FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Defining an Actor Ontology for Increasing Energy Efficiency and User Comfort in Smart Homes

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Information & Knowledge Management

eingereicht von

Dipl.-Ing. Simon Steyskal
Matrikelnummer 0828067

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Kastner
Mitwirkung: Dipl.-Ing. Dr.techn. Mario Kofler

Wien, 03.11.2014

(Unterschrift Verfasserin) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Defining an Actor Ontology for Increasing Energy Efficiency and User Comfort in Smart Homes

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Information & Knowledge Management

by

Dipl.-Ing. Simon Steyskal
Registration Number 0828067

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Kastner
Assistance: Dipl.-Ing. Dr.techn. Mario Kofler

Vienna, 03.11.2014

(Signature of Author) (Signature of Advisor)

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Erklärung zur Verfassung der Arbeit

Dipl.-Ing. Simon Steyskal
Anton Baumgartnerstraße 44 B3/106, 1230 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasserin)

Acknowledgements

First of all, I would like to thank Prof. Dr. Wolfgang Kastner, my supervisor, for the opportunity to write this thesis at the Institute of Computer Aided Automation at the Vienna University of Technology. Furthermore, I would like to thank my second supervisor Dipl.-Ing. Dr. Mario Kofler for his endless patience, extensive support and quick responses to uncountable many emails, that I've sent him.

I also want to thank my girlfriend Corinna, for being there for me all the time and her understanding in this time-consuming undertaking as well as my father for printing my thesis and poster.

Finally, special thanks go to my mother Siegrid and sister Sarah for their believe in me and their enormous support, especially in taking care of my two dogs, Faby and Pino.

Abstract

In the last years, the topic of smart home environments has gained more and more attention. Such an intelligent home offers several advantages to its users such as *(i) increased energy efficiency*, *(ii) cost reduction* or *(iii) supporting them in their daily life.* As part of a project called ThinkHome, an ontology-based intelligent home which utilizes artificial intelligence to improve control of home automation functions provided by dedicated automation systems, the present thesis aims at defining an actor preferences ontology which stores information about preferences of respective users. After classifying the different kinds of preferences and investigating the dependencies among them, relations to other already existing ontologies of the *ThinkHome* system as well as links to suitable external ontologies will be investigated. In order to be able to efficiently build the desired ontology, several ontology development approaches are explored and based on the principles of an approach called *METHONTOLGY*, the *Actor Preferences Ontology*, which enables the possibility to store, infer and schedule preferences and activities of actors within a smart home environment will be carried out. As addition to the development of a comprehensive ontology covering the domain of actor preferences, several state-of-the-art ontology reasoners will be evaluated, using domain-related reasoning tasks. The results of this evaluation can then be used to find the most suitable and best performing ontology reasoner for inferring new knowledge within the domain of the ThinkHome system.

Kurzfassung

In den letzten Jahren haben Smart Homes immer mehr an Bedeutung gewonnen. Ein Smart Home, oder auch *Intelligentes Wohnen*, offeriert seinen Bewohnern diverse Vorteile, wie zum Beispiel: *(i) gesteigerte Energieeffizienz*, *(ii) reduzierte Wohnkosten* und *(iii) Unterstützung im Alltag*. Als Teil eines Forschungsprojektes namens ThinkHome, ein auf Ontologien und künstlicher Intelligenz basierendes Smart Home, zielt die vorliegende Diplomarbeit auf die Definition einer Ontologie zur Speicherung von Präferenzen von Smart Home Akteuren ab. Hierzu werden zunächst die verschiedenen Typen von Präferenzen klassifiziert und deren Abhängigkeiten untereinander untersucht, um danach die Verbindungen zu anderen bereits existierenden Ontologien des ThinkHome Systems definieren zu können. Ebenso werden im Zuge dessen die Anknüpfungspunkte zu verwandten bzw. nützlichen externen Wissensbasen gesucht und diese gegebenenfalls integriert. Um die effiziente Entwicklung der angestrebten Ontologie gewährleisten zu können, werden unterschiedliche Entwicklungsansätze für Ontologien untersucht und schlussendlich ein Ansatz mit dem Namen *METHONTOLOGY* genauer beschrieben und verwendet. Nachdem die *Actor Preferences Ontology*, welche es ermöglicht Präferenzen zu speichern, neue abzuleiten bzw. sie zu planen entwickelt wurde, werden zusätzlich diverse aktuelle *Ontology Reasoner* basierend auf relevanten Aufgaben evaluiert. Dies ist notwendig um bei der Wahl des *Ontology Reasoners*, welcher für die Ableitung neuen Wissens im Rahmen des ThinkHome Projekts verantwortlich sein soll, jenen auswählen zu können, der für die Smart Home Domäne am besten geeignet ist.

Contents

CHAPTER 1

Introduction

Contents

1.1 Motivation

The topic of smart homes has become more and more popular over the last years. Besides the possibility to increase the energy efficiency of such homes by introducing automation technology to home environments, a smart home could furthermore support users in their daily life by remembering user preferences like room temperature or ambient light schemes as well as inferring new suitable preferences.

Since various aspects of such smart environments like multiple sensor and actuator data, user specific data and a high heterogeneity require a highly flexible technology which is able to deal with those kinds of data, Semantic Web technologies, especially ontologies and their reasoning capabilities have caught more and more attention.

Using ontologies as underlying knowledge base enables the possibility to deal with context on all levels required within a smart environment and furthermore offers the possibility to enrich the gathered sensor data with additional semantics. Based on those semantics, additional information can be derived, which can (e.g. in the domain of home automation systems) be used to increase energy efficiency or user comfort.

The present thesis is part of a research project called *ThinkHome* [77,78], which aims at developing an ontology-driven smart home system mainly serving the purpose of both providing efficient energy management of household appliances and increasing user comfort of its residents (cf. Section 3.2 for more detailed introduction). It consists of a knowledge base, represented as several interlinked ontologies which are responsible to store and maintain all data used within the system and a multi-agent system which is responsible to make decisions based on derived knowledge from the ontologies, learned experiences, and/or predefined rules.

Users of such a smart home system, whether they are human actors or system actors[1], represent a very important part of that ecosystem. Both main goals of the *ThinkHome* system (i.e. increasing *(i) energy efficiency and (ii) user comfort*) are related to these actors and especially to be able to provide a certain degree of user comfort, they have to be represented in the knowledge base. However, since persisting actors alone does not provide any user comfort at all, their preferences must be persistable too. Having both, actors and their preferences accessible to the multi-agent system, allows to increase user comfort of residents by automatically realizing the stored preferences and to increase energy efficiency by choosing the most energy preserving way to do that (e.g. the preference of *having a temperature value of 20°C in the living room* could be achieved by just opening the windows instead of using the air conditioner).

Thus, an ontology which is particularly dedicated to serve the purpose of storing information about actors and their preferences is needed and has to be integrated to the *ThinkHome* knowledge base.

1.2 Problem Statement and Aim of the Work

The already existing *ThinkHome* system implements several ontologies, which are responsible for e.g. storing and representing *Building & Architecture Information*, *Weather Data*, *User Behavior & Building Processes*, and *Energy & Resource Properties*. Additionally, a rudimentary actor preference ontology for representing actor information about the users in the system does already exist, but unfortunately, that ontology is neither complete nor does it satisfy the requirements stated to such an ontology like being able to store and schedule preferences, store occupancy information of the smart home, or automatically infer preference types based on their characteristics.

As the scope of modeling actors and their preferences together with the requirements we impose at an ontology which is capable of representing that kind of information is a rather domain specific one, we cannot reuse existing ontologies that might offer similar capabilities. Aside from that fact - to the best of our knowledge - there does not exists any ontology modeling human/system actors and their preferences.

Thus, aim of the present master thesis is the development of a comprehensive actor preference ontology, which is capable of storing preferences grouped in preference profiles for actors of the *ThinkHome* system, as well as providing additional semantics

[1] Although the present thesis primarily focuses on human actors.

to those information. These preferences must be assignable to time and location information which makes them schedulable and offers users the possibility to relate them to certain preference schedules that can be defined for certain e.g. days, weeks. Beside common preferences, this newly developed ontology shall cover the definition of activities, which groups several preferences together that shall be valid for the activity they are part of.

To summarize, users should be able to:

1. represent themselves in the ontology
2. define several different types of preferences
3. define activities and preferences which should be active whenever the said activity is performed
4. schedule preferences and activities
5. group preferences, activities, preference schedules, and activity schedules in arbitrary many preference profiles
6. state time frames the home will be unoccupied

The smart home system should be able to:

1. automatically derive types of concepts based on their characteristics
2. choose appropriate and the most energy preserving way to carry out tasks
3. detect inconsistencies within the ontology
4. re-schedule preferences within their time frames if necessary
5. use occupancy information of the smart home to efficiently carry out scheduled tasks

Following the principles of the Semantic Web, the developed ontology shall be highly interwoven with the existing *ThinkHome* and other related ontologies and must reuse their concepts whenever it is appropriate.

Although ontologies enable reasoners to infer new knowledge based on the present information, performing such reasoning tasks often comes in hand with performance issues like an extensive runtime, which makes it difficult to select the best performing reasoner for the problem. For that purpose, current state-of-the-art ontology reasoners are evaluated and compared with each other, using reasoning tasks related to the developed ontology.

1.3 Methodological Approach

The present thesis follows the well-known design science paradigm in information systems, proposed by Hevner et al. [100] which can be broken down into following steps:

1. **Design as an Artifact.** The aim of the present thesis is develop and define an ontology to represent the preferences of actors of a smart home system in order to increase the energy efficiency and user comfort within such a smart home environment. More precisely, following artifacts will be built in the course of this thesis:
    1. An investigation and analysis of various ontology development mechanisms especially regarding their applicability for our use cases.
    2. A thorough definition of the requirements and competency questions such an ontology must fulfill / stick to.
    3. An *Actor Preferences Ontology* which covers all previous stated requirements and which was developed following the most suitable and applicable ontology development approach discovered during previous investigations.
    4. Evaluation of current state-of-the-art OWL reasoners, based on reasoning tasks that were taken out on the previous developed, and other related ontologies.
2. **Problem Relevance.** The presence of an ontology which is capable of persisting preferences of smart home actors is crucial for a system which aims to achieve the goals of increasing energy efficiency and user comfort. For our use-case such an ontology must be integrated within an already existing ontology-based smart home system and thus, must be developed from scratch.
3. **Design Evaluation.** The ontology developed in the course of the present thesis will be evaluated by checking it against the imposed functional and non-functional requirements.
4. **Research Contributions.** Although no particular scientific contribution besides the present thesis are planned, the insights gained and documented by the evaluation of OWL reasoners as well as the development of the *Actor Preferences Ontology* can serve interested readers as starting point for further investigations.
5. **Research Rigor.** Some of the aforementioned challenges were already partially investigated in previous research, especially regarding ontology development mechanisms and OWL reasoner evaluations. Therefore, surveys and literature research will be carried out in the starting phase of the thesis before its artifacts will be evaluated and built, based on the gathered knowledge.

6. Design as a Search Process. The artifacts developed throughout the present thesis will be iteratively built and evaluated. This means that parts of the artifacts will be tested on simple sample examples and eventually adapted before more complicated scenarios will be taken into consideration.

1.4 Structure of the Work

The present thesis is structured as follows:

Chapter 2: introduces the preliminaries of this work, namely: the *concept of ontologies* 2.1, *RDF* 2.2, *RDFS* 2.3, *OWL* 2.4, *SPARQL* 2.5, and the *reused external ontologies* 2.6 *OWL-Time* and the *Ontology of Units of Measure*.

Chapter 3: defines the concept of smart homes 3.1, discusses the *ThinkHome system* itself 3.2, *energy efficiency and user comfort within* ThinkHome 3.3 & 3.4, and the *benefits of using ontologies as datastores* 3.5 before it emphasizes *related work* 3.6 in the field of related ontologies, modeling a similar domain as the *Actor Preferences Ontology*.

Chapter 4: investigates different kinds of ontology development tools or approaches such as: *Ontology Design Patterns* 4.1, *Methodology by Uschold & King* 4.2.1, *METHONTOLOGY* 4.2.2, the *UPON approach* 4.2.3, *Ontology 101* 4.2.4, *Ontology Learning* 4.3 and then focuses on explaining the approach of *METHONTOLOGY* 4.4 as chosen ontology development strategy in more detail.

Chapter 5: represents the main chapter of the present thesis and covers essential ontology development steps together with their documentation artifacts. The discussed steps are: *Specification* 5.1, *Conceptualization* 5.2, *Integration* 5.3, *Implementation* 5.4 and *Evaluation* 5.5.

Chapter 6: evaluates current *state-of-the-art OWL ontology reasoners* 6.1 based on *domain related reasoning tasks* 6.2 and illustrates the archived *results* 6.5.

Chapter 7: concludes the present thesis and gives an outlook on potential *further work* 7.1.

Appendix A: contains all documentation artefacts which were created during the development process.

Appendix B: contains the glossary of terms for properties of the ontology.

Appendix C: contains the glossary of terms for concepts of the ontology.

CHAPTER 2

Preliminaries

Contents

In the following chapter, we will introduce the main Semantic Web technologies used within this thesis[1] and conclude with a brief description of reused ontologies within the *Actor Preferences Ontology*.

2.1 Ontologies

There exist many similar definitions for the term *ontology* and probably one of the most-cited ones was presented by Gruber in 1993:

[1] Parts of this introduction were already published in [85] and reused in order to provide a self-contained thesis but at the same time avoid redundant work.

An ontology is an explicit specification of a conceptualization. [34]

So basically, ontologies are generic conceptual models of a domain of interest. A more formal definition of an ontology is shown underneath:

Definition 1. We consider an ontology as a triple:

$$O = \langle C, I, P \rangle$$

C - Set of Classes Classes or concepts are abstract representations of objects. They can be subsumed by other classes and inherit their properties. Furthermore - if not stated otherwise - a class can inherit from more than one superclass.

I - Set of Individuals Individuals are specific representations of objects and usually describe a very concrete type of concepts. The choice whether an object should be modeled as individual or as a class is often not very easy to make and heavily relies on the modeling domain. For example an object `Integer` can either be considered as a subclass of the concept `Datatype` having an individual called „`1`", or modeled as individual of the concept `Datatype` in the absence of more specific objects like „`1`".

P - Set of Properties P contains all properties which define data values for specific attributes (names, ids, . . .) and relations, describing possibilities to relate entities in ontologies with each other (subclass, equivalence relations, . . .).

In contrast to other structures which aim for storing data in a defined way, like relational databases, ontologies enable the possibility to store semantics of data, together with specific rules which describe the schema. The possibility to infer new knowledge based on the available data as well as to be able to detect semantic conflicts between the entities of an ontology are additional benefits for using ontologies over common databases to represent and store data.

To understand the principles behind the Semantic Web, some of its major technologies and standards are described in the following chapter and some of them are represented in the *Semantic Web Stack* in Figure 2.1.

