the SPARQL results would be to use csv as the accepted return format by setting the header *Accept* to *text/csv*[22]. However, the csv format lacks important information like the node type (URI node, blank node etc.) and the datatypes of literal nodes. For example, the blank nodes are just named like "bx" with x being a number. It would be difficult to identify those blank nodes between the normal URI nodes. Additionally, the literals are provided without hints that these nodes are literals and the data type is missing completely. Therefore, you can use *application/json*[23] to get the SPARQL results in the JSON format. The

[20] `https://www.w3.org/TR/2013/REC-sparql11-protocol-20130321/` (last accessed 10.07.2024)
[21] `https://www.w3.org/TR/2013/REC-sparql11-http-rdf-update-20130321/` (last accessed 10.07.2024)
[22] `https://www.w3.org/TR/sparql11-results-csv-tsv/` (last accessed 10.07.2024)
[23] `https://www.w3.org/TR/sparql11-results-json/` (last accessed 10.07.2024)

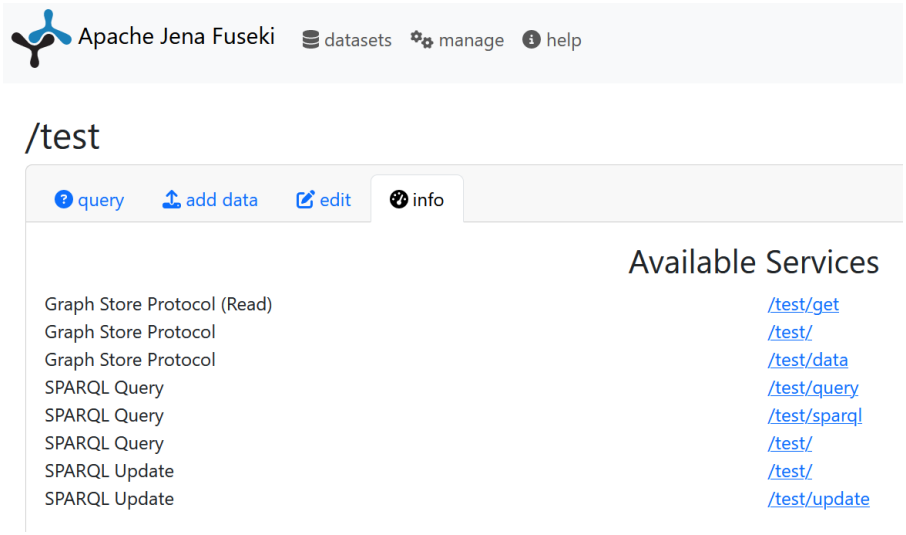


Figure 27: Endpoints for the dataset *test* in Fuseki 4.10.0.

JSON format includes all the necessary information and can be parsed by pandas using the *pandas.json_normalize()* function. You can now identify blank nodes and rename them from *bx* to *_:bx* so they can easily translated to Turtle triples again. Then you can also put the values of literals between quotation marks and attach their data type. For example, an integer with the value 73 would now be stored as *"73"8sd:integer* instead of just 73. By default this function selects the whole graph but it also allows the use of user-defined queries to further adjust the select queries. This allows the selection of specific triples and not only the whole graph. It also means that the user is responsible to get the result in the expected format when altering the queries.

The function *upload_df* from the SparqlBaseConnector utilizes *df_to_turtle* to translate the dataframe into Turtle triples and then uploads the Turtle triples to the SPARQL Graph Store Protocol endpoint via *upload_turtle*. The functions *add_classes*, *apply_prefixes* and *read_into_df* are used in *download_df* to download the triples and separate them into df_triples and df_classes. Since querying the data with prefixes does not abbreviate the resources in the result, *apply_prefixes* is used to abbreviate the resources to their qualified names in the query result. This helps during reasoning because you do not need to use lengthy URIs. *add_classes* extracts the triples that define the class declaration of nodes by searching for triples with the user-defined *class_predicate* in the given dataframe. Those class triples are then used to create the *df_classes* dataframe. Before saving the result to a file or endpoint, the reasoner can use *combine_dataframes* to combine df_triples and df_classes to a triples dataframe that includes the class declarations.

There are 3 types of triples that need to be considered when implementing a class inheriting

from SparqlBaseConnector: certain/precise triples, triples with a weight and triples with weight assigned with a model. For example:

1. `ex:coin1 ex:isCoinType ex:coinType1`
2. `ex:coin2 ex:isCoinType ex:coinType1` with a weight of 0.8
3. `ex:coinType1 ex:similarTo ex:coinType2` with a weight of 0.7 according to ex:model1

### 4.2.2. ReificationSparqlConnector

The ReificationSparqlConnector inherits from the SparqlBaseConnector and implements the functions *df_to_turtle* and *read_into_df* using the reification approach.

The function *read_into_df* only needs a default query to select all triples in the relevant

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX ex: <http://example.org/>

SELECT ?s ?p ?o ?weight ?model WHERE {
    {?b rdf:subject ?s .
    ?b rdf:predicate ?p .
    ?b rdf:object ?o .
    ?b ex:weight ?weight .
    OPTIONAL {?b ex:accordingTo ?model} .} UNION {
        ?s ?p ?o .
        FILTER NOT EXISTS{?s ex:weight ?certainty .}
    }
}
```

Figure 28: Default select SPARQL query for the ReificationSparqlConnector.

format. You can see the default query in Figure 28. The first intuition would be to use a simple graph pattern with blank node identifiers like in Figure 4 but using those do not work with the OPTIONAL keyword. OPTIONAL is needed to select the optional model assignments for certainty values like in example 3 from subsection 4.2.1, but it defines a new simple graph pattern and you cannot use the same blank node identifiers in two different graph patterns (SPARQL constraint). Therefore, the default query uses the normal variable identifier *?b* to address the blank nodes. The UNION keyword is used to also select certain triples that are not in the reification format like example triple 1. To select only certain/precise triples, it is not sufficient to check if the subject is not a blank node because it would disallow the use of blank

nodes except for assigning a weight to a triple. You need to check if the subject is a reification blank node by filtering for subjects without a `?s ex:certaintyValue ?certainty` triple.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix ex: <http://example.org/> .

# Example 1
ex:coin1 ex:isCoinType ex:coinType1

# Example 2
_:1 rdf:subject ex:coin2 .
_:1 rdf:predicate ex:isCoinType .
_:1 rdf:object ex:coinType1 .
_:1 ex:weight "0.8"^^xsd:decimal .

# Example 3
_:2 rdf:subject ex:coinType1 .
_:2 rdf:predicate ex:similarTo .
_:2 rdf:object ex:coinType2 .
_:2 ex:weight "0.7"^^xsd:decimal .
_:2 ex:accordingTo "ex:model1" .
```

Figure 29: Example triple translated to Turtle via ReificationSparqlConnector.

The naive solution for *df_to_turtle* would be to iterate the dataframe and create triples for each row but this is a rather slow approach. The strength of pandas are the vectorized functions and thus this is how to create the turtle triples. You first separate certain/precise triples from uncertain triples. Then you create a column for each turtle triple predicate by concatenating strings with information in the rows. Certain/precise triples need only one turtle triple and thus one column. Statements with a weight value need four or five triples and thus four or five columns (`rdf:subject`, `rdf:object`, ...) depending on having a model name attached or not. You can see the translated example triples in Figure 29. In the end, you concat the values of all columns to one big turtle string using *Series.str.cat()*.

### 4.2.3. SparqlStarConnector

The SparqlStarConnector implements the SPARQL-star approach. The default select query for the SparqlStarConnector in Figure 30 first selects the triples with a certainty value and a model like example 3. Then the triples which only have a certainty value like example 2. However, getting the certain/precise triples needs a bit more work since there no easy way to negate a filter. For example, to negate `FILTER(isTRIPLE(?s))` you cannot just write

`FILTER(!isTRIPLE(?s))`. One way to negate a filter is using `MINUS`. You select all triples and remove all triples where the subject is a triple itself, leaving the certain triples. That also means that the connector only allows the use of RDF-star triples for modeling the weight values.