Built upon the *URI/IRI* layer, all higher layers in the *Semantic Web Stack* can uniquely identify their defined resources by URIs (Uniform Resource Identifier) and IRIs (Internationalized Resource Identifier) which are common resource identifiers in the World Wide Web.

The next layers are XML (eXtensible Markup Language) and RDF (Resource Description Framework), which describe the basic language of the Semantic Web and using RDF, which is based on the XML format, enables the possibility to describe resources both in a human-readable and machine-processable way, which will be described together with its most common formats in Section 2.2.

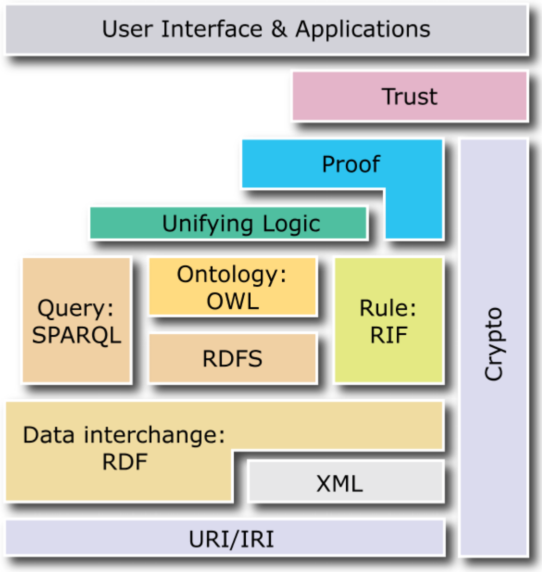

Figure 2.1: The Semantic Web stack [9]

There currently exist two extensions for RDF, namely *(i) RDF Schema* 2.3, and *(ii) the Web Ontology Language* 2.4. Only by the use of these extensions it is possible to define and model ontologies, since RDF does not provide the possibilities for describing properties or complex relations between resources [20].

In the following sections, we will describe selected parts of the Semantic Web stack in more detail.

2.2 Resource Description Framework (RDF)

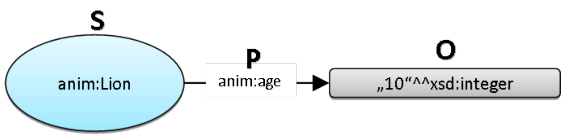

Figure 2.2: Simple RDF graph

The Resource Description Framework (RDF) [38] became a W3C recommendation in 1999 [57] and was revised several times until it became its last W3C recommendation in 2004 [4]. It is a framework for describing and representing information about resources in the World Wide Web and is both human-readable and machine-processable, which enables the possibility to easily exchange information among different applications using RDF triples, but still be easy to read.

In RDF everything is a resource, uniquely identified by its URI and all data is represented as (subject - predicate - object) triples, where subjects and predicates are URIs and objects can either be literals (strings, integers, ...) or URIs as shown in Figure 2.2.

Furthermore, subjects or objects can be represented using *blank nodes*, those *blank nodes* do not have a corresponding URI which could identify them and are usually used to express anonymous resources (e.g. »Pino has a friend who is 24 years old« where a *blank node* would represent the anonymous friend of Pino.)

Since one RDF triple usually does not describe a resource entirely, more triples are defined and combined in an *RDF Graph*. Such an *RDF Graph* connects those triples by a simple AND operator and can therefore easily be merged with other *RDF Graphs* without losing entailment information, relying on RDFs monotonicity of semantic extensions [38].

2.2.1 RDF Serialization Formats

There exist several formats for representing and serializing RDF data such as *Turtle(N3)* [5, 7] and *RDF/XML* [4]. In this section we will briefly explain each format and give an example serialization of a small sample triple set which is depicted in Figure 2.3.

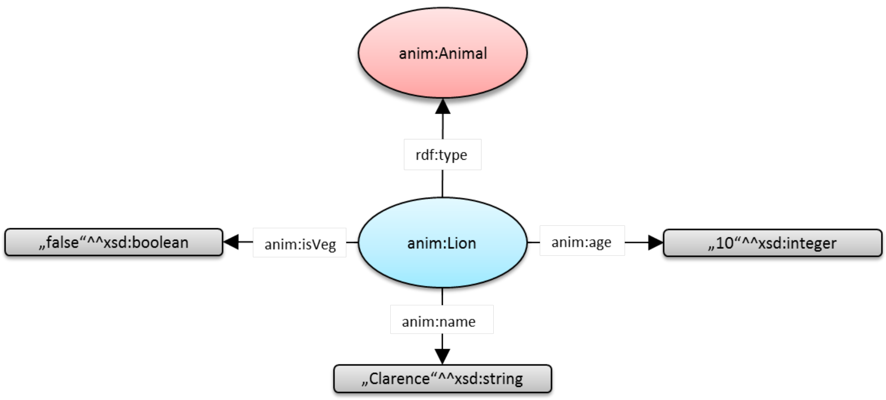

Figure 2.3: An RDF graph describing an animal domain.

2.2.1.1 Turtle and N3

The *Terse RDF Triple Language*, or *Turtle* [5], is a very lightweight and easy readable subset of the *Notation3* serialization format for RDF and became a W3C Candidate Recommendation in February 2013.

It is commonly used for representing ontologies since it perfectly illustrates the nature of RDF to model data as triples. The simplest statement using *Turtle* consists

subject	predicate	object
anim:Lion	rdf:type	anim:Animal
anim:Lion	anim:age	10
anim:Lion	anim:name	Clarence
anim:Lion	anim:isVeg	false

Table 2.1: The RDF triples encoded within the RDF graph shown in Figure 2.3

of a `subject`, `predicate` and `object` which are separated using whitespaces and terminated by a dot. Furthermore, it is possible to omit the leading subject, if several triples only vary in their predicates and objects but have the same subject, by terminating each triple (but not the last one) with a semicolon instead of a dot. Listing 2.1 shows the representation of a sample ontology using *Turtle*.

All examples within this thesis are serialized using the *Turtle* format.

Listing 2.1: Turtle representation of the RDF graph in Figure 2.3

```
@prefix : <http://ontology.org/onto1.owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix anim: <http://www.example.org/animal_onto#> .
@base <http://www.w3.org/2002/07/owl#> .

<http://www.example.org/animal_onto/Lion>
       rdf:type <http://www.example.org/animal_onto/Animal>;
       anim:name "Clarence" ;
       anim:isVeg "false" ;
       anim:age "10" .
```

2.2.1.2 RDF/XML

RDF/XML is the most common format for representing ontologies and natively supported by all RDF parsers. Unlike *Turtle*, it is an XML-based serialization of RDF, which inevitably leads to a larger overhead when representing RDF triples. It was introduced together with the RDF specification in 1999 and became a W3C Recommendation in February 2004.[2] As shown in Listing 2.2, RDF/XML serializations tend to be verbose and more difficulty readable by humans. An approach to make *RDF/XML* more concise was proposed by Brickley in 2002 and is called „Striped RDF/XML Syntax“ [10]. The striped *RDF/XML* syntax introduces XML elements for nodes and arcs of an RDF graph and provides the possibility to group triples as shown in Listing 2.3.

[2]http://www.w3.org/TR/REC-rdf-syntax/

Listing 2.2: RDF/XML description of the RDF graph in Figure 2.3

```
<?xml version="1.0"?>

<rdf:RDF xmlns="http://ontology.org/onto1.owl#"
     xml:base="http://ontology.org/onto1.owl"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:anim="http://www.example.org/animal_onto#">

    <rdf:Description rdf:about="http://www.example.org/animal_onto/Lion">
        <rdf:type rdf:resource="http://www.example.org/animal_onto/Animal"/>
    </rdf:Description>

    <rdf:Description rdf:about="http://www.example.org/animal_onto/Lion">
        <anim:name>Clarence</anim:name>
    </rdf:Description>
        ....
</rdf:RDF>
```

Listing 2.3: Striped RDF/XML description of the RDF graph in Figure 2.3

```
<?xml version="1.0"?>
<rdf:RDF xmlns="http://ontology.org/onto1.owl#"
     xml:base="http://ontology.org/onto1.owl"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:anim="http://www.example.org/animal_onto#">

    <anim:Animal rdf:about="http://www.example.org/animal_onto/Lion">
      <anim:name>Clarence</anim:name>
      <anim:age>10</anim:age>
      <anim:isVeg>false</anim:isVeg>
    </anim:Animal>
</rdf:RDF>
```

2.3 RDF Schema (RDFS)

Although RDF provides the basic elements and tools for describing web resources, it does not offer the possibility to describe relations or constraints between entities and therefore is not able to describe ontologies. This lack of functionality included the development of *RDF Schema (RDFS)*, which is a semantic extension to the basic RDF specification and provides the capability to describe properties and relations among resources and therefore offers basic elements for ontology description.

RDFS was firstly published in 1998 and became a W3C recommendation in 2004 [11].

RDFS now divides resources into two groups:

Classes: The first group of resources is called classes. Those classes are usually identified by URIs and described using RDF properties, a member of a specific class is called its instance, which is denoted by the rdf:type property. A class can have a set of instances of itself, which is called its class extension. Furthermore classes can share the same set of instances although they might be different classes (e.g. Alice defines dogs as animals and Bob defines them as carnivores, it is possible for those two classes to have the same instances but of course, different properties).

RDFS introduces subclass relations among classes, namely if there exists a class A which is a subclass of a class B, then all instances of A will also be instances of B. Vice versa, if a class B is a superclass of a class A, then all instances of A are also instances of B. The `rdfs:subClassOf` property may be used to represent this subclass relation.

A small sample ontology using *RDFS* features and describes a teacher/pupil domain is shown in Figure 2.4.

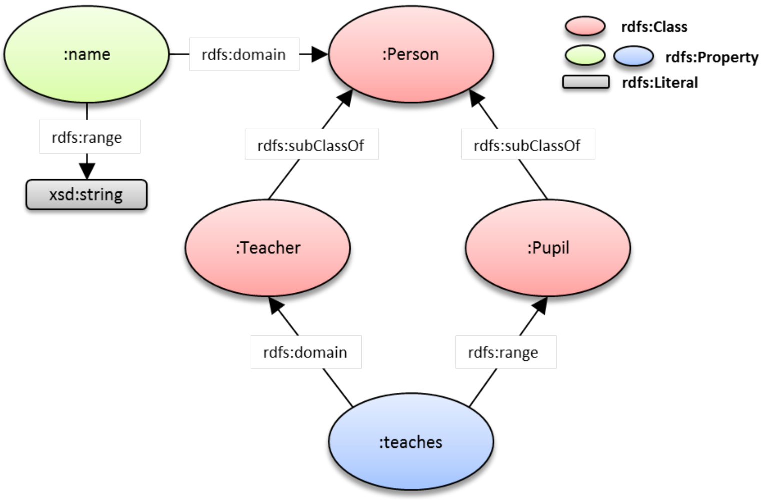

Figure 2.4: Small sample ontology using RDFS features

Important Classes:

rdfs:Resource: Everything described in RDF is a resource and instance of the class `rdfs:Resource`. `rdfs:Resource` is an instance of `rdfs:Class` and all other classes are its subclasses.

rdfs:Class: `rdfs:Class` is an instance of `rdfs:Class` and is the class of resources that are RDF classes. (cf. `:Teacher`)

rdfs:Literal: As mentioned earlier, an object of an RDF triple might be a literal. Those literals are instances of the class `rdfs:Literal` and divided into typed literals, which are instances of respective datatype class, and plain literals.

rdfs:Datatype: This class describes the class of datatypes and all instances of it are related to a datatype described in the RDF Concepts specification citerdf2. It is both an instance and a subclass of `rdfs:Class` and each instance of `rdfs:Datatype` is a subclass of the `rdfs:Literal`.

rdf:Property: `rdf:Property` is the class of RDF properties and an instance of `rdfs:Class`. (cf. `:teaches`)

Properties: The second group of resources are properties which are defined by [51] as relations between subject resources and object resources.

Like the subclass relation, *RDFS* also introduces the concept of subproperties. If a property A is a subproperty of a property B, then all resources which are connected by A are also connected by B and vice versa. This subproperty relation indicated by the `rdfs:subPropertyOf` property.

Important Properties:

rdfs:domain This property states that any resource which has a given property must be an instance of the class referenced by `rdfs:domain`. (cf. `:teaches` and `:Teacher`)

rdfs:range `rdfs:range` is used to state that the values of a given property are instances of the class referenced by `rdfs:range`. (cf. `:teaches` and `:Pupil`)

rdf:type An important property which states that a resource is an instance of a class.

rdfs:subClassOf & rdfs:subPropertyOf As mentioned above, these properties are used to state the subclass and subproperty relations among classes and properties. Both are instances of `rdf:Property`. (cf. `:Teacher` and `:Pupil` are subclasses of `:Person`)

rdfs:label & rdfs:comment These properties are instances of `rdf:Property` and may be used to provide a human-readable description of the resource itself as well as its name.

2.4 Web Ontology Language (OWL)

Although *RDFS* allows the representation of simple ontologies by using properties, which describe the hierarchical relation among entities, it lacks in the support of defining more sophisticated entity relations (e.g. disjointness), cardinality (e.g. exactly one), equality (e.g. equivalences between classes/properties/instances) and characteristics of properties (e.g. symmetry). For that purpose the *Web Ontology Language (OWL)*, which was firstly published in 2002 and became a W3C recommendation in 2004 [64], was developed. Since 2012 an extension to *OWL*, called *OWL 2*, is available as W3C recommendation [32].

In general, *OWL* is used to describe complex ontologies and furthermore introduce the possibility to automatically process the content in the given ontology by using the previous mentioned constructs which were not available in *RDFS*.

2.4.1 OWL Sub-languages

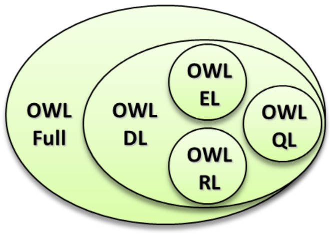

Figure 2.5: The OWL sub-language hierarchy as indicated in [64]

In order to sufficiently fulfill the different requirements of ontologies and especially avoid an unnecessary increase of complexity of those ontologies, three different sub-languages of *OWL* were developed, namely: *OWL Lite*, *OWL DL* and *OWL Full*. Each of these sub-languages is a subset of the more complex one as indicated in Figure 2.5. As a result, following validity conclusions hold [64]:

OWL EL/QL/RL: These three profiles introduce restrictions on *OWL* in order to allow more efficient reasoning. *OWL EL* provides the expressiveness of large-scale ontologies but only needs polynomial time for selected reasoning problems such as classification and instance checking. *OWL QL* is used to implement sound and complete query answering on top of relational databases and *OWL RL* provides the possibility to run rule-based reasoning algorithms in polynomial time.

OWL DL: Increasing the expressiveness of *OWL EL/QL/RL* but still be computational complete and decidable, leads to *OWL DL*, which is translatable into the expressive Description Logic *SROIQ* [3] . Although it includes all language concepts of *OWL*, they can only be used under special conditions (e.g. a class cannot be an instance of another class, but of course be its subclass).

OWL Full: Losing the restriction of using *OWL* language constructs only under certain conditions and therefore retrieving the most expressiveness and syntactic freedom for defining OWL ontologies, unfortunately comes in hand with the loss of computational guarantees. As indicated in [64], it is very unlikely, that any reasoning software will be able to support complete reasoning for every feature of OWL Full.

Remark: Although every ontology expressed in *OWL* is a valid *RDF* document, not every *RDF* document is a valid *OWL* ontology. Only *OWL Full* is a complete extension of *RDF*, whereas *OWL DL* and the profiles *OWL EL/QL/RL* are restricted extensions of *RDF*. When migrating from an *RDF* document to an *OWL DL/EL/QL/RL* ontology, those restrictions must be fulfilled.

2.4.2 OWL Features

OWL introduces many new features for describing information and knowledge about a domain and is even more expressive than *RDFS*. We will now introduce some of this features in more detail based on a sample ontology, illustrated in Figure 2.6.[3]

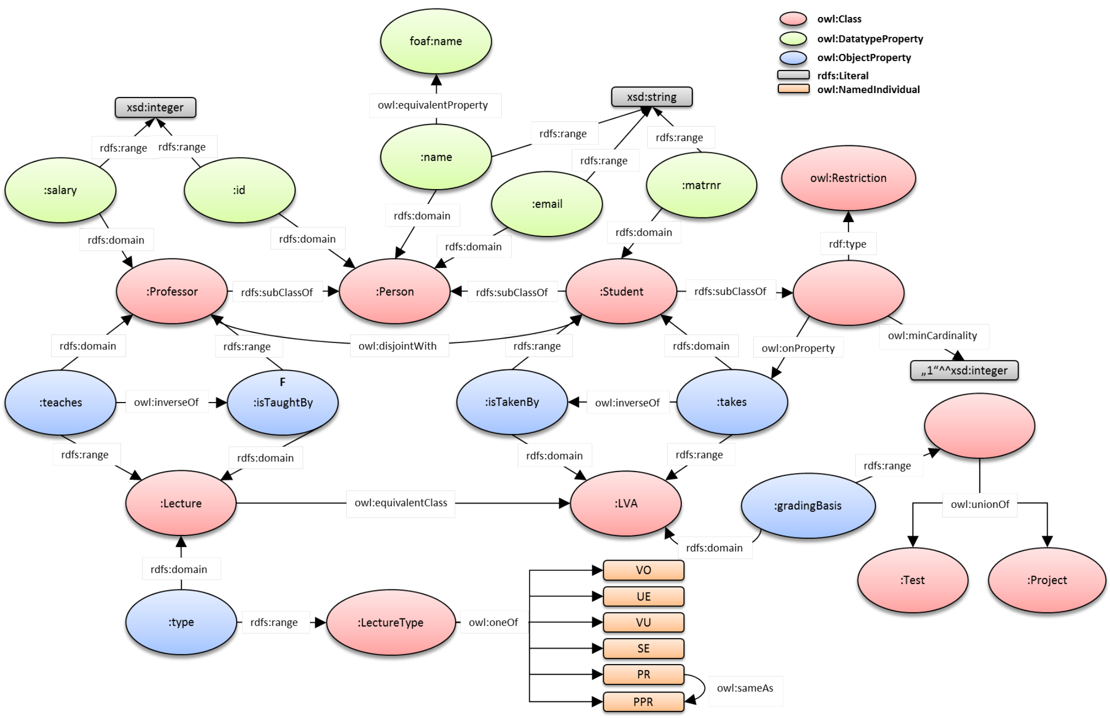

Figure 2.6: Sample ontology, which uses selected OWL features

[3]Note that the following list is only a small subset of the features *OWL* provides and only contains those features of *OWL*, which we use in the present thesis.

2.4.2.1 Properties

owl:DatatypeProperty Datatype properties link individuals to data values. (cf. `:id`, `:email`)

owl:ObjectProperty Object properties are used to define relations between classes (i.e. link individuals to individuals). (cf. `:isTakenBy, :takes`)

owl:FunctionalProperty If a property is defined as `owl:FunctionalProperty` it can have only one unique value for each instance of this property. (cf. `:isTaughtBy`; a `:Lecture` can only be held by one `:Professor`)

2.4.2.2 Relations between Entities

owl:equivalentClass This property is used to define the similarity between two classes. (cf. `:Lecture` and `:LVA`)

owl:equivalentProperty This property is used to define the similarity between two properties. (cf. `:name` is equivalent to the property `foaf:name`, defined in the FOAF ontology[4])

owl:sameAs This property is used to define the similarity between two instances. (cf. the two lecture types `:PR` and `:PPR` are equivalent)

owl:inverseOf Using `owl:inverseOf` offers the possibility to state an inverse similarity between two properties. (cf. `:isTakenBy` and `:takes`)

owl:disjointWith Using `owl:disjointWith` offers the possibility to state that two entities are disjoint. (cf. an instantiation of `:Professor` cannot be a `:Student` too)

2.4.2.3 Boolean Connectives and Enumeration

owl:unionOf This property links a class to a union of class descriptions. (cf. *:gradingBasis*, its `rdfs:range` is a blank node, which is defined as union of `:Test` and `:Project`)

owl:oneOf This property is used to define enumerations within ontologies. (cf. `:LectureType` contains one of the listed individuals)

[4]http://xmlns.com/foaf/spec/

2.4.2.4 Restrictions

owl:Restriction Restrictions are subclasses of `owl:Class` and are used to define value constraints for specific properties. (cf. a `:Student` must take at least one `:Lecture`)

owl:onProperty The `owl:onProperty` property defines the specific property for which the restriction holds. (the above mentioned restriction is defined for the property `:takes`)

owl:(min | max)Cardinality One example of a restriction mentioned above are cardinality constraints. Whereas `owl:minCardinality` and `owl:maxCardinality` define lower and upper bounds for cardinalities, the `owl:Cardinality` property is used to define a precise value for the cardinality. (`owl:minCardinality 1` is used to state, that a `:Student` must take at least one `:Lecture`)

2.4.3 OWL Serialization Formats

As an addition to the previous defined serialization formats for RDF 2.2.1, some serialization formats were primarily developed to represent ontologies using *OWL* features. In the following we will briefly introduce two of those formats based on the example shown in Figure 2.4.

2.4.3.1 OWL Manchester Syntax

The Manchester OWL syntax [40] is a very light-weight and user-friendly serialization format for OWL and has its intended use-case in representing OWL descriptions, although it is also able to represent entire OWL ontologies. Famous ontology development editors such as Protégé 4.x [1] use the Manchester syntax for defining and displaying descriptions associated with entities. An example of an ontology represented using the OWL Manchester syntax is shown in Listing 2.4.

Listing 2.4: Sample Ontology in OWL Manchester Syntax

```
Prefix: : <http://ontology.org/onto1.owl#>
Prefix: owl: <http://www.w3.org/2002/07/owl#>
Prefix: rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
Prefix: xsd: <http://www.w3.org/2001/XMLSchema#>
Prefix: rdfs: <http://www.w3.org/2000/01/rdf-schema#>

Ontology:

AnnotationProperty: rdfs:label
Datatype: xsd:string

ObjectProperty: teaches
    Domain:
```

```
        Teacher
    Range:
        Pupil

DataProperty: hasName
    Domain:
        Person
    Range:
        xsd:string

Class: Teacher
    Annotations:
        rdfs:label "A person who teaches pupils"@en
    SubClassOf:
        Person and teaches some Pupil

Class: Person
    SubClassOf:
        hasName exactly 1 xsd:string

Class: Pupil
    Annotations:
        rdfs:label "A person whos taught by teachers"@en
    SubClassOf:
        Person
```

2.4.3.2 OWL Functional Syntax

In December 2012, *OWL2* together with its functional-style syntax became a W3C Recommendation[5]. It was initially used for defining *OWL2* in its W3C specifications and tends to be a clean and easy parseable, adjustable and modifiable ontology format. Nevertheless it is less intuitive and human readable than *Turtle* since its definition of statements does not follow the simple `subject`, `predicate` and `object` principle as illustrated in Listing 2.5.

Listing 2.5: Sample Ontology in OWL Functional Syntax

```
Prefix(:=<http://ontology.org/onto1.owl#>)
Prefix(owl:=<http://www.w3.org/2002/07/owl#>)
Prefix(rdf:=<http://www.w3.org/1999/02/22-rdf-syntax-ns#>)
Prefix(rdfs:=<http://www.w3.org/2000/01/rdf-schema#>)

Ontology(
        Declaration(Class(:Person))
            SubClassOf(:Person DataExactCardinality(1 :hasName xs:string))

        Declaration(Class(:Pupil))
```

[5]http://www.w3.org/TR/owl2-syntax/

```
        AnnotationAssertion(rdfs:label :Pupil "A␣person␣whos␣taught␣
            by␣teachers"@en)
        SubClassOf(:Pupil :Person)

    Declaration(Class(:Teacher))
        AnnotationAssertion(rdfs:label :Teacher "A␣person␣who␣teaches
            ␣pupils"@en)
        SubClassOf(:Teacher ObjectIntersectionOf(ObjectSomeValuesFrom
            (:teaches :Pupil) :Person)

    Declaration(ObjectProperty(:teaches))
        ObjectPropertyDomain(:teaches :Teacher)
        ObjectPropertyRange(:teaches :Pupil)

    Declaration(DataProperty(:hasName))
        DataPropertyDomain(:hasName :Person)
        DataPropertyRange(:hasName xsd:string)
)
```

2.4.4 OWL Reasoning

One of the core features of *RDFS* and *OWL* is, that you can exploit their constructs to infer new knowledge based on the stored semantics within the ontology. Nevertheless, if you extend your ontology by using *OWL* constructs and did not restrict it at least by the rules which were defined for the previous introduced sublanguage of *OWL*, *OWL DL*, reasoning becomes undecidable [45].

A very simple example of such an exploitation is shown in Listing 2.6, where we can infer that `:ComfortTemperaturePref` is not only a `:TemperaturePreference` but also a `:Preference` based on the semantics of the `rdfs:subClassOf` relation [11].

Listing 2.6: Inferring new knowledge using reasoning

```
:TemperaturePreference rdfs:subClassOf :Preference .
:ComfortTemperaturePref rdf:type :TemperaturePreference .
```

For our application domain of an ontology based and energy aware home automation system, reasoning is a crucial part in order to realize artificial intelligence based control strategies that allow maximizing energy efficiency and user comfort [78].

2.5 SPARQL Protocol And RDF Query Language (SPARQL)

The last Semantic Web technology we want to discuss is the standard query language for RDF called SPARQL [74], which has become a W3C Recommendation in version 1.1 in 2013 [37]. Its syntax is highly influenced by the previous introduced RDF serialization format Turtle [5] and SQL [17] a query language for relational data.

Besides basic query operations such as union of queries, filtering, sorting and ordering of results as well as optional query parts, version 1.1 extended SPARQLs portfolio by aggregate functions (`SUM, AVG, MIN, MAX, COUNT,...`), the possibility to use subqueries, perform update actions via SPARQL Update and several other heavily requested missing features [73].

A query, illustrating some of the features offered by SPARQL is shown in Listing 2.7. It queries recursively (indicated by the * after the property) for all classes which are subclasses of `act:TemperaturePreference` and then asks for all individuals, which are instantiations of those classes. After receiving all relevant preference values stored in variable `?prefValue` only those having a `?value` over 20 are kept. In the end the average of the remaining values is calculated using the aggregate function `AVG`.

Listing 2.7: Querying the average temperature of temperature preferences having a value over 20 using SPARQL

```
PREFIX act: <https://www.auto.tuwien.ac.at/.../ActorOnt.owl#>
SELECT (AVG(?value) as ?avg)
WHERE {
   ?tmp rdfs:subClassOf* act:TemperaturePreference.
   ?preference rdf:type ?tmp;
               act:hasPreferenceValue ?prefValue.
   ?prefValue act:hasValue ?value.
   FILTER(?value > 20)
 }
```

?avg
"21.86667 "^^xsd:float

Table 2.2: Results of query shown in Listing 2.7

2.6 Linked Ontologies

One of the major concepts of several ontology development approaches (e.g. [23,69,92]) is the reuse of already existing ontologies and vocabularies whenever it is suitable [83]. The advantage of this strategy is, that (*i*) it saves time since you do not have to redevelop already existing concepts, and (*ii*) it supports the interlinking with other ontologies since they might use the same concepts to describe their resources.

For the development of our *Actor Preferences Ontology* we used two already existing ontologies to describe time concepts (Time Ontology [39]) and to describe units of measurement (Ontology of Units of Measure and Related Concepts [79]), which are described in more detail in the following subsections.

2.6.1 Time Ontology (owl-time)

As mentioned above, we used the OWL-Time ontology, offered by the W3C as working draft [39] since 2006, to be able to define temporal properties. One of the main concepts of this ontology is `time:TemporalEntity` which can either be a `time:Instant` or `time:Interval`. The latter two concepts are especially useful in our domain, since they offer the possibility to schedule activities and preferences (cf. Chapter 5).

In Listing 2.8 we exemplified an instantiation of `time:Interval`, called `:sampleInterval`. In this example the defined interval has a `time:hasDurationDescription` and a designated beginning and end time of type `time:Instant`. Using both, a `time:hasDurationDescription` to define the actual duration of a specific task and `time:hasBeginning` & `time:hasEnd` to define to lower and upper bound for the execution of this task, allows the business logic using this ontology to schedule the actual task within bounds on its own behalf.

Whereas `:sampleDuration` is a `time:DurationDescription` of length 1 hour and 30 minutes, `:Monday_1000` and `:Monday_1200` are instantiations of type `time:Instant` and are referring to their respective `time:DateTimeDescription`.

Listing 2.8: Time interval definition with OWL-Time

```
@prefix time: <http://www.w3.org/2006/time#> .
@prefix : <https://www.auto.tuwien.ac.at/.../ActorOnt.owl#> .

:sampleInterval rdf:type time:Interval ;
    time:hasDurationDescription :sampleDuration ;
    time:hasBeginning :Monday_1000 ;
    time:hasEnd :Monday_1200 .

:sampleDuration rdf:type time:DurationDescription ;
    time:hours 1.0 .

:Monday_1000 rdf:type time:Instant ;
    time:inDateTime :DateTimeMonday_1000 .

:Monday_1200 rdf:type time:Instant ;
    time:inDateTime :DateTimeMonday_1200 .

:DateTimeMonday_1000 rdf:type time:DateTimeDescription ;
    time:minute "00"^^xsd:nonNegativeInteger ;
    time:hour "10"^^xsd:nonNegativeInteger ;
    time:dayOfWeek time:Monday ;
    time:unitType time:unitMinute .

:DateTimeMonday_1200 rdf:type time:DateTimeDescription ;
    time:minute "00"^^xsd:nonNegativeInteger ;
    time:hour "12"^^xsd:nonNegativeInteger ;
    time:dayOfWeek time:Monday ;
    time:unitType time:unitMinute .
```

2.6.2 Ontology of Units of Measure and Related Concepts (OM)

The Ontology of Units of Measure and Related Concepts (OM) [79] offers the possibility to easily define different types of units of measurement. Furthermore, it allows the definition of quantities, measurement scales and dimensions, providing a powerful way to access and combine different types of units.