The function *df_to_turtle* is similar to the one from the ReificationSparqlConnector, but

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX ex: <http://example.org/>

SELECT ?s ?p ?o ?weight ?model WHERE {
    {
        << << ?s ?p ?o >> ex:weight ?weight >> ex:accordingTo ?model .
 } UNION {
        << ?s ?p ?o >> ex:weight ?weight
    } UNION {
        ?s ?p ?o .
        MINUS {
            ?s ?p ?o .
            FILTER(isTRIPLE(?s)) .
        }
    }
}
```

Figure 30: Default select SPARQL query for the SparqlStarConnector.

instead of having up to five turtle triples per internal dataframe row, the SparqlStarConnector only needs up to three (the triple itself, the certainty value, and the model of the certainty value). In Figure 31, you can see the example triples translated using the SparqlStarConnector class. In the end, everything is concatenated using *Series.str.cat()* again.

## 4.3. Reasoner

The reasoner of this thesis will use axioms like in the AMT from subsection 2.2.4 for reasoning. However, this implementation will be a more general approach to support a wider variety of graphs. Instead of having roles and concepts like the AMT, you have the more generic *predicates* and *classes*. *Predicates* are just the predicates (also called properties) of the RDF triples and *classes* are the objects of class declarations with, for example, the predicate `rdf:type`. *Axioms* are the rules to infer new triples and are often for specific predicates or classes. The core idea is that the user can adjust the reasoner without many restrictions. This also means that it is up to the user to check whether the reasoning done using this reasoner does make sense.

In this implementation, the reasoning is split into three stages: The *Pre-Processing Stage*,

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix ex: <http://example.org/> .

# Example 1
ex:coin1 ex:isCoinType ex:coinType1

# Example 2
<< ex:coin2 ex:isCoinType ex:coinType1 >> ex:weight "0.8"^^xsd:decimal .

# Example 3
<< << ex:coinType1 ex:similarTo ex:coinType2 >> ex:weight "0.7"^^xsd:decimal >>
    ex:accordingTo "ex:model1" .
```

Figure 31: Example triple translated to Turtle via SparqlStarConnector.

the *Rule-Based Reasoning Stage*, and the *Post-Processing Stage* like you can see in Figure 32. The Pre-Processing Stage contains axioms that are only applied once before the rule based reasoning such as axioms that aggregate the similarity values from different models that are seen in the use case from subsection 3.2. The main objective of this stage is to eliminate multiple weights on triples from different models. Multiple weights for triples would not be dealt properly during the current chain rule reasoning based on the AMT. Therefore, at the end of this stage all triples with a model value are filtered out.

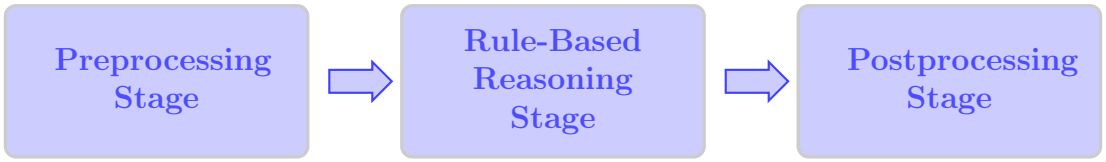


Figure 32: Reasoning stages.

The Rule-Based Reasoning Stage uses rules like the AMT chain rules to infer new triples or higher triple weights. This thesis will only implement the AMT chain rules but allows the creation of custom rules by implementing a abstract Axiom class. This stage applies the given axioms until the maximum number of iterations is reached or the weights of the triples did not change and no new triples were inferred during the iteration.

The last stage is the Post-Processing Stage which comes after the Rule-Based Reasoning Stage and applies each axiom once. This could be used in the future for post-processing like normalization or filtering for triples. However, this thesis will focus on the Pre-Processing and the Rule-Based Reasoning Stage.

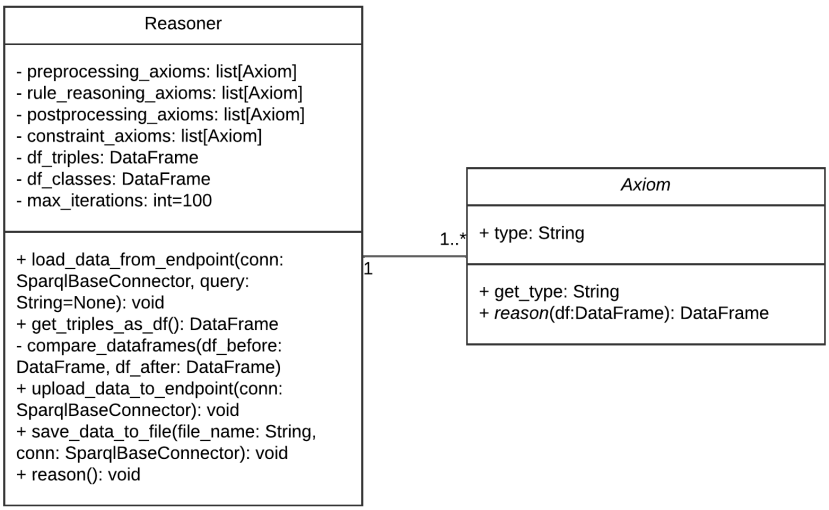


Figure 33: Part of the UML diagram for the reasoner.

In Figure 33, you can see the current reasoner implementation as a UML diagram. The *Reasoner* class coordinates the reasoning process and is initialized with a list of axioms that are split into the three stages. The function *load_data_from_endpoint* utilizes a *SparqlBaseConnector* class to load the triples and classes dataframe from the SPARQL endpoint and saves them into *df_triples* and *df_classes*. The reasoning is then triggered via the *reason* function. Each axiom in the Pre-Processing Stage is applied once to *df_triples* and then the axioms in the Rule-Based Reasoning stage are applied until it reaches the max iterations or until *df_classes* does not change after a iteration. In the end, all axioms in the Post-Processing Stage are applied and the function *compare_dataframes* compares a dataframe containing the triples before reasoning with the result of the reasoning to mark the triples that were inferred by the reasoner. The new inferred triples are marked with a user-defined reasoner name as their model. You can either upload the result to a SPARQL endpoint or save it to a Turtle file with *upload_data_to_endpoint* and *save_data_to_file*. Additionally, you can access the triples dataframe by using *get_triples_as_df*.

The *Axiom* class is a abstract class that defines the abstract method *reason* that processes *df_triples* and *df_classes*, and returns the transformed *df_triples*. When implementing this class, the stage of this axiom is set via the *stage* variable. The actual reasoning of this implementation is then done in the *reason* function of the implemented *Axiom* class. Those implementation can be more general axioms like the *ChainRuleAxiom* from subsection 4.3.2 or use case dependent like the *AFEDempsterShaferAxiom* from subsection 4.3.1.

### 4.3.1. Pre-Processing Axioms

There are currently four Axioms for the Pre-Processing Stage. There are generic axioms like the *AggregationAxiom*, the *CertaintyAssignmentAxiom* and the *DempsterShaferAxiom* but also the use-case-specific *AFEDempsterShaferAxiom*.

The *AggregationAxiom* implements the simple aggregation of weights from multiple models like in use case 3.2. The constructor of the AggregationAxiom expects a predicate and an aggregation type. Triples with the specified predicate are then grouped by subject, model and object, and their weights are aggregated using the specified aggregation type. Currently only mean and median are supported aggregation types for this axiom.

Sometimes, the triples may have not been assigned with a certainty value and thus a weight yet. Therefore, there is the *CertaintyAssignmentAxiom* to assign certainty weights to triples by using a simple heuristic. The constructor expects a predicate, a uncertainty object (default "ex:uncertain") and a uncertainty value $p_{unc}$ (default 0.2). The triples are then filtered for the predicate and grouped by subject, predicate and model to count the amount of objects (without the uncertainty objects) $n_s$ for each subject $s$ with this predicate. Additionally, it is also determined which of those subjects have a connection to the uncertainty object with the defined predicate. If a subject does not have a connection to the uncertainty object then each triple is assigned with the weight $\frac{1}{n_s}$. If there is a connection with the uncertainty object then the triple with the uncertainty object is assigned the uncertainty value $p_{unc}$ and the rest of the triples are assigned $\frac{1-p_{unc}}{n_s}$. Therefore, the uncertainty object can be used to indicate that the selected objects may not include the true object for this selection and thus their certainty weights should be reduced by a defined amount. In Figure 34, you can see an example for the predicate `ex:isCoinType` with two coins. `ex:coin2` has a connection with the uncertainty object and therefore the weights for the triples to `ex:coinType1` and ex:coinType2 are reduced by the default uncertainty value which is then assigned to the connection to the uncertainty object.