In contrast to the OWL-Time ontology, we didn't integrate the whole OM ontology in order to be able to define units of measurement, but referred to their concepts using an appropriate URI. This decision was primarily based on the fact, that (*i*) we needed only a small subset of the actual OM ontology (i.e. instantiations of `om:unit_of_measure`), and (*ii*) the original OM ontology is up to seven times larger than the actual *Actor Preferences Ontology*.

In Listing 2.9 a sample temperature preference is defined, referring to the `om:degree_Celsius` definition in the OM ontology.

Listing 2.9: Preference value definition in degree Celsius(°C)

```
@prefix om: <http://www.wurvoc.org/vocabularies/om-1.6#> .
@prefix : <https://www.auto.tuwien.ac.at/.../ActorOnt.owl#> .

:sampleTemperaturePrefValue rdf:type :ContinuousPreferenceValue;
    :hasValue "21.5"^^xsd:float ;
    :hasUnitOfMeasure om:degree_Celsius .
```

CHAPTER 3

Smart Homes, Energy Efficiency and User Comfort

Contents

In the present chapter, we will discuss related work in the fields of home automation, smart homes, and ontologies representing actors and their preference information (cf. Section 3.6), introduce the *ThinkHome* system (cf. Section 3.2), discuss the topics of energy efficiency and user comfort within *ThinkHome* and in combination with the *Actor Preferences Ontology* (cf. Section 3.3 & Section 3.4), and finally investigate some of the benefits of using an ontology-driven way of persisting data (cf. Section 3.5).

3.1 Smart Homes - A Definition

Over the last couple of years, the idea of a smart home which can support its residents in their daily lives whilst at the same time decrease living costs has gained more and more popularity. There exist several slightly different definitions of a smart home but in general they can be perceived as a building that has some sort of intelligent control system which can be controlled by the residents, interacts with appliances of the smart home and monitors the home environment with sensors. A very prominent example is monitoring of the refrigerator and its content, and if some of its supplies run short the smart home would initiate their rebuy. Other examples could include the control over several types of household appliances, fully automatic realization of heating, ventilation, and air condition (HVAC) [36] system values and many more.

In [50] the authors define a smart home as:

> *A dwelling incorporating a communications network that connects the key electrical appliances and services, and allows them to be remotely controlled, monitored or accessed.*

which needs 3 parts to make it smart, namely: an *Internal Network* consisting of all the hardware which is necessary to connect appliances and equipment of the building, an *Intelligent Control* which works as a gateway to manage the system, and a *Home Automation* for working with equipment within the house and linking to external systems and services.

They furthermore identify six main areas of appliances and services a smart home should cover as depicted in Figure 3.1, two of whom will play an important role within the present thesis (i.e. the *Actor Preferences Ontology*), namely: *Environment* and *Domestic Appliances*.

3.1.1 Related Ontology-Based Smart Home Concepts

Obviously, the notion of an ontology-based smart home system is neither a new nor unique one and was already topic of several research projects over the last couple of years. In [19] the authors propose a context-aware system using OWL ontologies to represent their knowledge base, in [6] a multi-agent system is discussed which focuses on user context and behavior and utilizes ontology alignments to describe mappings to its environment, and in [81] the authors propose a more general approach by developing a requirements ontology that models the design process of home automation systems.

In general, most of these smart home concepts have their focus on either the definition of a comprehensive knowledge base to model user context or on the usage of a multi-agent system but rarely try to combine them both to exploit each others benefits (i.e. ground actions of agents on the knowledge and information a knowledge base

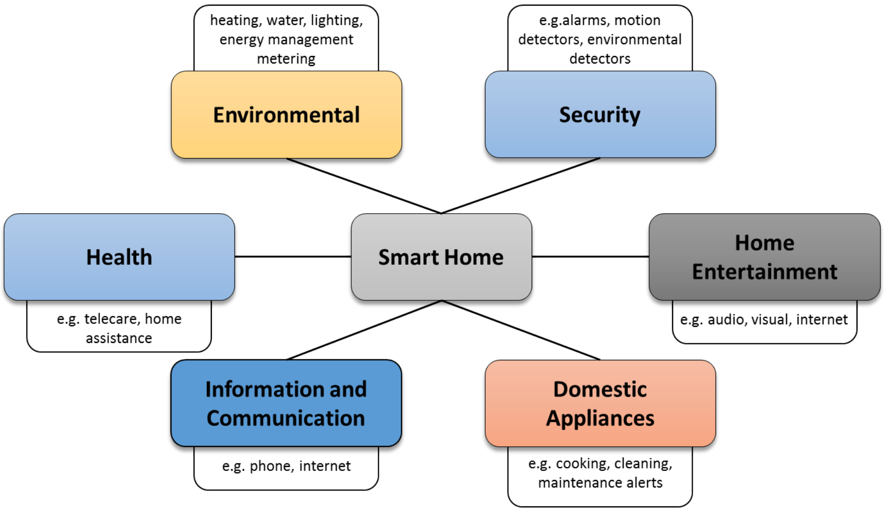

Figure 3.1: 6 Main areas served by smart home systems according to [50]

can provide). The *ThinkHome* system (cf. Section 3.2 for a more detailed introduction) aims to utilize this synergy by grounding its smart home concept on a comprehensive ontology-based knowledge base and an intelligent multi-agent system.

3.2 The ThinkHome System

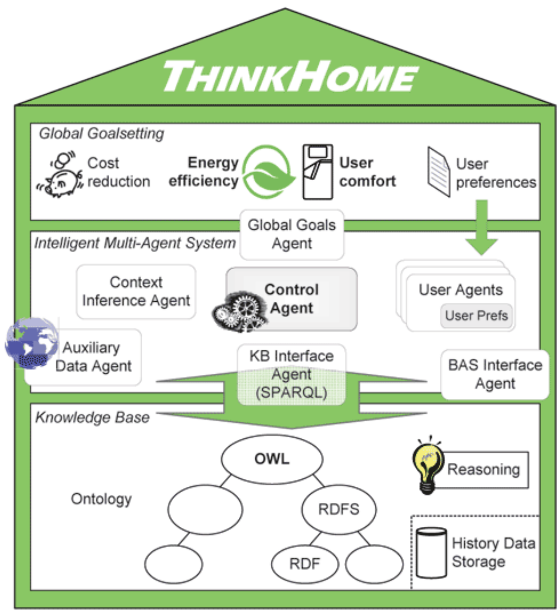

Figure 3.2: Overview of the ThinkHome system [77]

The *ThinkHome* system [77,78] is an ontology-based smart home approach, which leverages the combination of semantic web technologies and a multi-agent system in home automation. Main goals of the *ThinkHome* system are the reduction of energy consumption and the increase of user comfort of residents, which can easily be achieved due to the flexibility and expressiveness this combined approach comes with. The *ThinkHome* system is divided into two main parts, namely: *(i) an intelligent multi-agent system (MAS)*, and *(ii) a knowledge base (KB)*. While the KB is primarily responsible of storing and persisting data dedicated to dynamic and static information about the smart home itself, the MAS utilizes this information and especially the additional advantages an ontology-based knowledge base comes with (cf. Section 3.5) to be able to take actions or predict possible future behavior of residents within the smart home environment.

Due to the previous mentioned fact that the additional semantics an ontology-based knowledge base comes with can be used for rapid and flexible adaption to changes, makes this approach superior to others, which was extensively studied and investigated in previous publications [48,53–55,77,78,96–99,101].

In the following, we will give an introduction into these two components, i.e. a discussion about the multi-agent system in Section 3.2.1 and about the comprehensive knowledge base in Section 3.2.2, the *Actor Preferences Ontology* will be part of.

3.2.1 Intelligent Multi-Agent System (MAS)

Multi-agent systems as paradigm for distributed artificial intelligence as introduced in [22] are responsible for achieving the two primary goals *(i) energy efficiency*, and *(ii) user comfort* within the scope of *ThinkHome*. For that purpose, agents are able to interfere with the KB as well as the underlying home automation systems and various available data-sources to be able to perform actions based on the information provided. Such agents can e.g. realize certain user preferences, re-schedule tasks and preferences to provide a more energy preserving environment if possible, or even predict possible preferences or actions a resident of the smart home might have or wants to perform [98].

The MAS of the *ThinkHome* system distinguishes eight different types of agents (cf. Figure 3.3), namely [76]:

User Agent - The user agent is acting on behalf of users and primarily responsible for proving user comfort. This can be achieved by reducing interactions with control systems a resident might have to do in order to realize certain conditions (e.g. temperature, lighting level, etc.) by learning user habits and by being aware of occupancy status. Additionally, it provides the interface for users to store their preference profiles, general preferences, preferences for activities, and/or schedules for preferences and activities. This agent is the one using the information the *Actor Preferences Ontology* provides.

Room Agent - The room agent controls all building services and processes which are located within its associated zone. Every room/zone has therefore its own room agent which locally influences its environment and which is responsible for executing the intelligent control strategies.

Automation System Agent - The automation systems agent collects data from sensors which are installed within the smart home and routes all communication to and from automation devices, e.g. obtaining information requested by the agent system from any automation device.

Data Management Agent - The data management agent primarily observes and provides information about the whole smart home environment and is aware of a variety of information on the world which taken together forms the world state. Based on these information, other agents are able to execute their tasks such as control strategies.

Global Management Agent - The global management agent is responsible for handling all tasks which require a complete view of the whole smart home ecosystem. Especially, safety & security tasks, as well as global energy efficiency strategies are taken out by this agent. Additionally, it serves as sole interface to external services which want to interfere with the smart home system such as the smart grid, demand side management or performance contracting.

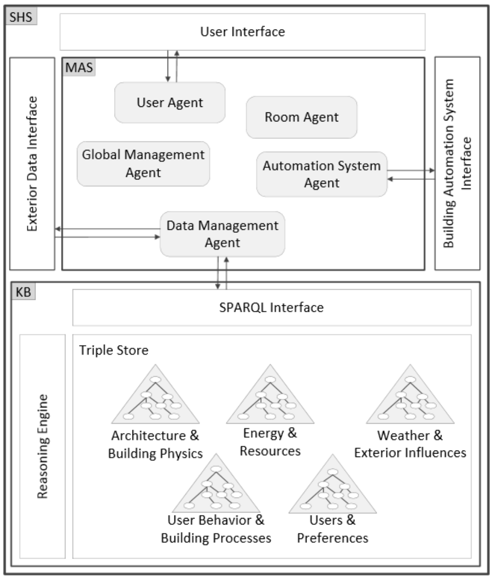

Figure 3.3: Smart Home System Components [52]

3.2.2 Comprehensive Knowledge Base (KB)

The knowledge base of the *ThinkHome* system is represented by a highly interwoven set of ontologies, where each ontology is responsible for representing a particular part of the smart home ecosystem. By choosing Semantic Web technologies and particularly ontologies as main approach to model the knowledge base, the `ThinkHome` system is able to exploit the additional semantic information such an infrastructure comes with. Most of the tasks the previous introduced agents perform would not be executable without the help of reasoning and inferencing capabilities an ontology-based knowledge base offers.

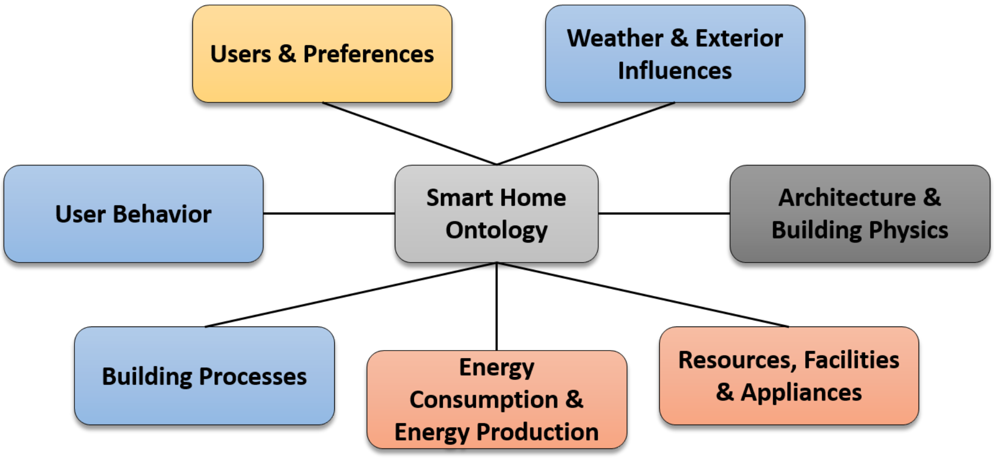

Figure 3.4: Overview about the ontologies within the *ThinkHome* system and their field of application (same color indicates same ontology affiliation).

As depicted in Figure 3.4, the previously introduced KB stores information (i.e. defines concepts, their attributes, and relationships among them) about five different domains of the `ThinkHome` system, namely [52]:

Weather & Exterior Influences - Weather and climate information can be exploited to infer certain actions that are necessary to realize preferences in the most energy-efficient way possible.

Architecture & Building Physics - Information about the architecture and physics of the building is of major importance to the smart home system, but often rather extensive in its representation. Storing such information and therefore releasing the users of the burden to enter that information by themselves, contributes to a user friendly smart home ecosystem.

Resources, Facilities, Appliances, Energy Consumption & Energy Production - Energy and resource information can e.g. be utilized as decision support for agents to find the momentarily best option for energy efficient energy consumption.

User Behavior & Building Processes - User behavior is represented as (derived) habit profiles and patterns of actors as well as possible predicted schedules for different occupancy states of the smart home, whilst at the same time a building process describes a concept that contains information about elementary operations within the building.

Users & Preferences - Since information of users of the smart home system is at least as important as the other already introduced types of information, this part focuses on the representation of human or system actors of the smart home as well as the representation of the preferences they might have.

Since some of the aforementioned domains of the *ThinkHome* system are directly related to the *Actor Preferences Ontology*, we will now give a brief description of those ontologies that are used to model them.

Energy & Resources Ontology (ero)

Figure 3.5: The *Energy & Resource Ontology*.

The *Energy & Resources Ontology* (cf. Figure 3.5) is used to describe two main areas important to smart home systems, *(i) energy information*, and *(ii) resource information*.

While energy information comes into play when integrating the *ThinkHome* system into a smart grid, where this information can be used to provide e.g. the momentarily best option for energy consumption [77], the resource information represents all equipment available in the smart home. The *Actor Preferences Ontology* refers to two concepts of the *Energy & Resources Ontology*, namely: `Appliances` such as blinds, lamps, dishwashers, etc. for which an `Actor` can define `Preferences` and `States` which can be assigned to `PreferenceValues` and represent different state values a certain `Appliance` might have (e.g. On/Off, High/Medium/Low). A `Preference` which was defined for a certain `Appliance` refers to it via its `controlsAppliance` property and is then assumed to be an `ApplianceCentricPreference`.

User Behavior & Building Processes Ontology (ppo)

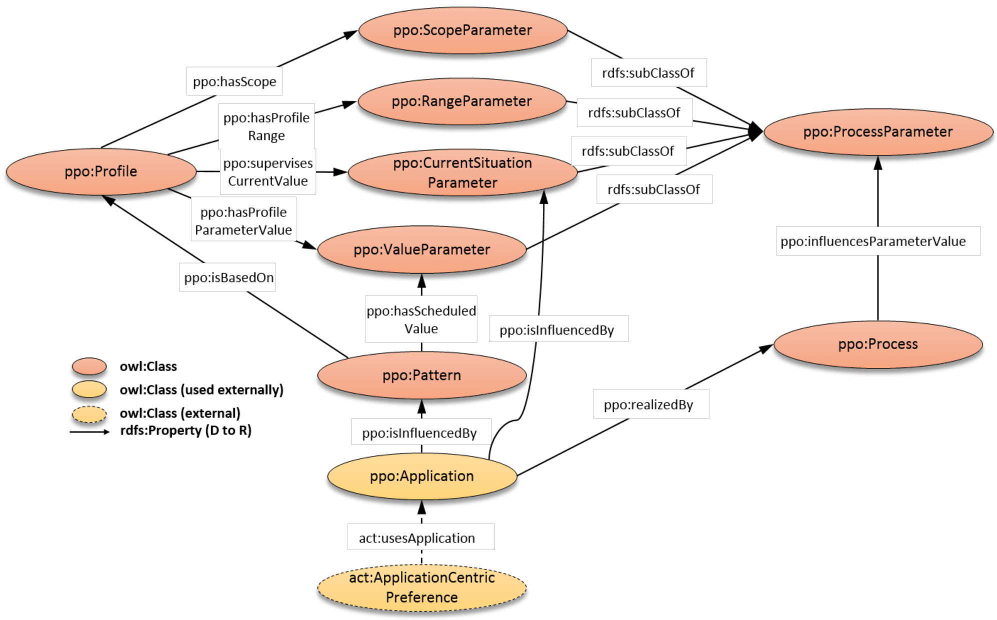

Figure 3.6: The *Process & Profiles Ontology*.

The *User Behavior & Building Processes Ontology* is on the one hand responsible for storing habit profiles and patterns of residents of the smart home, which enables the possibility to manage predicted future behavior of mentioned users and on the other hand to model and store building processes called `Applications` which contain a set of actions and operations that can be taken out in the building. Especially the latter concept of `Applications` is important for the *Actor Preferences Ontology* as shown in Figure 3.6. Certain `Preferences` might require the use of `Applications` in order to be able to be realized, which is indicated by the `usesApplication` property. Such a `Preference` is defined as `ApplicationCentricPreference`.

Architecture & Building Physics Ontology (gbo)

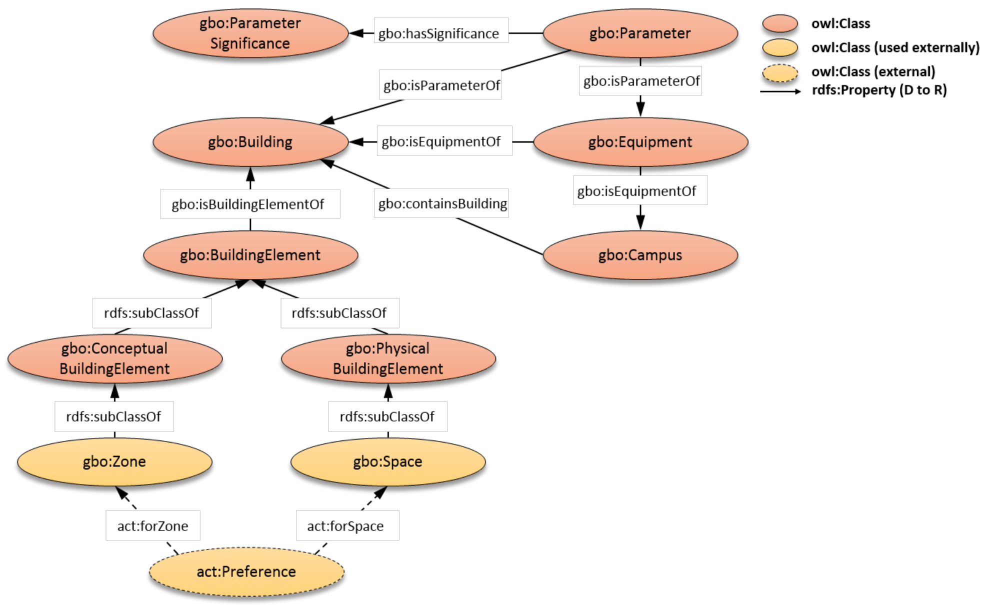

Figure 3.7: The *Building Information Ontology*.

Main purpose of the *Architecture & Building Physics Ontology* (cf. Figure 3.7) is the storage of building information. Storing this information is of utter importance for the smart home system as it can e.g. be used to define `Preferences` that should only be valid in certain areas of the building. For this purpose, two concepts which describe areas of the building, namely `Zones` from a conceptual and `Spaces` from a physical point of view, were referred to by the *Actor Preferences Ontology* via the properties `forZone` and `forSpace`, respectively.

3.3 Energy Efficiency in *ThinkHome*

As first of the two main goals the *ThinkHome* project aims to achieve, *Energy Efficiency* is the one having the largest improvement opportunities among other smart home systems. While most of those systems use different kinds of strategies to provide a certain degree of energy efficiency, these strategies often do not take both interior and exterior conditions into account or only have simple realization strategies for e.g. achieving a certain temperature level in this case, instead of using the air conditioner to lower the room temperature, a smart home system could take exterior influences such as the current weather state into account and just open the windows to achieve the same goal but in a way more energy efficient manner. Other approaches have shown, that it is possible to foresee potential behaviour of smart home residents by deriving their behavioural profiles based on patterns and thus, be able to perform energy preserving actions before incidents occur [95].

Moreover, the usually large amount of appliances makes it necessary to keep track of their energy consumption and to schedule their execution taking the current energy supply situation into account [52].

In the following, we will discuss some scenarios in which the *Actor Preferences Ontology* supports one of the major goals of the *ThinkHome* system - *Energy Efficiency* - in more detail.

Schedulable Preferences

Figure 3.8: A potential washing machine job can be re-scheduled within its time frame to compensate high energy consumption peaks.

By assigning time information to preferences their realization can be planned in an energy efficient manner. In the example illustrated in Figure 3.8, a user wants his washing machine job, which takes one hour to be finished, to be executed during 12am - 6pm. Taking the overall energy consumption of the smart home system into consideration, a control agent could start this job at any times with low energy consumption peaks within its time frame if it still finishes before 6pm. Other benefits arise in combination with the later discussed occupancy of the smart home. If a user e.g. states that he wants a particular room to have 20°C when arrives back home at 5pm, then a control agent does not have to advise an air conditioner to maintain this temperature the whole time till 5pm but can plan actions to reach this preference value in a timely and energy preserving manner (e.g. open the windows two hours prior to the scheduled time if the current weather state is suitable) .

Occupancy Detection

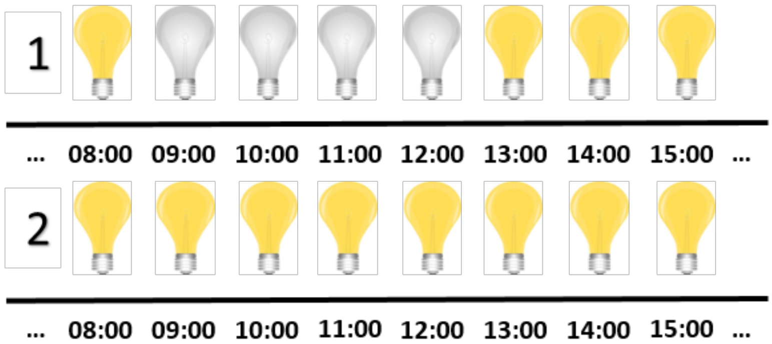

Figure 3.9: Low (1) vs. high (2) energy consumption due to available (1) or missing (2) occupancy detection.

Occupancy detection is a major influence factor of efficient energy management and contributes to unnecessary energy consumption if missing as exemplified in Figure 3.9 and as reported in [8]. In this example, a potential smart home is unoccupied from 9am to 12am and therefore does not impose the necessity of having any turned on lamps during that time period (unless it was stated otherwise). So there sometimes might occur a situation where a resident of the smart home forgets to turn off the lights before he leaves the house which leads to a waste of energy (Situation (2)). With occupancy detection on the other hand, either automatically sensed or statically persisted in `PresenceSchedules` as `PresencePreferences`, the smart home system would be able to start energy preserving actions such as lowering the temperature or as in the present example turning the lights off (Situation (1)).

3.4 User Comfort in *ThinkHome*

User Comfort as second major goal of *ThinkHome* represents the need for an integrated and predictive system environment that works on behalf of the residents and automatically realizes comfortable living conditions, whilst at the same time offering enough possibilities to interfere with the system and manually adjust settings if necessary [52].

In the following, we will discuss two sub goals of *User Comfort* within ThinkHome in more detail.

Improved and Simplified User Interaction And improved and simplified interaction with the system is essential to achieve a certain level of user satisfaction. If e.g. the entering of user preferences is too complicated residents of the smart home might get upset and frustrated and either limit the amount of interaction with the system to a minimum or even stop it at all. Additionally, it is of major importance to offer users insight into current energy consumption levels in a way that they might rethink their habits based on that feedback and therefore maybe adjust their lifestyle into a more energy efficient one [46].

The *Actor Preferences Ontology* with its main focus on storing and persisting preferences of *ThinkHome* actors builds the foundation for that sub goal. By offering a simple but at the same time expressive way to persist information about actors, their preferences, and activities, the *ThinkHome* system can increase the living comfort of its users.

Automatic Realization of Preference Processes It is a tedious task for residents to take care of the realization of comfortable living conditions by themselves, especially if they aim to do that in an energy efficient way. The *ThinkHome* system relieves its users from that burden by offering them a simple and convenient way to store their preferences about living conditions and by realizing them automatically in the most energy efficient way possible, which represents a significant additional value to non-automated homes as their residents would have to take care of that realization by themselves.

3.5 Benefits of using Ontologies as Datastores

In contrast to using common relational databases for storing data about a system, ontologies offer a large variety of benefits especially regarding intelligent data management, which makes them superior to the mentioned relational approaches. By assigning semantic information to entities of the ontology, making the data accessible through an inference system and thus offering the possibility to e.g. deduct implicit knowledge, these benefits can be utilized which is often referred to as *Ontology Based Data-Access (ODBA)* [102].

One drawback, ODBA in general and ontology driven datastores - called triplestores - suffered from, was the misconception that they are immature compared to other widely used relational database management systems (RDMS). But systems like Sesame [12] or AllegroGraph [2] which become more and more popular in terms of their usage have proven to be a sophisticated alternative particularly for the emerging trend of *Big Data scenarios* [80].

In the following, we will discuss three major benefits of using ODBA in more detail.

3.5.1 Automatic Type Inference

The first advantage we will discuss is the possibility to automatically infer the type of individuals of concepts based on their characteristics. Assuming a thoroughly defined ontology, OWL reasoners like Pellet [71, 84] or HermiT [42, 82] (cf. Section 6.1 for a more detailed introduction) are able to perform this task, thus, facilitating data management and integration whilst at the same time avoiding probable errors originating from manual assignments.

We have exemplified such a scenario in Figure 3.10. Two individuals, `InstanceAge1` and `InstanceAge2` are defined as individuals of type `Age`, while the concept `Age` consists of one sub concept `HumanActorAge` which is defined as concept having a related `hasYears` property. `HumanActorAge` itself can be distinguished into `YoungHumanActorAge` whose `hasYears` value is below 14 and `AdvancedHumanActorAge` whose `hasYears` value is over 65. These associated `owl:equivalentClass` axioms allow the assertion of every individual which fulfills the stated constraints to the concept the respective axiom was defined for.

In the example illustrated in Figure 3.10, a reasoner is able to infer that `InstanceAge2` is additionally of type `YoungHumanActorAge` and `InstanceAge1` of type `AdvancedHumanActorAge`.

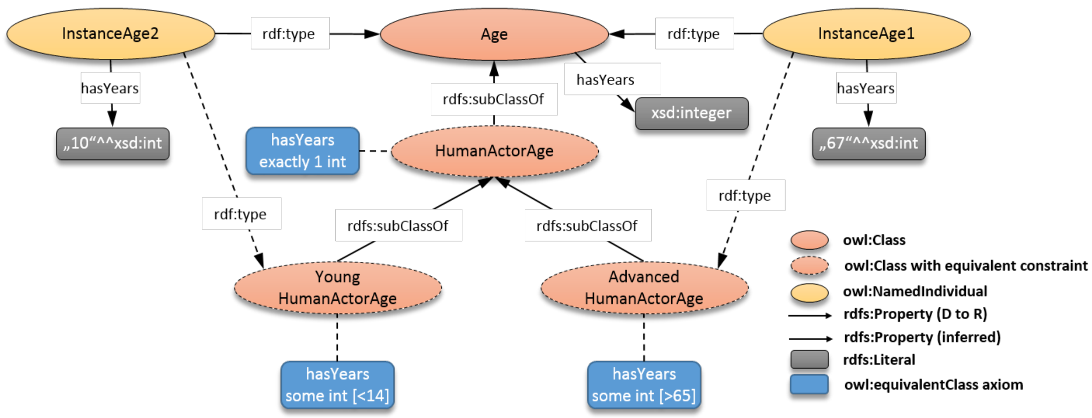

Figure 3.10: Automatic type inference of individuals of concept `Age`.