The *DempsterShaferAxiom* implements the Dempster-Shafer Theory approach to aggregate certainty values from multiple machine learning models or experts. It utilizes a simpler version of the Dempster-Shafer combination rule from Equation 1 that only supports subsets with one element or the whole set of discernment (all possible atomic states). You can find the Python code implementation in Figure 35. The combination algorithm compares the elements of two Python dictionaries which represent the mass function values for the subsets. The key of the subset is the single element of the subset or "*" and the value is the mass function value for this subset. The subset * denotes the *ignorance* of the mass function and thus contains the whole set of discernment. Increasing the ignorance of a mass function decreases the weight of the non-

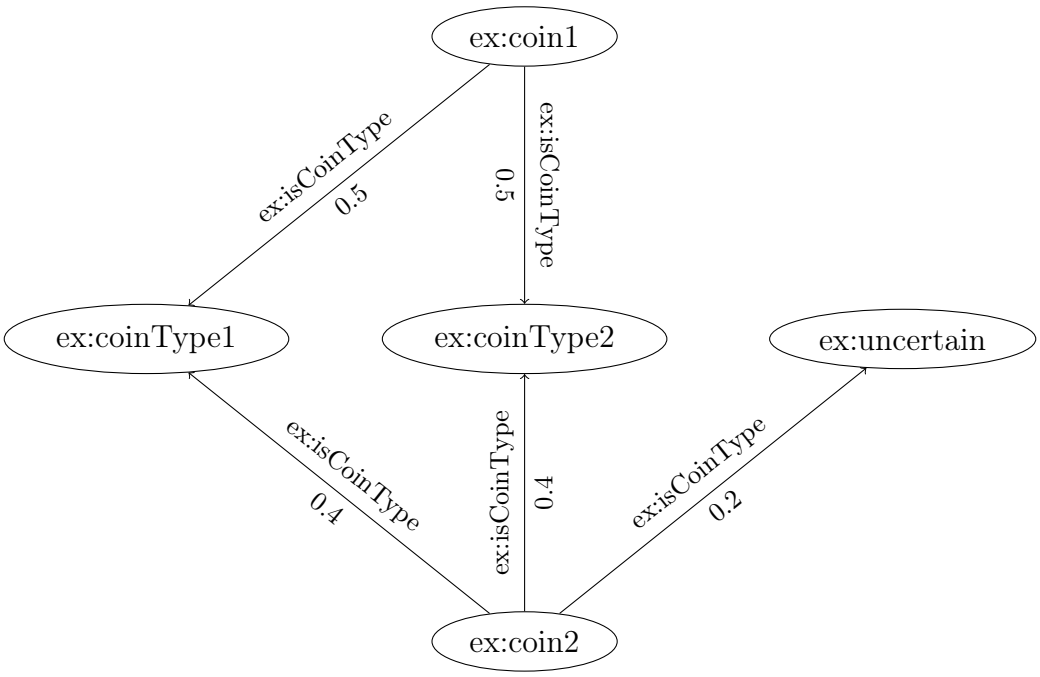


Figure 34: Example for the CertaintyAssignmentAxiom. ex:uncertain is the uncertainty object and the uncertainty value is 0.2.

ignorance subsets during the combination since all mass values add up to one. Additionally, it increases the joint mass value of all subsets A from the other mass function because the combination rule summarizes the products of the mass values of all subset combination from both mass functions where the intersection is A. In short $A \cap * = A$ for every subset A and therefore $m_1(A) \cdot m_2(*)$ is added to the numerator in Equation 1.

To combine two mass function dictionaries $m_1$ and $m_2$, the algorithm used in this thesis iterates both dictionaries and compares the keys. Let $k_1$ and $k_2$ be the currently selected keys for $m_1$ and $m_2$, and $m_3$ the resulting mass function dictionary. If both keys are equal then you increase the value of $m_3(k_1)$ by $m_1(k_1) \cdot m_2(k_2)$. If $k_1$ is * then $m_3(k_2)$ is increased by $m_1(k_1) \cdot m_2(k_2)$ and the other way around if $k_2$ is *. If both keys are unequal and both are not * then the empty set value $e$ is increased by $m_1(k_1) \cdot m_2(k_2)$. In the end you divide each value of $m_3$ by $1 - e$ to apply the denominator of Equation 1.

The constructor of the DempsterShaferAxiom expects a predicate, a ignorance object, a ignorance dictionary and a default ignorance $i$. It selects all the triples with the defined predicate and iterates all distinct subjects of those triples. For every subject it iterates the distinct model values that are present in the triples of the current subject. The function *df_to_subset_dict* is then used to translate the selected triples for this subject and model into a mass function dictionary. The mass function of * is the ignorance value from the ignorance dictionary or the default ignorance if the model name is not a key in the ignorance dictionary. If there is

also a connection to the uncertainty object in the selected triples then you add the weight of this connection to the * subset. All triples except those with the uncertainty object are then translated into dictionary key-value pairs and reduced by $\frac{i}{n}$ with n being the amount of triples without those with the uncertainty object. The dictionaries are then used to create *MassFunction* objects and are joint using the *join_masses* function. After all mass functions are joint the resulting mass function dictionary is iterated and used to fill the resulting dataframe.

However, the DempsterShaferAxiom would not work for the AFE use case from subsection 3.5

```
def join_masses(self, m: 'MassFunction') -> 'MassFunction':
    result = {}
    empty_set_value = 0
    mass_values1 = self.get_mass_values()
    mass_values2 = m.get_mass_values()
    for subset1 in mass_values1.keys():
        for subset2 in mass_values2.keys():
            if subset1 == subset2:
                result[subset1] = result.get(subset1, 0) + mass_values1[subset1] *
                    mass_values2[subset2]
            elif subset1 == '*':
                result[subset2] = result.get(subset2, 0) + mass_values1[subset1] *
                    mass_values2[subset2]
            elif subset2 == '*':
                result[subset1] = result.get(subset1, 0) + mass_values1[subset1] *
                    mass_values2[subset2]
            else:
                empty_set_value += mass_values1[subset1] * mass_values2[subset2]
    for subset in result.keys():
        result[subset] /= 1 - empty_set_value

    return MassFunction(result)
```

Figure 35: Mass function combination algorithm as Python code from the MassFunction class. *mass_values1* and *mass_values2* are both dictionaries.

because in this use case the mass functions are encoded in three different predicates. That is why there is the use-case-specific *AFEDempsterShaferAxiom* that implements the Dempster-Shafer Theory approach for this use case. The AFEDempsterShaferAxiom expects an issuer predicate, an issuing for predicate, a domain knowledge predicate, a uncertainty object and a default ignorance value. Instead of searching for one predicate the AFEDempsterShaferAxiom filters for three predicates (by default `ex:issuer`, `ex:issuing_for` and `ex:domain_knowledge`) and saves them into three different temporary dataframes. Firstly, all distinct subjects of the `ex:issuer` triples are iterated and all `ex:issuing_for` triples filtered for this subject. If this

`ex:issuer` subject has no `ex:issuing_for` triples then the weight values from `ex:issuer` are added to the result unchanged since there is no supporting evidence to add. Otherwise, the selected issuer triples are translated into a mass function. If there is a `ex:issuing_for` connection to the uncertainty object then the ignorance of the domain knowlege is increased by the weight of the triple. Then all the objects of the filtered `ex:issuing_for` triples are iterated and used to find the domain knowledge triples in the `ex:domain_knowledge`) dataframe. The domain knowledge triples are also translated into a mass function using the domain knowledge ignorance and joined with the issuer mass function. After all mass functions are joined the joint mass function is translated back to a dataframe and added to the result.

### 4.3.2. Chain Rule Reasoning Axioms

By default the reasoner supports four AMT axioms from subsection 2.2.4 for the Chain Rule Reasoning Stage: The *InverseAxiom*, the *ChainRuleAxiom*, the *DisjointAxiom*, and the *SelfDisjointAxiom*.

The *InverseAxiom* defines the inverse relationship from Figure 19. The constructor expects a predicate for the antecedent and the inverse relationships. During reasoning, the axiom filters for the antecedent predicate, swaps subject and object, and replaces the antecedent predicate of the result with the inverse predicate. Then it is concatenated to the existing triple dataframe and only the triples with the highest weight for each (s, p, o) triple are kept. You can, for example, define `ex:isParent` as the inverse of *ex:isChild* to infer the `ex:isParent` triple with the same weight as the *ex:isChild* counterpart. Sometimes the antecedent and inverse predicate can be the same like with `ex:similarTo` because the inverse of `A ex:similarTo B` is `B ex:similarTo A`.

The main reasoning of this stage is done in the *ChainRuleAxiom*. It implements a modified version of the AMT chain rule reasoning from Figure 18. Contrary to the AMT, this implementation of the ChainRuleAxiom allows to check the classes of the nodes of the antecedent relationships. In Figure 36, you can see the updated chain rule that also checks the classes of the nodes. However, defining classes A, B and C is optional, and not yet using any reasoning on the class hierarchy. This means, in an example ontology where `Cat` is a sublass of `Animal` defining class B as `Animal` would not select `Cat` classes even though there is a subclass relationship. This could be implemented in the future or done before using this reasoner. Another change to the original AMT chain rule is the option to summarize the path values. This allows the reasoner to mix vague and uncertain statements in the chain rule reasoning (see subsection 3.1). Additionally, the axiom allows to set input and output thresholds for the weights to reduce the workload during reasoning by ignoring low precision triples. The last parameter for this axiom

is the reasoning logic. Like the AMT this *ChainRuleAxiom* supports the Product Rule, Goedel Logic and Lukasiewicz Logic for the reasoning.

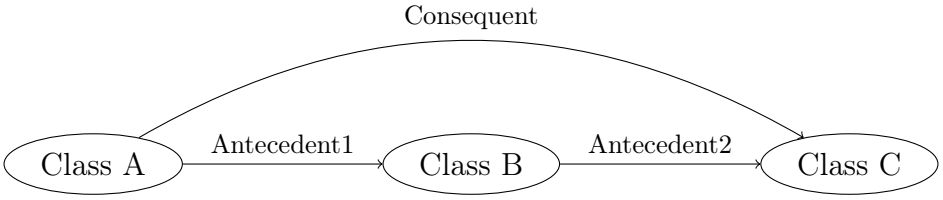


Figure 36: Updated Figure 18 with classes.

During reasoning the axiom searches for both antecedent triples in *df_ triples* and optionally applies the input threshold as a filter. It splits the selected triples into two dataframes: Left (Antecedent1) and right (Antecedent2). Those dataframes are then optionally filtered for the classes if those classes are selected. If, for example, class B is selected then the left dataframe filters the object for class B and right dataframe filters the subject for this class. This can be done by inner merging the corresponding dataframe with *df_ classes*. To get the consequent triples, you now merge the left and the right dataframe by object on the left and subject on the right. Then you can apply the selected reasoning logic and drop columns that are not needed any more. If the option to sum the viable paths is selected, then you group by subject, predicate and object, and summarize the weight. In the end, you just need to apply the optional output threshold and concat the result with the current *df_ triples* to then select the highest weight for each triple.

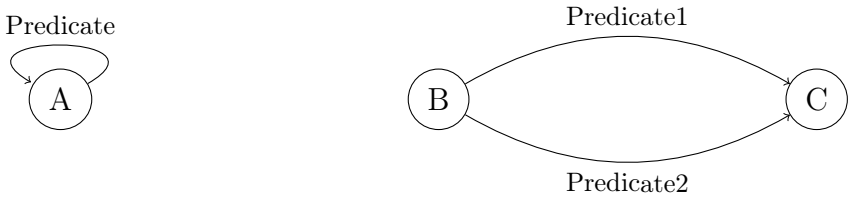


Figure 37: Example for the SelfDisjointAxiom (left) and the DisjointAxiom (right).

There are also two constraints that can either throw an exception or remove undesired triples. The *SelfDisjointAxiom* searches for triples that reference themselves like in Figure 37 on the left. The constructor expects a predicate and a boolean whether the axiom should throw an exception or remove the triples. It then searches for triples where subject and object are the same, and either removes them or throws a *ContraintException*. The *DisjointAxiom* works similar but searches for two triples with different predicates but equal subjects and objects like Figure 37 on the right. It expects two predicates *predicate1* and *predicate2*, a boolean whether the axiom should throw an exception or remove the triples, and a boolean whether it should

keep *predicate1* or *predicate2* during conflict. The axioms searches for two predicates that have the same subject and object, and either throws an error or removes one of the triples.

# 5. Solving the Use Cases and Benchmarks

In this section, the use cases defined in section 3 are solved using the reasoner introduced in section 4. Firstly, subsection 5.1 describes the general usage of the reasoner. Then the next subsections describe how to solve the use cases using the reasoner and test its performance. It will describe the capabilities of the reasoner but also its limitations. Each use case will be implemented using the reasoner and then tested using hand-crafted and random synthetic data. The hand-crafted data will be used to ensure that the reasoner works correctly while the synthetic data is utilized to analyze the runtime of the reasoning process. The *data retrieval runtime* is the time needed to fetch the data from the SPARQL endpoint while the *reasoning runtime* only includes the time of the actual reasoning. The *total runtime* is the sum of both the data retrieval runtime and the reasoning runtime. The runtime benchmarks measure the mean runtime of ten runs with the same configuration.

The tests were conducted on a computer with the following configuration:

- Operating System: Windows 11
- CPU: Intel Core i7 10700k
- RAM: 32 GB RAM
- Python: Version 3.12.0

This computer runs the benchmarks as well as the Apache Jena Fuseki 4.10.0 server that is used as a triple store. The Fuseki server is configured to use up to 12 GB of RAM. The most important Python packages used for the benchmarks can be found in Appendix B. The whole Anaconda[24] environment is also included in the Github repository.

## 5.1. Solving Use Cases with the Reasoner

The general structure for using the reasoner from section 4 can be seen in Figure 38 which implements the first use case. Using lines 4 to 11 you define the SPARQL communication with the triple store. In line 11, you could also add your own prefixes by using the parameter *prefixes*. The reasoning rules are defined in lines 13 to 15 by creating a list of axioms. These axioms are then used to create a reasoner object in line 17. The reasoner first fetches the graph using the provided connection in line 18 and then starts the actual reasoning in line 19. You can get the results by either retrieving it in the internal format (line 21), saving it to a turtle file (line 22), or uploading the result in a triple store (line 23). The code implementation for the remaining use cases can be found in Appendix C.

[24] `https://www.anaconda.com/` (last accessed 10.07.2024)

```python
import SparqlConnector
import Reasoner

# Query endpoint
QUERY_ENDPOINT = "http://localhost:3030/test/query"
# Update endpoint
UPDATE_ENDPOINT = "http://localhost:3030/test/update"
# Graph Store Protocol endpoint
GSP_ENDPOINT = "http://localhost:3030/test/data"

conn = SparqlConnector.SparqlStarConnector(QUERY_ENDPOINT, UPDATE_ENDPOINT,
    GSP_ENDPOINT)

axioms = [
     Reasoner.AggregationAxiom('ex:similarTo', 'mean')
]

reasoner = Reasoner.Reasoner(axioms)
reasoner.load_data_from_endpoint(conn)
reasoner.reason()

df_result = reasoner.get_triples_as_df()
reasoner.save_data_to_file('result.ttl', conn)
reasoner.upload_data_to_endpoint(conn)
```

Figure 38: Code for the first use case using the reasoner of this thesis, assuming the triples are already in the triple store.