3.5.2 Extensive Reasoning Support

Since ontologies were all along built to manage and use semantic information of the data they contain, reasoners which are able to utilize that semantic information have always been an important part of OBDA. Apart from the task of inferring and deducting new knowledge, reasoners can furthermore be used to detect inconsistencies and unsatisfiable individuals within the ontology as depicted in Figure 3.11. In that example, two concepts `YoungHumanActorAge` and `AdvancedHumanActorAge` are defined as being disjoint from each other (i.e. an individual cannot be of both types at the same time) and a new individual of type `Age` called `InstanceAge3` is introduced having two values assigned via its `hasYears` property. Since one value each now fits exactly to one `owl:equivalentClass` axiom, a reasoner will declare `InstanceAge3` as being both, of type `YoungHumanActorAge` and `AdvancedHumanActorAge`, which cannot be the case based on the previous stated disjointness of both concepts. Thus, a reasoner will throw an error and defines the ontology to be *inconsistent*, based on the wrongly defined individual `InstanceAge3`[1].

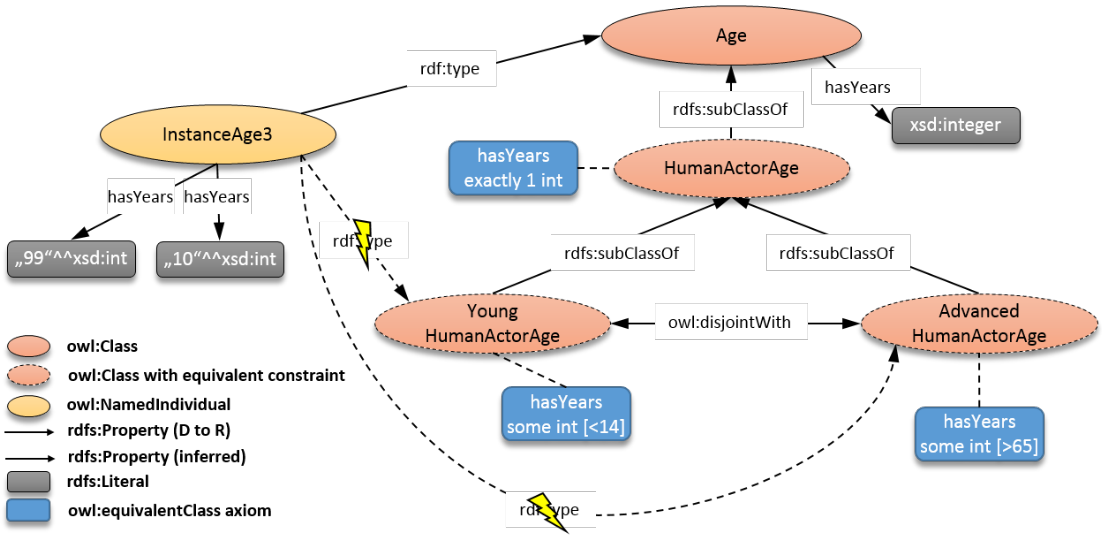

Figure 3.11: Detection of wrongly defined individual `InstanceAge3`.

The example in Figure 3.11 represents only one case for which a reasoner can detect errors within the ontology. Other common reasoning tasks include the detection of *unsatisfiability of concepts* (i.e. these concepts are defined in a way that there cannot exist any individuals of them), declaring the ontology to be *incoherent* (i.e. it contains unsatisfiable concepts which are not instantiated and there exist others that are satisfiable) or declaring the ontology to be *inconsistent* if there exists no model of the ontology which makes all axioms hold.

3.5.3 Reasoning under Closed and Open World Assumption

One of the major benefits of SWTs for integration scenarios are their well defined semantics and the extensive reasoner support as already discussed previously. With OWL and OWL reasoners it is e.g. possible to describe cardinality constraints, perform automatic type inferencing as discussed above and to check for inconsistency in the given ontologies.

[1]Readers which might be already familiar with Semantic Web technologies probably have noticed that this behavior could also have been realized by defining the property `hasYears` as functional property.

While Semantic Web languages underlie the *Open World Assumption (OWA)* (i.e., if a statement is not explicitly stated, it does not mean that it does not exist), software engineering languages are mostly based on *Closed World Assumption (CWA)* (i.e., if a statement is not present, it does not exist) [75]. To deal with this issue, constraints expressed as OWL axioms and which cannot be checked by common OWL reasoners due to the OWA can be checked via SPARQL queries.

With SPARQL, it is possible to query for the presence of individuals which violate those constraints and thus, detect inconsistencies.

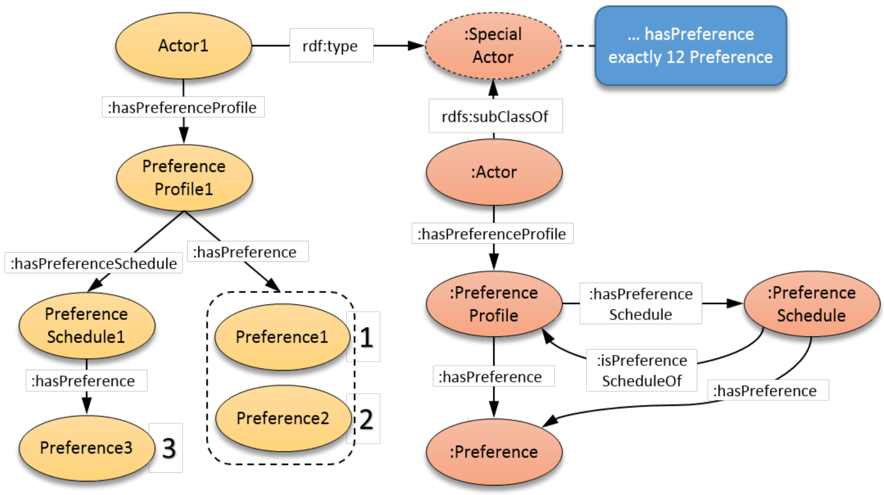

Figure 3.12: Detection of violated cardinality constraints.

Consider the example illustrated in Figure 3.12 and especially the cardinality constraint `...hasPreference exactly 12 Preference` for concept `SpecialActor` expressed in *OWL Manchester Syntax*[2] (this cardinality constraint shall only represent an abbreviation). The answer to the question whether or not a particular `Actor` has the right amount of `Preferences`, would not be directly decidable for OWA, because there might be some preferences which are not already identified, but is decidable for CWA with the support of SPARQL as depicted in Listing 3.1.

Listing 3.1: ASK Query which returns `true` if an `Actor` has not exactly 12 `Preferences`

```
ASK WHERE {
 { SELECT (count(?preferences) AS ?numbOfPrefs) ?actor WHERE {
     ?actor a :Actor .
     ?actor :hasPreferenceProfile/:hasPreferenceSchedule?
                                 /:hasPreference ?preferences.
   } GROUP BY ?actor
 } FILTER(?numbOfPrefs != 12) }
```

[2] http://www.w3.org/TR/owl2-manchester-syntax/

3.6 Related Work

While reusing ontologies for its own purpose is highly encouraged by numerous ontology development approaches (cf. Chapter 4), their reusability highly depends on their intended purpose. Although ontologies that model domains like time information [39] or units of measurement [79] can in general be used out of the box, those having a more precise and narrow field of application also often have a different focus than the one intended. Moreover, the available ontologies that have a similar focus as the *Actor Preferences Ontology* mostly specialize themselves on representing users and their context within a smart home, rather than considering user preferences/activities and the scheduling of preferences/activities with the same importance. Developing the ontology by ourselves additionally comes with the benefit of being in charge of the creation of documentation artefacts which we ascribe high importance to.

In the following, we will list some of those mentioned ontologies where some of them served as inspiration for the conceptualization of the *Actor Preferences Ontology*:

- In [67, 91], the authors present an ontology which models the relationship between objects in the environment and human intentions. The main field of application of the presented ontology lies in the support of elderly people and the interaction with service robots. They discuss an approach of retrieving human preferences with reinforcement learning having a major focus on preferences regarding actions with household equipment (e.g. opening a lunch box, taking a pen).

- Residing in a completely different domain, the authors of [49] describe the construction of a user preference ontology for anti-spam mail systems. Although, as already mentioned that ontology was modeled for a completely different domain, the principle of using persisted preference information only in a semi-automatic manner, i.e. keeping human actors in charge and offer them the possibility of interfere with the system at any time, will be reused within the *Actor Preference Ontology*.

Apart from approaches that use ontologies for modeling users and their preferences for an arbitrary domain, there also exist some work on using those ontologies within the scope of smart home concepts:

- In [35], the authors describe an context model to represent, manipulate and access context information in intelligent environments and reasoning capabilities such a context model provides, having a focus on persons, locations and activities but do not take user preferences into account.

- One article that describes an approach which is quite similar to ours can be found in [58]. There, the authors use user profiles to store general preferences about the environment and activity profiles to represent preferences which should be valid whenever their asserted activity will be performed.

- In [94], an ontology for persisting user context which stores data about environment, activity, preference information is discussed. They additionally present an approach to exclude duplicated information which results in a 70 times smaller amount of data that has to be persisted.
- Finally, in [19], an ontology that is able of storing user information, instantiations and rules, which can be defined, is presented. The context-aware model the authors describe can be used within a pervasive environment in a way that control services are able to perform decisions and actions on behalf of the user they represent. Unfortunately, the exact structure (i.e. used concepts, properties, etc.) of the ontology is missing.

CHAPTER 4

Ontology Engineering

Contents

Ontologies have become an important component in many areas including *information retrieval and extraction*, *knowledge management*, *ontology-based data access* [72] and are part of a new approach for building intelligent information systems [21, 83]. As a result there exist a variety of different ontology engineering approaches and ontology development methodologies, all having different advantages and disadvantages, which makes it difficult to choose the most suitable approach for the problem at hand.

In the present chapter, we will focus on discussing different strategies for conducting the process of developing an ontology, and introduce selected ontology development methodologies in more detail by explaining their internal workflow and analyzing their advantages and disadvantages, rather than investigating general methodologies for defining an ontology from scratch as discussed in [87]. We then emphasize our decision for using METHONTOLOGY [23] for creating the *Actor Preferences Ontology*, by introducing its underlying workflow and methodology in more detail.

4.1 Ontology Design Patterns (ODP)

Originating from the field of software engineering, the authors of [25] proposed the reuse of small, task-oriented ontologies called *Ontology Design Patterns (ODP)* as strategy to solve specific types of design and implementation issues by using common solutions (cf. a comprehensive collection of common ODPs [27]). Such an ontology design pattern serves as possible solution to a recurrent ontology design problem and can be differentiated into six different ODP families [26] (cf. Figure 4.1).

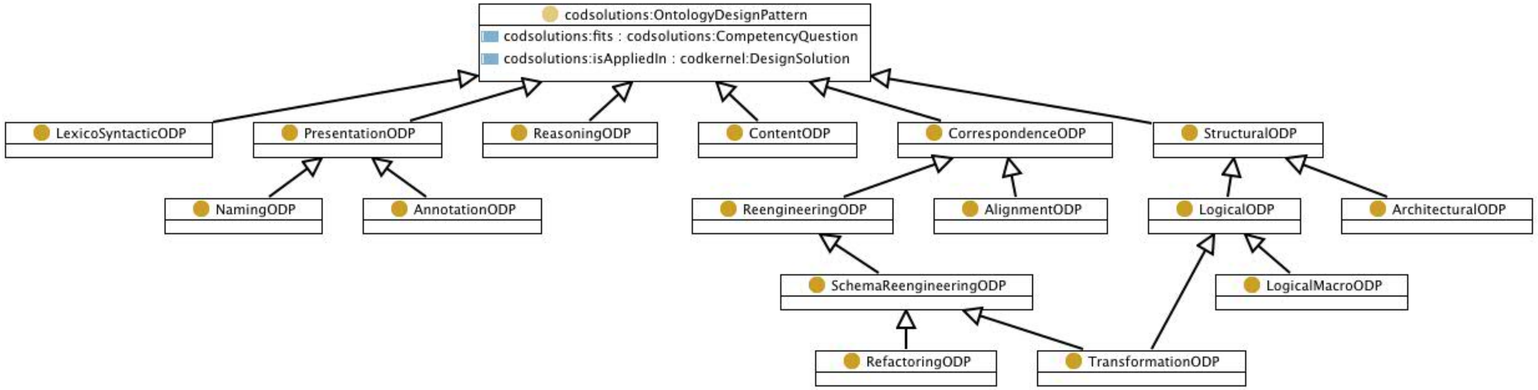

Figure 4.1: Different types of *Ontology Design Patterns* [26].

We will now briefly introduce the distinct types of ODPs based on their hierarchy, shown in Figure 4.1 and their definition given in [25].

Structural ODPs are divided into *Logical ODPs* and *Architectural ODPs*.

Logical ODPs focus on solving issues of expressivity, which may be caused by limitations of the chosen representation language, by composing logical constructs to overcome those problems.

Example: An example of such a *Logical ODP* is the *n-ary relation pattern* [68] (cf. Figure 4.2), which defines a best practice to model n-ary relationships between concepts in OWL, which natively supports only bidirectional relations between concepts. This can be achieved by introducing a new concept, representing the relationship between the *n* concepts and additional relations from/to them.

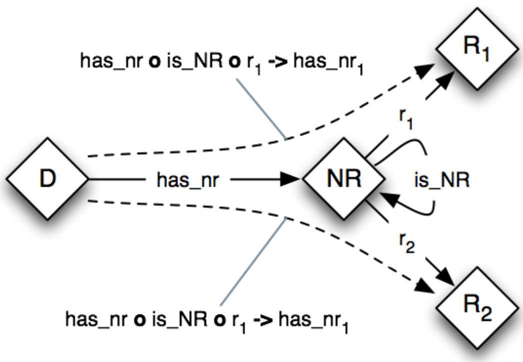

Figure 4.2: Representation of a n-ary relationship in OWL [68].

Architectural ODPs constrain the overall shape of the ontology. They can be either *(i) internally* - defining a set of *Logical ODPs* that have to be used when developing an ontology or *(ii) externally* - defining meta-level constructs which should build the overall structure of the ontology. *Architectural ODPs* usually serve as reference documentation for further ontology design steps.

Correspondence ODPs are further divided into *Reengineering ODPs* and *Mapping ODPs*. While *Reengineering ODPs* offer solutions to transform conceptual models, which may have their origin in other resources than ontologies, into ontologies, *Mapping ODPs* provide best practices to map concepts of two existing ontologies with each other (i.e. creating semantic aligments).

Content ODPs are focussed on encoding conceptual design patterns and on solving design problems for specific domains. In contrast to *Logical ODPs* which define patterns for problems independent of a specific application domain.

Example: A typical *Content ODP* is the `hasPart/isPartOf` relationship among entities and their parts (cf. Listing 4.1). This pattern provides an example to model such a conceptual relationship and can directly be used in the specific part of the ontology, dealing with that issue.