## 5.2. Use Case 1

To solve the first use case from subsection 3.2, you only need one AggregationAxiom that aggregates the data and calculates the mean weight for each triple as you can see in Figure 38. The benchmark for this use case generates similarity triples between n coin types for m models that are then used for the reasoning. Each of the n coin types gets a similarity triple to the n-1 other coin types for each of the models. The triples are then assigned with random number between 0.0 and 1.0 using the numpy package and with their respective model. This means the amount of triple in the internal format is $3 \cdot (n \cdot (n-1))$, leading to a quadratic growth of triples when you increase the number of coin types.

In Figure 39, you can see the benchmark results for this use case using either the ReificationSparqlConnector or SparqlStarConnector. The choice of the SPARQL connector only affects the data retrieval runtime, because the SPARQL connector translated the given triples into the internal format, so the reasoning runtime, which does not include the data retrieval runtime,

stays the same. Parsing the endpoint results (Json to dataframe) is also unaffected by the choice of the SPARQL connector since the results of both queries are the same. Only the pure endpoint communication runtime is higher with the ReificationSparqlConnector which is expected with the results of [30]. Since the reasoning and data parsing runtime (Json to dataframe) is unaffected by the weight modelling approach, using both connectors for the benchmarks will not generate new insights. Therefore, the next benchmark tests will only use the SparqlStarConnector.

The biggest part of the total runtime for this use case is not the actual reasoning but the data retrieval runtime. This makes sense since you only aggregate the data and calculate the mean value which is easy to compute.

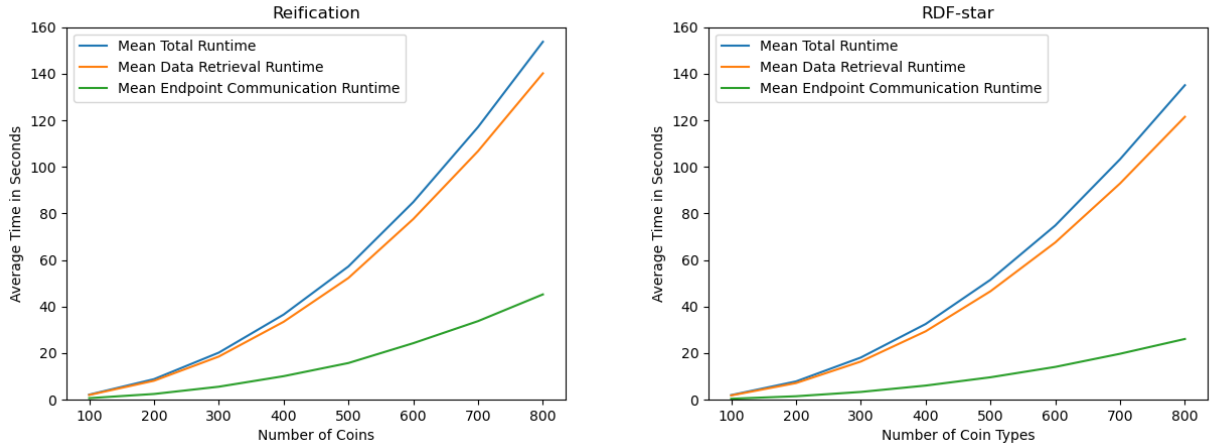


Figure 39: Runtimes for the first use case with synthetic data using the ReificationSparqlConnector (left) and the SparqlStarConnector (right). The data retrieval runtime includes the endpoint communication runtime and the parsing of the query results.

## 5.3. Use Case 2

Before solving the second use case from subsection 3.3, you need to translate the relational database schema of AFE into RDF triples. The tests only include the issuer, the depicted person (Issuing for), and the domain knowledge triples. Before reasoning, you need to export the database data into Excel or csv files, and transform them into the internal dataframe format. This dataframe can be uploaded to the SPARQL endpoint using either of the two SPARQL connectors.

After uploading the data, you can solve this use case using the CertaintyAssignmentAxiom on the issuer, depicted person, and domain knowledge predicates. To benchmark this use case, the DataGenerator class defines 5 possible issuers for each depicted person and selects one to three

of those to be the domain knowledge. Then each coin is assigned one to three depicted persons and one to three issuers from the possible issuer pool from the selected depicted persons. The issuer and the depicted person selection each have a chance of 20% to be marked as uncertain using the uncertainty object.

In Figure 40, you can the result for this benchmark. This time the total runtime is linear to the amount of coins, that is used in input data. The main part of the total runtime is the data retrieval runtime since the actual reasoning is written using pandas functions and therefore really fast.

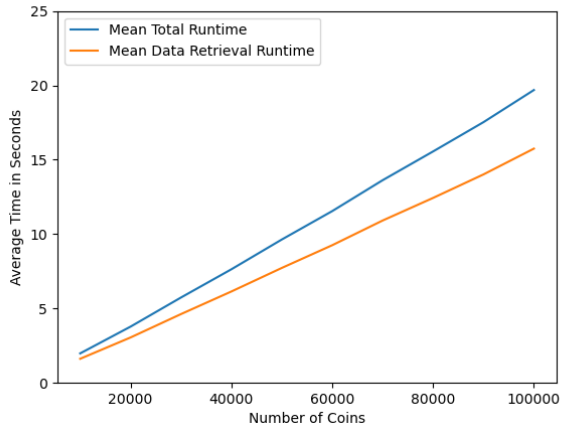


Figure 40: Benchmark results for the second use case using CertaintyAssignmentAxioms and synthetic data.

## 5.4. Use Case 3

The third use case from subsection 3.4 aggregates certainty values from multiple models using the Dempster-Shafer Theory. This can be done using the generic DempsterShaferAxiom. To benchmark, the *DataGenerator* class generates coin type assignments for $n_c$ coins that are assigned one to three of $n_{ct}$ possible coin types. Additionally, there is a 20% chance that the selection is marked as uncertain using the uncertainty object. This process is repeated for each of the $n_m$ models.

In Figure 41, you can see the results for $n_{ct} = 1000$ coin types and $n_m = 3$ models. In contrast to use case 1, the total runtime of this use case is not mainly the data retrieval runtime but rather the reasoning runtime. Also, the number of triples for this benchmark does not grow quadratic. Therefore, the reasoning runtime for the DempsterShaferAxiom is probably not linear, but also not quadratic regarding the curve. This reasoning runtime could be improved

in the future by implementing the Dempster-Shafer Theory using numpy[25] or pandas functions instead of pure Python like in the implementation of this thesis. If those packages do not provide the necessary functions then you could also try using Cython. Pure Python for-loops like in Figure 35 are a bottleneck for reasoning here.

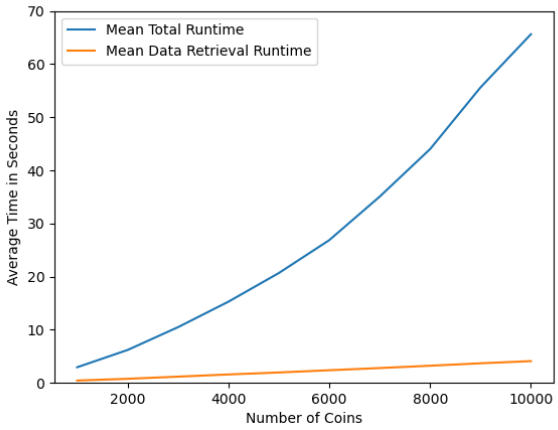


Figure 41: Benchmark results for the third use case using the DempsterShaferAxiom and synthetic data. There are 1000 possible coin types and 3 models.

## 5.5. Use Case 4

The fourth use case from subsection 3.5 extends the fourth use case by adding supporting evidence to the issuer selection for coins. This is done using the use-case-specific AFEDempsterShaferAxiom. The axiom works similarly to the DempsterShaferAxiom but the evidence for the issuer selection is encoded in the issuer, the depicted person (issuing for), and the domain knowledge triples.

The benchmark for this use case creates synthetic data like the fourth use case, applies the CertaintyAssignmentAxiom on the triples, and then aggregates them with the AFEDempsterShaferAxiom. In Figure 42 you can the mean runtime for different amounts of coins. The main reasoning runtime dominates the total runtime which is expected. The AFEDempsterShaferAxiom is a more complicated DempsterShaferAxiom and therefore even slower. Like the DempsterShaferAxiom, the performance of the AFEDempsterShaferAxiom could be improved in future research.

[25] `https://numpy.org/` (last accessed 10.07.2024)

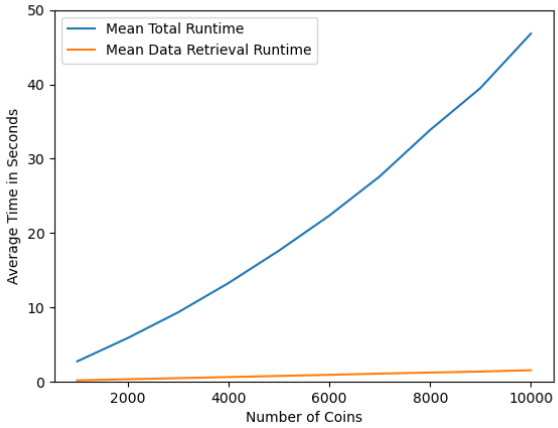


Figure 42: Mean runtime for the fourth use case using the AFEDempsterShaferAxiom (and CertaintyAssignmentAxioms) and synthetic data.

## 5.6. Use Case 5

The fifth use case can be solved using the classes ChainRuleAxiom, DisjointAxiom, SelfDisjointAxiom, and InverseAxiom for the chain rule reasoning. Additionally, the similarity values are aggregated using the AggregationAxiom and the coin type classifications using the DempsterShaferAxiom. For the benchmarks, the DataGenerator class generates coin type triples (for example, `ex:coin1 ex:isCoinType ex:cn_type_1`), coin type similarity triples (for example, `ex:cn_type_1 ex:similarTo ex:cn_type_2`), and class declarations for coins and coin (for example, `ex:cn_type_1 rdf:type ex:CoinType`). The benchmarks only used 100 coin types and only triples with a weight of at least 0.8 due to performance. The chain rule axioms also used this 0.8 as the output threshold for the similarity triples. This means the reasoner would only infer high similarity triples.

In Figure 43 on the left, you can see the results of the benchmark for this use case. Since the chain rule "If A similar to B and B similar to C then A similar to C" would infer similarity triples from each of the nodes to all the others it makes sense to define thresholds for the reasoning. Additionally, you can split this chain rule in 4 parts by defining the classes of A, B and C. This helps reduce memory usage during reasoning. Otherwise, you may get a pandas error that pandas cannot allocate enough memory for this dataframe operation. Even after setting those thresholds and only using 100 coin types, the performance for this use case is bad. The chain rule, that was mentioned before, is too costly in terms of reasoning runtime. Removing this chain rule improves the reasoner runtime a lot as you can see in Figure 43 on the right. However, the time complexity of this use case is still not linear. This could also be due to the simple reasoning approach of the chain rule stage. For each iteration, each axiom is

applied and then the reasoner checks whether at least one value changed or one new triple is inferred. If just one value changed or one new triple is inferred, then the reasoner does another iteration with all the data again. Future research could try to limit the reasoning on triples that could be affected by the new inferred weights.

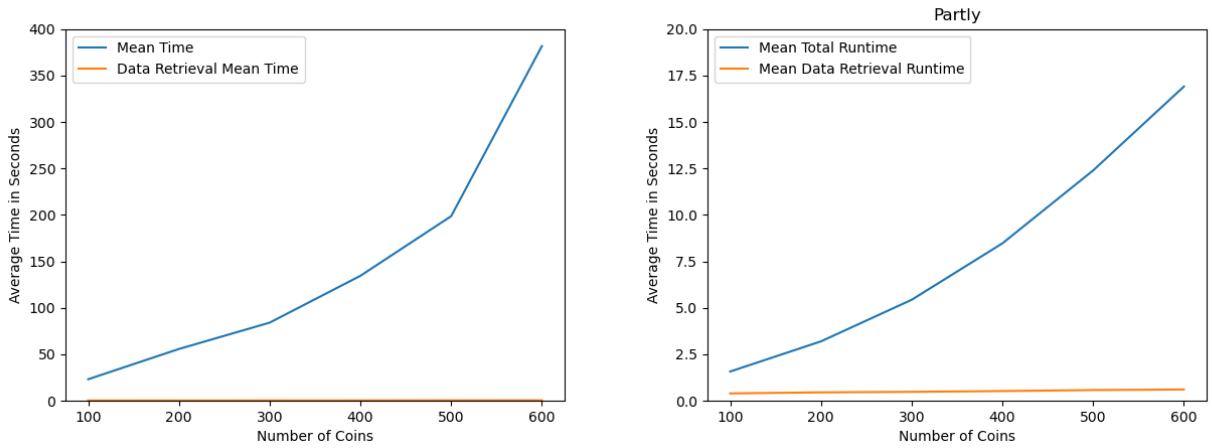


Figure 43: Reasoner benchmark for the fifth use case with synthetic data. Firstly, with all the axioms (left) and then without one chain rule that has the worst performance (right).

# 6. Conclusion and Future Work

When dealing with RDF graphs, a portion of the information is encoded implicitly in the graph and requires a reasoner to be utilized. This information could be inferred using chain rules such as: "If A is the parent of B and B is the parent of C then A is the grandparent of C". If the statements, used in the chain rules, are now uncertain or vague then you need a special kind of reasoner to handle those statements. This thesis implemented an Open-Source RDF reasoner with current reasoning approaches to solve vague and uncertain use cases for numismatic researchers. It utilizes techniques such as simple rule-based reasoning and is applicable for situations where you only have a few rules and do not need a reasoner that supports a big subset of OWL. The reasoner extends the AMT chain rule reasoning to also handle uncertain statements instead of only supporting vague statements. Additionally, it supports the creation of chain rules where one statement is vague and the other is uncertain. To support a wider variety of reasoning techniques, reasoning is now split into three stages. Simple aggregation functions and the Dempster-Shafer Theory can be used before the chain rule reasoning to aggregate certainties and precision degrees from multiple models or experts. The reasoner is then used to solve five of the six numismatic use cases defined in Table 8.

To achieve this, the reasoner of this thesis is written as an easy-to-use Python API. This means, that the reasoner is ready to be used with a few lines of code and many options to make the reasoner fit their graphs such as using user-defined predicates. By default, the reasoner supports two common approaches to model weights like certainties in the RDF graph: Reification and RDF-star. Regarding the performance, the preferred modeling approach is RDF-star but since RDF-star is not widely adopted yet, reification can be used for those triple stores. The communication to the RDF triple stores is implemented using W3C HTTP communication protocols. By using abstract classes, this reasoner also enables the users to extend the code to write their own axioms and SPARQL connectors for reasoning.

The benchmark results from section 5 showed the simplicity of the reasoner but also its drawbacks. All the use cases from section 3, except use case three where the reasoning approaches did not make sense, can now be solved with a few lines of code. Regarding the performance, the reasoner runtime highly depends on not using pure Python, especially Python for-loops, and instead using libraries like pandas that are optimized using C and Cython. Additionally, the RDF-star approach to model the weights showed significant runtime improvements over the reification approach which was expected regarding previous research [30].

In future research, you could improve the Python implementation of the reasoner by reducing pure python for loops like in Figure 35. Those for loops could be implemented using pandas, numpy, or Cython instead, increasing the performance of the reasoner. You could also advance

the reasoning logic to support more use cases. Currently, the reasoner does not support setting an upper bound of a statement's truthfulness when handling vague statements. Additionally, the reasoner can only handle statements that are either vague or uncertain. Future research could try to overcome those limitations by testing different or completely new reasoning approaches. Another topic for future research could be to try solving the sixth use case from subsection 3.7. The approaches, used in this thesis, were not applicable but there could be another technique to solve this use case.

# 7. References


[1] *AFE-Web*. URL: http://www.bigdata.uni-frankfurt.de/afe-web/ (visited on 07/10/2024).