Listing 4.1: *Ontology Design Pattern* for representing entities and their parts (taken from [27])

```
...
:hasPart rdf:type owl:ObjectProperty ,
                  owl:TransitiveProperty ;
         rdfs:label "has␣part"@en ;
         owl:inverseOf :isPartOf ;
         rdfs:range owl:Thing ;
         rdfs:domain owl:Thing .

:isPartOf rdf:type owl:ObjectProperty ,
                   owl:TransitiveProperty ;
          rdfs:label "is␣part␣of"@en ;
          rdfs:domain owl:Thing ;
          rdfs:range owl:Thing .
...
```

Reasoning ODPs enable ontology engineers to model *Logical ODPs* in a way that desired reasoning results are obtainable by certain reasoners. Such patterns can include examples for *classification, inheritance, subsumption, . . .* and can be modeled in a domain independent way (i.e. not limited to certain application domains).

Presentation ODPs are in contrast to previous introduced DPs concerned with the usability and readability of ontologies from a user perspective. While *Naming ODPs* define best practices to create names for entities of an ontology, namespaces, and files, *Annotation ODPs* provide conventions for annotating entities in order to improve the understandability of ontologies and their elements.

Lexico-Syntactic ODPs are linguistic structures that consist of certain types of words following a specific pattern (similar to naming patterns proposed for business process activities in [59]), and allow to derive some conclusions about the intended semantics/meaning of entities within an ontology.

4.2 Ontology Development Methodologies

In contrast to the previously introduced ontology design approach of *Ontology Design Patterns* 4.1, the area of specific *Ontology Development Methodologies* differs primarily in its intended purpose. While ODPs propose patterns to model certain aspects of an ontology, ontology development methodologies define guidelines for conducting the entire development process. In the following section, we will introduce some of the most common and well established methodologies, by discussing their internal workflow and analyzing their advantages and disadvantages.

4.2.1 Methodology by Uschold & King

The authors of [92] were one of the first to propose a skeletal methodology, which is illustrated in Figure 4.3, for developing ontologies based on guidelines and hints reported in related literature such as *(i) principles for designing ontologies [33]*, and *(ii) evaluation of knowledge sharing technology [30]*.

Workflow Description

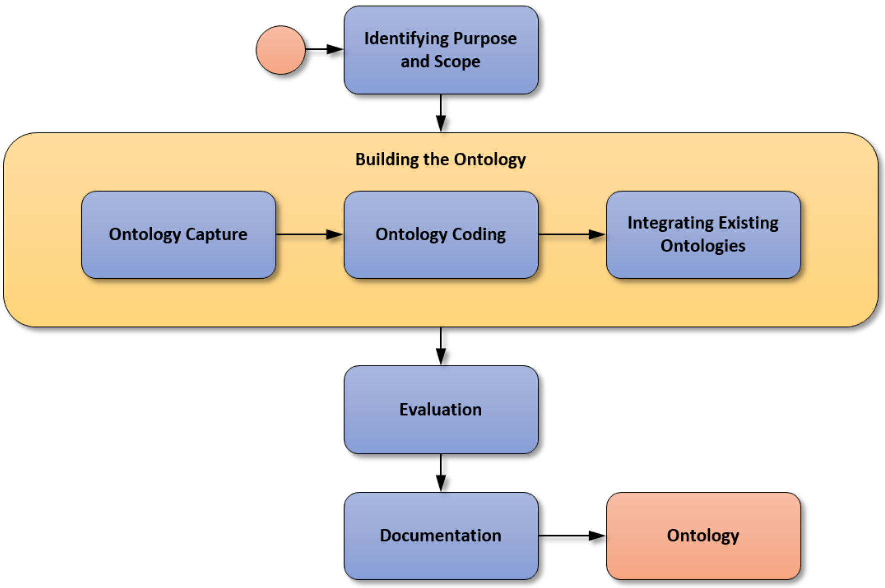

Figure 4.3: Workflow of the ontology development methodology proposed by *Uschold & King* [92].

Identifying Purpose and Scope As a first step, the actual purpose of the ontology together with its scope shall be identified (e.g. as competency questions).

Building the Ontology This step deals with the actual implementation of the ontology and can be broken down into three sub-steps, namely:

Ontology Capture The first part of building the ontology deals with the *(i) identification of key concepts and relationships*, *(ii) annotating such concepts and relationships with text definitions*, and *(iii) identification of referring terms*.

Ontology Coding After all relevant terms are captured and documented, the ontology gets serialized in a formal language which is capable of representing those elements.

Integrating Existing Ontologies This step, which can be considered to be conducted in parallel to those explained above, deals with the question if there are any existing ontologies which might be relevant for the application domain and could be integrated into the ontology.

Evaluation During the evaluation, previous defined competency questions are used to evaluate whether or not the requirements are met.

Documentation All important assumptions should be documented, especially those about main concepts of the ontology.

Although the methodology of *Uschold & King* can be considered to be outdated, its main steps served as foundation of many other development approaches proposed in the recent past (e.g. *Ontology 101* [69], *METHONTOLGY* [23]). Hence, e.g. steps similar to *Identifying Purpose* and *Ontology Capture/Coding* can be found in nearly every ontology building approach.

Analysis

Advantages: The approach proposed by *Uschold & King* is a quite simple and straightforward methodology for creating ontologies. It only consists of few steps to be considered while developing an ontology and therefore deemed to be suitable for small-scale ontologies and non-productive ones.

Disadvantages: Due to the fact that this methodology describes a general workflow to develop ontologies rather than providing precise tasks that must be conducted in order to develop an ontology, it should definitely not be used to build ontologies which need to be well documented and/or are used in an highly productive environment.

Applicability for the Actor Preferences Ontology: Although there exist some ontologies which were successfully developed following the methodology proposed by *Uschold & King* (e.g. *Enterprise Ontology*[1] and an e-Government Ontology [24]), we decided against its application based on following major shortcomings:

[1] `http://www.aiai.ed.ac.uk/project/enterprise/enterprise/ontology.html`

1. **No mandatory creation of documentation artifacts** - Besides a documentation task at the end of the development life-cycle, this methodology does not require a mandatory creation of documentation artifacts throughout the development process. For a productive environment and especially if more than one ontology engineer is involved, a precise documentation is essential, which cannot be guaranteed following this approach.
2. **Vague description of development steps** - Again, for the usage within a productive environment where standardized development steps are important in order to guarantee a fully integrateable ontology, this methodology is not suitable.

4.2.2 METHONTOLOGY

Following the basics and recommendations of *Uschold & King*, the authors of [23] proposed one of the first comprehensive methodologies for developing ontologies. In contrast to many other approaches, which either do not include documentation as step in the ontology development life cycle or enforce users to explicitly create a documentation after the completion of the ontology, the documentation in *METHONTOLOGY* is created throughout the whole development process.

Since we have chosen this approach for the development of our ontology and will describe it in more detail in Section 4.4, we will introduce the different phases of its workflow very briefly.

Workflow Description

Planification In this very first step, METHONTOLOGY proposes a plan which describes the scheduling of each task (i.e. what has to be done at which time?)

Specification During the *Specification* an *Ontology Requirement Specification Document* is defined which describes the scope of the ontology using competency questions.

Conceptualization In the *Conceptualization* step, the domain knowledge will be structured in a conceptual model which is divided into a set of diagrams and tables (cf. Figure 4.8).

Formalization Using the previous defined conceptual model, a formal representation of that model is produced.

Integration Whenever applicable, existing ontologies which are able to cover parts of the application domain shall be reused and integrated into the ontology.

Implementation During the *Implementation*, the results of the *Formalization* and *Integration* step are combined and merged into an ontology.

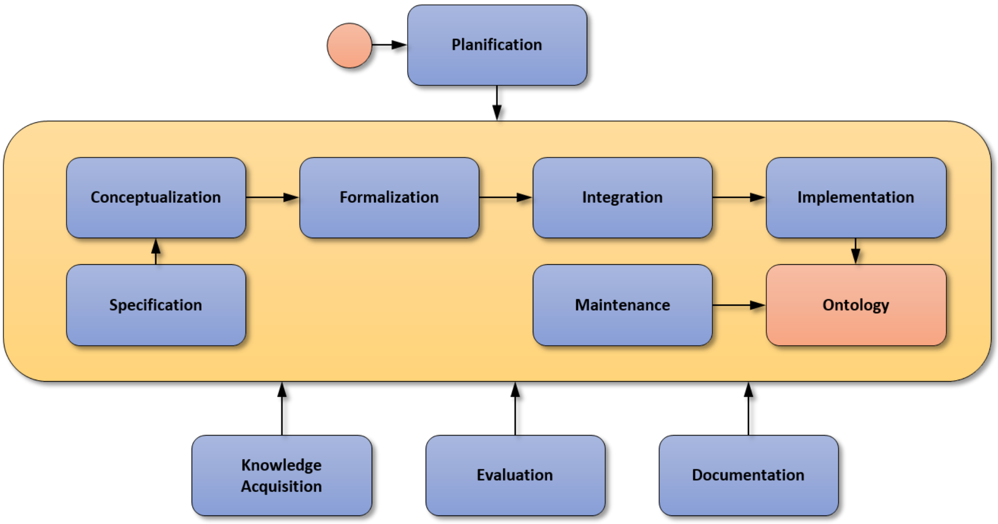

Figure 4.4: Process of creating an ontology using the METHONTOLOGY approach [23].

Maintenance After the ontology has been released, some modifications and maintenance tasks may have to be carried out.

The next three phases are not performed only once or at a specific point during the development process, but multiple times along with the previous defined phases.

Knowledge Acquisition As the name already suggests, this phase mainly covers the acquisition of knowledge about the domain of interest.

Evaluation On the basis of previous defined documents, the ontology is evaluated throughout the whole development process in order to ensure it is meeting its requirements.

Documentation After each phase, a document describing its results is created.

Analysis

Advantages: As already mentioned earlier, the creation of a comprehensive documentation is enforced during the whole development life cycle (i.e. a document for every phase of the development process describing its outcomes). Furthermore, a lot of ontologies were created following the *METHONTOLOGY* approach including a chemical ontology [62], a legal ontology [16] and a graduation screen ontology [70] (cf. [29] p. 141-142 for a more comprehensive list of ontologies).

Disadvantages: For the development of small ontologies, which will be most likely not being used in a productive environment, another more light-weight approach like *Ontology 101* might be more suitable.

Applicability for the Actor Preferences Ontology: Since *METHONTOLOGY* does perfectly fit our requirements (i.e. *(i) provide detailed instructions to perform development steps* and *(ii) enforce the creation of documentation artifacts throughout the whole development process*) having at the same time a reasonable development effort, it is perfectly applicable as development methodology for our Actor Preferences Ontology.

4.2.3 Unified Process for Ontology Building (UPON)

The authors of [18] wanted to map principles of software engineering, more precisely principles of the *Unified Process* [47], to ontology engineering and proposed the *Unified Process for Ontology Building (UPON)*, which is depicted in Figure 4.5.

Workflow Description

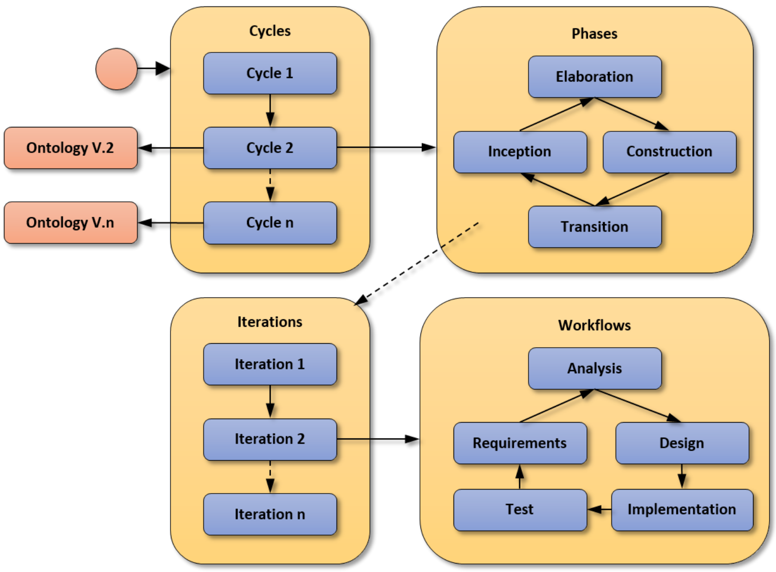

Figure 4.5: Process description of developing an ontology using the UPON approach [18].

Cycle The development process of the *UPON approach* consists of n *Cycles* where each *Cycle* results in a new version of the ontology and consists of several *Phases*.

Phases There exist four different *Phases*, namely:

- **Inception** During *Inception*, requirements are gathered.
- **Elaboration** The *Elaboration* phase includes the identification and organization of important main concepts.
- **Construction** Elements of the ontology to be built are created during the *Construction* phase.
- **Transition** The optional *Transition* phase involves the evaluation and testing of the generated ontology.

Iteration Each previous mentioned *Phase* consists of several *Iterations* by itself and each *Iteration* can be broken down into five *Workflows*. Notice that, not for every *Phase* all *Workflows* are performed, e.g. *Inception* is primarily concerned with capturing requirements and partly performing some conceptual analysis but neither performs implementation nor testing [18].

Workflow Five different *Workflows* can be distinguished within the UPON approach:

- **Requirements** As a very important first step, the requirements of the ontology, which should be developed, are investigated. Competency questions along with use cases, relevant terminology and the purpose of the ontology are defined and passed to the *Analysis.*
- **Analysis** Using the generated artifacts of the *Requirements* workflow, those requirements get refined and structured. Additionally, the reuse of existing resources is investigated based on the previous created scope definition and a first version of a glossary of concepts is built.
- **Design** The *Design* workflow incorporates the more precise refinement of elements discovered during *Analysis* and the identification of relationships among them.
- **Implementation** The formalization of the informal ontology, which was created in the previous workflow, into an ontology language takes place in this step.
- **Test** The *Test* workflow verifies the correctly implemented requirements of the ontology (i.e. by trying to answer competency questions using concepts of the ontology) together with the coverage of the ontology over its application domain.

Analysis

Advantages: The biggest advantage of building an ontology with the *UPON approach* is the exhaustiveness of the ontology which should be built once the development process is finished, based on the fact that it maps best practices from software development to ontology engineering (i.e. *Unified Process* [47,56]).

Since the UPON approach proposes only an informal guideline for the ontology development process, it is possible to slightly alter the different phases or workflows in order to fit the scope of the ontology more precisely.

Disadvantages: As the description above might already suggests, there is a huge effort in carrying out a development of an ontology with the UPON approach. The execution of various development cycles, each consisting of various phases, iterations and their underlying workflows leads to a development effort, which is only feasible when building large-scale knowledge bases.

Applicability for the Actor Preferences Ontology: Although the *UPON approach* incorporates two of the major requirements we stated for our ontology development methodology and therefore would be in general applicable to develop the Actor Preferences Ontology, we do not choose it as our preferred development strategy due to one major aspect: **Tremendous development effort**. As already mentioned above, the effort in using the *UPON approach* to develop an ontology definitely exceeds the needs of our scope and therefore disqualifies it as suitable development methodology.