[2] Hamdi Aloulou, Mounir Mokhtari, Thibaut Tiberghien, Jit Biswas, Clifton Phua, Jin Hong Kenneth Lin, and Philip Yap. "Deployment of assistive living technology in a nursing home environment: methods and lessons learned". In: *BMC medical informatics and decision making* 13.1 (2013), pp. 1–17.

[3] Hamdi Aloulou, Mounir Mokhtari, Thibaut Tiberghien, Romain Endelin, and Jit Biswas. "Uncertainty handling in semantic reasoning for accurate context understanding". In: *Knowledge-Based Systems* 77 (2015), pp. 16–28.

[4] *Ambient Assisted Living*. URL: https://wirtschaftslexikon.gabler.de/definition/ambient-assisted-living-53583/version-276661 (visited on 01/05/2024).

[5] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. "Dbpedia: A nucleus for a web of open data". In: *international semantic web conference*. Springer. 2007, pp. 722–735.

[6] Jürgen Bock, Peter Haase, Qiu Ji, and Raphael Volz. "Benchmarking OWL reasoners". In: *ARea2008-Workshop on Advancing Reasoning on the Web: Scalability and Commonsense*. Tenerife Spain. 2008.

[7] *CORPUS NUMMORUM – A Digital Research Infrastructure for Ancient Coins*. Zenodo, Feb. 2024. DOI: 10.5281/zenodo.10633905. URL: https://doi.org/10.5281/zenodo.10633905.

[8] Didier Dubois and Henri Prade. "Possibility theory, probability theory and multiple-valued logics: A clarification". In: *Annals of mathematics and Artificial Intelligence* 32 (2001), pp. 35–66.

[9] Birte Glimm, Ian Horrocks, Boris Motik, Giorgos Stoilos, and Zhe Wang. "HermiT: an OWL 2 reasoner". In: *Journal of automated reasoning* 53 (2014), pp. 245–269.

[10] Bernardo Cuenca Grau, Ian Horrocks, Boris Motik, Bijan Parsia, Peter Patel-Schneider, and Ulrike Sattler. "OWL 2: The next step for OWL". In: *Journal of Web Semantics* 6.4 (2008), pp. 309–322.

[11] Volker Haarslev, Ralf Möller, and Michael Wessel. "Querying the semantic web with Racer+ nRQL". In: *Proceedings of the KI-2004 International Workshop on Applications of Description Logics (ADL'04), Ulm, Germany*. Vol. 24. 2004.

[12] Petr Hájek. In: *Metamathematics of fuzzy logic* (1998).

[13] Olaf Hartig. *Position Statement: The RDF* and SPARQL* Approach to Annotate Statements in RDF and to Reconcile RDF and Property Graphs*. URL: https://blog.liu.se/olafhartig/2019/01/10/position-statement-rdf-star-and-sparql-star/ (visited on 01/05/2024).

[14] Pascal Hitzler, Markus Krotzsch, and Sebastian Rudolph. *Foundations of semantic web technologies*. CRC press, 2009.

[15] Brigitte Jaumard and Bruno Simeone. "On the complexity of the maximum satisfiability problem for Horn formulas". In: *Information Processing Letters* 26.1 (1987), pp. 1–4.

[16] David S Johnson. "Approximation algorithms for combinatorial problems". In: *Proceedings of the fifth annual ACM symposium on Theory of computing*. 1973.

[17] Yevgeny Kazakov, Markus Krötzsch, and František Simančík. "ELK: a reasoner for OWL EL ontologies". In: *System Description* (2012).

[18] Atanas Kiryakov, Damyan Ognyanov, and Dimitar Manov. "OWLIM–a pragmatic semantic repository for OWL". In: *International Conference on Web Information Systems Engineering*. Springer. 2005, pp. 182–192.

[19] Chang Liu, Guilin Qi, Haofen Wang, and Yong Yu. "Fuzzy reasoning over RDF data using OWL vocabulary". In: *2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology*. Vol. 1. IEEE. 2011, pp. 162–169.

[20] Chang Liu, Guilin Qi, Haofen Wang, and Yong Yu. "Reasoning with large scale ontologies in fuzzy pD* using MapReduce". In: *IEEE Computational Intelligence Magazine* 7.2 (2012), pp. 54–66.

[21] Thomas Lukasiewicz and Umberto Straccia. "Managing uncertainty and vagueness in description logics for the semantic web". In: *Journal of Web Semantics* 6.4 (2008), pp. 291–308.

[22] Christopher J Matheus, Ken Baclawski, and Mieczyslaw M Kokar. "Basevisor: A triples-based inference engine outfitted to process ruleml and r-entailment rules". In: *2006 Second International Conference on Rules and Rule Markup Languages for the Semantic Web (RuleML'06)*. IEEE. 2006, pp. 67–74.

[23] *MK-B | Augustus ca. 25-23 v. Chr.* URL: https://ikmk.smb.museum/object?id=18207657 (visited on 06/20/2024).

[24] *MK-B | Augustus ca. 25-23 v. Chr.* URL: https://ikmk.smb.museum/object?id=18207649 (visited on 06/20/2024).

[25] Ndapandula Nakashole, Mauro Sozio, Fabian M Suchanek, and Martin Theobald. "Query-time reasoning in uncertain RDF knowledge bases with soft and hard rules." In: *VLDS* 884.6 (2012), pp. 15–20.

[26] *Owlim*. URL: `https://www.w3.org/2001/sw/wiki/Owlim` (visited on 07/10/2024).

[27] *Pandas vectorization: faster code, slower code, bloated memory*. URL: `https://pythonspeed.com/articles/pandas-vectorization/` (visited on 04/06/2024).

[28] Bijan Parsia and Evren Sirin. "Pellet: An owl dl reasoner". In: *Third international semantic web conference-poster*. Vol. 18. 2004, p. 13.

[29] Hoifung Poon, Pedro M Domingos, and Marc Sumner. "A General Method for Reducing the Complexity of Relational Inference and its Application to MCMC." In: *AAAI*. Vol. 8. Chicago, IL. 2008, pp. 1075–1080.

[30] Jan Luca Pöpperl. "Modellierung von Unsicherheiten in Daten: Benchmarktests verschiedener Ansätze". In: (2023).

[31] Thanyalak Rattanasawad, Marut Buranarach, Kanda Runapongsa Saikaew, and Thepchai Supnithi. "A comparative study of rule-based inference engines for the semantic web". In: *IEICE TRANSACTIONS on Information and Systems* 101.1 (2018), pp. 82–89.

[32] *RDF-star Working Group Charter*. URL: `https://www.w3.org/2022/08/rdf-star-wg-charter/` (visited on 01/05/2024).

[33] *Reasoners and rule engines: Jena inference support*. URL: `https://jena.apache.org/documentation/inference/` (visited on 01/18/2024).

[34] Kari Sentz and Scott Ferson. "Combination of evidence in Dempster-Shafer theory". In: (2002).

[35] *Sesame to Eclipse RDF4J Migration*. URL: `https://rdf4j.org/documentation/reference/migration/` (visited on 07/10/2024).

[36] Robert DC Shearer, Boris Motik, and Ian Horrocks. "Hermit: A highly-efficient OWL reasoner." In: *Owled*. Vol. 432. 2008, p. 91.

[37] Parag Singla and Pedro Domingos. "Memory-efficient inference in relational domains". In: *AAAI*. 2006.

[38] Fabian M Suchanek, Gjergji Kasneci, and Gerhard Weikum. "Yago: a core of semantic knowledge". In: *Proceedings of the 16th international conference on World Wide Web*. 2007, pp. 697–706.

[39] Herman J. ter Horst. "Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary". In: *Journal of Web Semantics* 3.2 (2005). Selcted Papers from the International Semantic Web Conference, 2004, pp. 79–115. ISSN: 1570-8268. DOI: https://doi.org/10.1016/j.websem.2005.06.001. URL: https://www.sciencedirect.com/science/article/pii/S1570826805000144.

[40] Karsten Tolle and Sebastian Gampe. "Creating an Additional Class Layer with Machine Learning to counter Overfitting in an Unbalanced Ancient Coin Dataset". In: 2023. DOI: https://doi.org/10.5281/zenodo.8298078.

[41] Karsten Tolle and David Wigg-Wolf. "Uncertainty handling for ancient coinage". In: 2014, pp. 171–178.

[42] Martin Unold, Florian Thiery, and Allard Mees. "Academic Meta Tool. Ein Web-Tool zur Modellierung von Vagheit". In: *Zeitschrift für digitale Geisteswissenschaften, Die Modellierung des Zweifels—Schlüsselideen und-Konzepte zur Graphbasierten Modellierung von Unsicherheiten, Sonderband* 4 (2019).

[43] Ruben Verborgh and Jos De Roo. "Drawing conclusions from linked data on the web: The EYE reasoner". In: *IEEE Software* 32.3 (2015), pp. 23–27.

[44] *W3C Uncertainty Reasoning for the World Wide Web XG*. URL: https://www.w3.org/2005/Incubator/urw3/group/draftReport.html#classes (visited on 02/01/2024).

[45] Jens Weigel. "Handling Uncertainty in RDF Graphs for numismatic Use Cases". In: 2024. URL: http://www.bigdata.uni-frankfurt.de/wp-content/uploads/2024/01/Uncertainty_Jens_Weigel_Jan2024_public.pdf.

[46] *What Is the Semantic Web?* URL: https://www.ontotext.com/knowledgehub/fundamentals/what-is-the-semantic-web/ (visited on 06/09/2024).

[47] Lotfi Asker Zadeh. "Fuzzy sets". In: *Information and control* 8.3 (1965), pp. 338–353.

# List of Tables



# List of Figures

# Appendices

## A. RDF Namespaces

| Prefix | Namespace |
|---|---|
| rdf | http://www.w3.org/1999/02/22-rdf-syntax-ns# |
| rdfs | http://www.w3.org/2000/01/rdf-schema# |
| owl | http://www.w3.org/2002/07/owl# |
| nm | http://nomisma.org/id/ |
| nmo | http://nomisma.org/ontology# |
| ex | http://example.com/ |

Table 11: Table with RDF namespaces that are used in this thesis.

## B. Anaconda Environment

| Package | Version | Description |
|---|---|---|
| pandas | 2.1.4 | Pandas is used as a backend for storing the data and data transformation. |
| numpy | 1.26.4 | Numpy is heavily used by pandas and also utilized to create random synthetic data for the benchmarks. |
| requests | 2.31.0 | This package is used to communicate with the SPARQL endpoint via web protocols. |
| matplotlib | 3.8.4 | Matplotlib is used to create plots for the benchmark results. |

Table 12: Most important packages of the python environment used for the benchmarks. The full environment can be found on Github.

# C. Use Case Implementations

```
import SparqlConnector
import Reasoner

# Query endpoint
QUERY_ENDPOINT = "http://localhost:3030/test/query"
# Update endpoint
UPDATE_ENDPOINT = "http://localhost:3030/test/update"
# Graph Store Protocol endpoint
GSP_ENDPOINT = "http://localhost:3030/test/data"

conn = SparqlConnector.SparqlStarConnector(QUERY_ENDPOINT, UPDATE_ENDPOINT,
    GSP_ENDPOINT)

axioms = [
     Reasoner.CertaintyAssignmentAxiom("ex:issuer"),
     Reasoner.CertaintyAssignmentAxiom("ex:issuing_for"),
     Reasoner.CertaintyAssignmentAxiom("ex:domain_knowledge"),)
]

reasoner = Reasoner.Reasoner(axioms)
reasoner.load_data_from_endpoint(conn)
reasoner.reason()

df_result = reasoner.get_triples_as_df()
reasoner.save_data_to_file('result.ttl', conn)
reasoner.upload_data_to_endpoint(conn)
```

Figure 44: Code for the second use case using the reasoner of this thesis, assuming the triples are already in the triple store..

```
import SparqlConnector
import Reasoner

# Query endpoint
QUERY_ENDPOINT = "http://localhost:3030/test/query"
# Update endpoint
UPDATE_ENDPOINT = "http://localhost:3030/test/update"
# Graph Store Protocol endpoint
GSP_ENDPOINT = "http://localhost:3030/test/data"

conn = SparqlConnector.SparqlStarConnector(QUERY_ENDPOINT, UPDATE_ENDPOINT,
    GSP_ENDPOINT)

axioms = [
     Reasoner.DempsterShaferAxiom('ex:isCoinType')
]

reasoner = Reasoner.Reasoner(axioms)
reasoner.load_data_from_endpoint(conn)
reasoner.reason()

df_result = reasoner.get_triples_as_df()
reasoner.save_data_to_file('result.ttl', conn)
reasoner.upload_data_to_endpoint(conn)
```

Figure 45: Code for the third use case using the reasoner of this thesis, assuming the triples are already in the triple store..

```
import SparqlConnector
import Reasoner

# Query endpoint
QUERY_ENDPOINT = "http://localhost:3030/test/query"
# Update endpoint
UPDATE_ENDPOINT = "http://localhost:3030/test/update"
# Graph Store Protocol endpoint
GSP_ENDPOINT = "http://localhost:3030/test/data"

conn = SparqlConnector.SparqlStarConnector(QUERY_ENDPOINT, UPDATE_ENDPOINT,
    GSP_ENDPOINT)

axioms = [
     Reasoner.CertaintyAssignmentAxiom("ex:issuer"),
     Reasoner.CertaintyAssignmentAxiom("ex:issuing_for"),
     Reasoner.CertaintyAssignmentAxiom("ex:domain_knowledge"),
     Reasoner.AFEDempsterShaferAxiom('ex:issuer', 'ex:issuing_for',
         'ex:domain_knowledge')
]

reasoner = Reasoner.Reasoner(axioms)
reasoner.load_data_from_endpoint(conn)
reasoner.reason()

df_result = reasoner.get_triples_as_df()
reasoner.save_data_to_file('result.ttl', conn)
reasoner.upload_data_to_endpoint(conn)
```

Figure 46: Code for the fourth use case using the reasoner of this thesis, assuming the triples are already in the triple store..

```
import SparqlConnector
import Reasoner

# Query endpoint
QUERY_ENDPOINT = "http://localhost:3030/test/query"
# Update endpoint
UPDATE_ENDPOINT = "http://localhost:3030/test/update"
# Graph Store Protocol endpoint
GSP_ENDPOINT = "http://localhost:3030/test/data"

conn = SparqlConnector.SparqlStarConnector(QUERY_ENDPOINT, UPDATE_ENDPOINT,
    GSP_ENDPOINT)

axioms = [
     Reasoner.DempsterShaferAxiom('ex:isCoinType'),
     Reasoner.AggregationAxiom('ex:similarTo', 'mean'),
     Reasoner.ChainRuleAxiom('ex:similarTo', 'ex:similarTo', 'ex:similarTo',
                             'lukasiewicz'),
     Reasoner.ChainRuleAxiom('ex:isCoinType', 'ex:similarTo', 'ex:similarTo',
                             'product', sum_values=True),
     Reasoner.ChainRuleAxiom('ex:isCoinType', 'ex:isCoinTypeOf', 'ex:sameCoinTypeAs',
                             'product'),
     Reasoner.DisjointAxiom('ex:isCoinType', 'ex:similarTo', throw_exception=False),
     Reasoner.DisjointAxiom('ex:isCoinTypeOf', 'ex:similarTo',
         throw_exception=False),
     Reasoner.SelfDisjointAxiom('ex:similarTo', throw_exception=False),
     Reasoner.InverseAxiom('ex:similarTo', 'ex:similarTo'),
     Reasoner.InverseAxiom('ex:isCoinType', 'ex:isCoinTypeOf'),
     Reasoner.InverseAxiom('ex:isCoinTypeOf', 'ex:isCoinType'),
     Reasoner.InverseAxiom('ex:sameCoinTypeAs', 'ex:sameCoinTypeAs'),
]

reasoner = Reasoner.Reasoner(axioms)
reasoner.load_data_from_endpoint(conn)
reasoner.reason()

df_result = reasoner.get_triples_as_df()
reasoner.save_data_to_file('result.ttl', conn)
reasoner.upload_data_to_endpoint(conn)
```

Figure 47: Code for the sixth use case using the reasoner of this thesis, assuming the triples are already in the triple store..