4.2.4 Ontology 101

In their approach called *Ontology 101* [69], the authors describe an iterative strategy to develop an ontology as depicted in Figure 4.6. Furthermore, they emphasize three fundamental rules, which can easily be applied to any other ontology building methodology and should help to make decisions during the design of the ontology (paraphrased from [69]):

1. *There is no one correct way to model a domain. The best solution almost always depends on the application that you have in mind and the extensions that you anticipate.*

2. *Ontology development is necessarily an iterative process.*

3. *Concepts in the ontology should be close to objects and properties in your domain of interest. These are most likely to be nouns or verbs in sentences that describe your domain.*

Workflow Description

Definition of Ontology Scope A step which can be found in all ontology building methodologies and is essential for creating a comprehensive knowledge base, is the definition of competency questions. Those competency questions define the scope of the ontology to be built and are important to clarify the purpose of an ontology in very specific terms [72, 92].

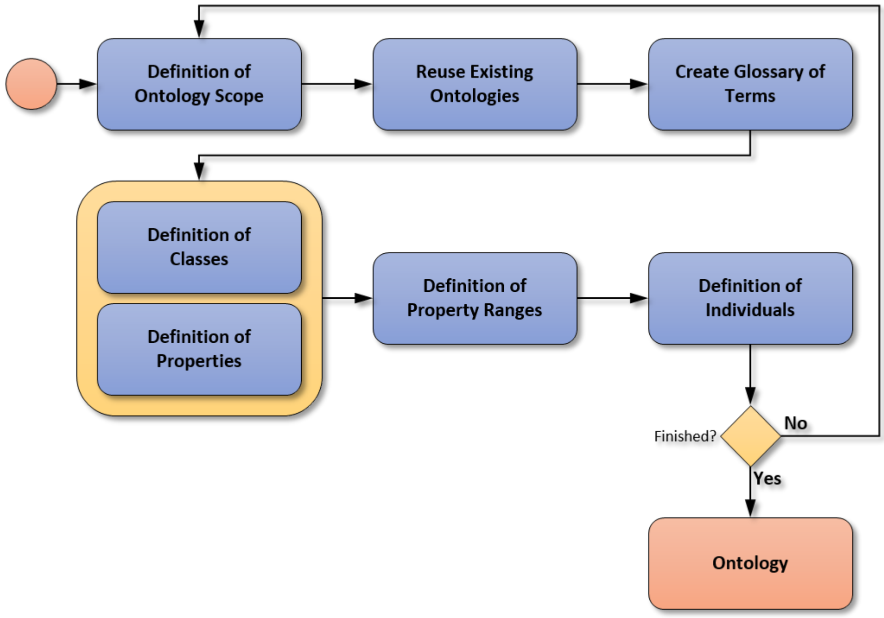

Figure 4.6: Workflow of the Ontology 101 development approach.

Reuse Existing Ontologies The reuse of already existing ontologies is not only beneficial if the ontology has to interact with other applications which are built upon ontologies, but also saves a lot of time since particular parts of the knowledge base do not have to be reinvented [83].

Create a Glossary of Terms Building a glossary of terms, which consists of a comprehensive list of terms, is an essential step of *Ontology 101*. Without worrying about the type (i.e. classes or properties), all terms which might be related to the application domain and the owner of the ontology wants to make statements about are gathered and then further processed in the next steps.

Definition of Classes & Properties First, all terms within the glossary which represent classes are identified and then used to define a class hierarchy among them. Based on that class hierarchy, the internal structure of concepts is described using the remaining terms of the glossary, which are most likely properties of those classes.

Definition of Property Ranges In this step, the cardinality, the value type and domain/range of properties are identified in order to define those properties more precisely.

Definition of Individuals Taking the remaining terms of the glossary into consideration, individual instances of classes are created. If the ontology can be considered to be finished, the result of this step leads to the completed ontology, otherwise the whole ontology development process starts with a new iteration.

Apart from the *Ontology 101* development approach, the authors provide several informal guidelines and best practices, which shall help data scientists in the task of building an ontology.

Analysis

Advantages: *Ontology 101* is probably one of the most prominent representatives of ontology development approaches and due to its simplicity rather easily understandable. Based on that simplicity, it is perfectly suitable for developing small-scale ontologies which does not need to be thoroughly documented in a fast way.

Disadvantages: Like the previous introduced methodology by Uschold & King, it lacks in a standardized way the tasks must be performed in order to develop an ontology. That might lead to inconsistent results when integrating ontologies, created following that approach. Furthermore, it does not enforce the creation of documentation artifacts in any way.

Applicability for the Actor Preferences Ontology: *Ontology 101* is one of the best known and easiest methodology to develop ontologies and was used to create ontologies such as an ontology for supporting engineering analysis models [31]. Nevertheless, due to several facts which are stated underneath it is not entirely applicable for our scope.

1. **Imprecise description of methodology tasks** - *Ontology 101* offers a set of tasks which should be performed within their iterative development process, but does not provide a detailed and standardized description on how that should look like. Since our ontology will be part of a system which incorporates several ontologies, a thorough and standardized development step description is mandatory.
2. **No documentation enforced** - *Ontology 101* does not require any documentation of the development process and/or the ontology itself, which makes it less applicable than other investigated methodologies.

4.3 Ontology Learning

Ontology Learning [63] focuses on the (semi-)automatic acquisition of knowledge and its respective transformation into ontologies. Rather than solely using domain experts to model and develop ontologies, ontology learning methods should support those experts in deriving relations and concepts within the domain of interest using data mining techniques [15] in a (semi-)automatic manner.

One of the major benefits of ontology learning is the reduction in costs of creating and maintaining ontologies, which has led to a vast amount of ontology learning frameworks (e.g. OntoLT [14]) which have been integrated with common ontology engineering tools (e.g. Protégé).

Ontology learning frameworks are usually based on the same conceptual architecture which is exemplified in Figure 4.7.

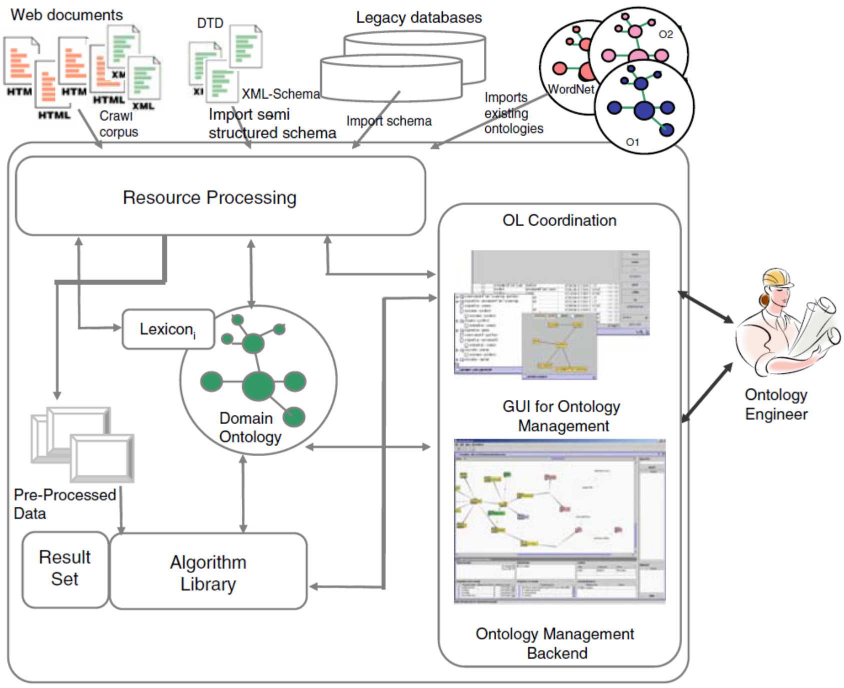

Figure 4.7: *Ontology Learning* conceptual architecture [15].

This general architecture distinguishes different kinds of data (structured, semi-structured and unstructured) and applies knowledge extraction strategies based on those respective types of data.

structured Processing structured data (e.g. extracting data from a database) facilitates the use of machine learning techniques [60] and the reuse of schemas, which can be derived from the database structure.

semi-structured Semi-structured data (e.g. web-pages, XML, . . .) sometimes require a data pre-processing step to identify common concepts and similarities among them (e.g. using Natural Language Processing (NLP) approaches [13,61])

unstructured Text documents (e.g. documentation) are classified as unstructured data and have to undergo a data pre-processing step as introduced above.

If the respective data-source has been processed and the knowledge extraction process has been finished, the results are usually presented to a domain expert and/or ontology engineer, who then investigates and reviews the generated results. Since a fully automatic knowledge acquisition is currently not achievable, this last step is mandatory and leads to a semi-automatic ontology generation process with human intervention [63].

Analysis

Advantages: *Ontology Learning* approaches usually decrease the amount effort required to develop an ontology. Especially for use-cases were an ontology must be created based on data stored in a set of documents or information which is distributed across several resources, ontology learning approaches have proven their feasibility.

Disadvantages: If an ontology is created in a (semi-)automatic manner, it usually must be revised by some domain experts to guarantee the correctness of inferred entities. While that revision might not impose an additional burden in terms of complexity for small ontologies which have almost no relations to external ontologies, it becomes quite complex when trying to create an ontology which has relations to others and thus, must stick to a potential nomenclature.

Applicability for the Actor Preferences Ontology: There are several reasons why think that the usage of ontology learning approaches for creating the *Actor Preferences Ontology* is not feasible:

1. **Compatibility with related ontologies must be ensured** - Supposing that we would have carried out the creation of the ontology with (semi-)automatic approaches, we could not guarantee its compatibility with other related ontologies of the *ThinkHome* system anymore. A complex revision of the generated ontology fragments would have to be carried out which in turn would exceed the effort of developing the ontology from scratch with another development approach like *METHONTOLOGY*.
2. **No documentation enforced** - One of our main requirements is to produce thorough documentation artifacts throughout the whole development process of the ontology. Unfortunately, ontology learning approaches, which (semi-)automatically derive entities of an ontology from a set of resources, do not impose the creation of any documentation artifacts and thus, do not fulfill this requirement.

4.4 The METHONTOLOGY Approach

For the development of the *Actor Preferences Ontology* we have decided to choose the *METHONTOLOGY* approach, due to *(i) its large acceptance in the ontology development community*, *(ii) its elaborate development process* and *(iii) its approach to produce comprehensive documentation*.

In the following, we will introduce each phase of the *METHONONTOLOGY* approach in more detail and give an example for the documentation artifacts which will be produced throughout the whole process.

4.4.1 Planification

Fernández et al. [23] argue, that an ontology development process only defines what activities have to be carried out when developing an ontology rather than defining a concrete order in which they have to be performed. Therefore, a plan which schedules the whole development process must be created during the *Planification* step. Since *METHONTOLOGY* already offers such a plan (cf. the workflow of *METHONTOLOGY* in Figure 4.4) , this step is often omitted when carrying out the development of an ontology with *METHONTOLOGY*.

4.4.2 Specification

During *Specification* an *Ontology Requirement Specification Document* must be generated which have to include at least following information:

- Purpose of the ontology, use cases, end-users, ...
- Level of formality of the ontology, ranging from highly informal to rigorously formal [93].
- Scope of the ontology, including a set of terms which should be represented together with its characteristics and granularity.

Furthermore, since the total completeness of a specification document [23] cannot be ensured, following properties must hold for the generated *Ontology Requirement Specification Document*:

Concision There must be no irrelevant or duplicated terms in the specification document.

Partial Completeness Although total completeness cannot be ensured, coverage of the terms must be as high as possible according to their granularity levels.

Consistency All terms and their meanings must be consistent to the application domain.

METHONTOLOGY does not propose any formal requirements to a specification document except for the guidelines introduced above. Hence, we rely on the definition of [86] for defining an *Ontology Requirement Specification Document* (cf. Section 5.1.2).

4.4.3 Knowledge Acquisition

One of the activities which is performed throughout the whole development process and orthogonal to all others is *Knowledge Acquisition*. It is mainly carried out simultaneously to the *Specification* phase and decreases as the development process continues.

The step of *Knowledge Acquisition* is very important since it is unlikely that the person who is developing the ontology has enough knowledge to build a comprehensive and complete ontology from scratch without consulting any domain experts or possible end-users. Other sources of knowledge are for example books, handbooks, figures, tables or related ontologies.

Once sources of additional knowledge are identified, techniques such as interviews, brainstorming, text analysis and knowledge acquisition tools can be used to gather that knowledge [23].

4.4.4 Conceptualization

Following a more fine-grained description of the *Conceptualization* step [29], which is illustrated in Figure 4.8, the task of structuring the domain knowledge in a conceptual model can be divided into following sub-tasks:

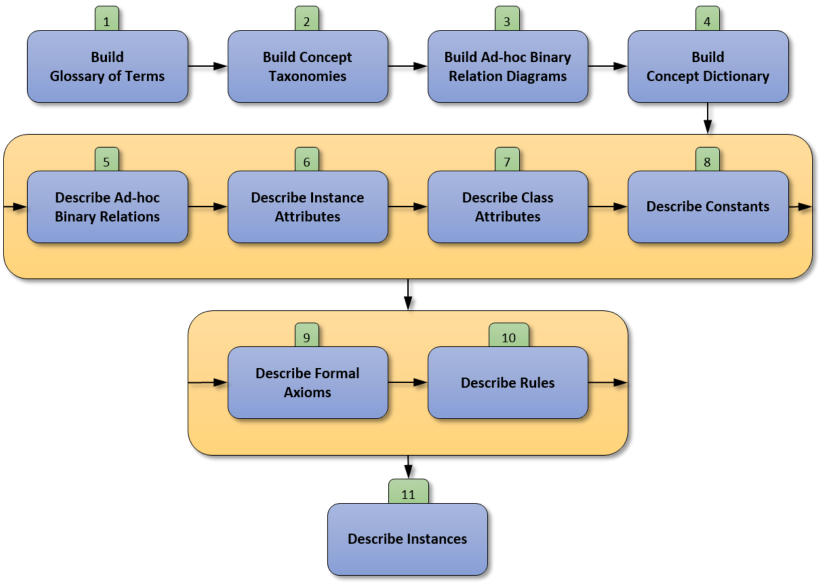

Figure 4.8: Set of tasks performed during *Conceptualization*.

1. **Building a Glossary of Terms** The first task of *Conceptualization* is to build a *Glossary of Terms*. Such a glossary includes all relevant terms of the domain of interest (i.e. concepts, instances, properties, relationships, ...) and is exemplified in Table 4.1.

Name	Synonyms	Acronyms	Description	Type
...	...	...	...	...

Table 4.1: Glossary of Terms template proposed by *METHONTOLOGY*

In early stages of the development, the glossary might get refactored many times in order to reduce redundancy among terms.

2. **Building Concept Taxonomies** Once the glossary contains a first set of concepts, *Concept Taxonomies* as exemplified in Figure 4.9 are built to define a hierarchical ordering among classes.

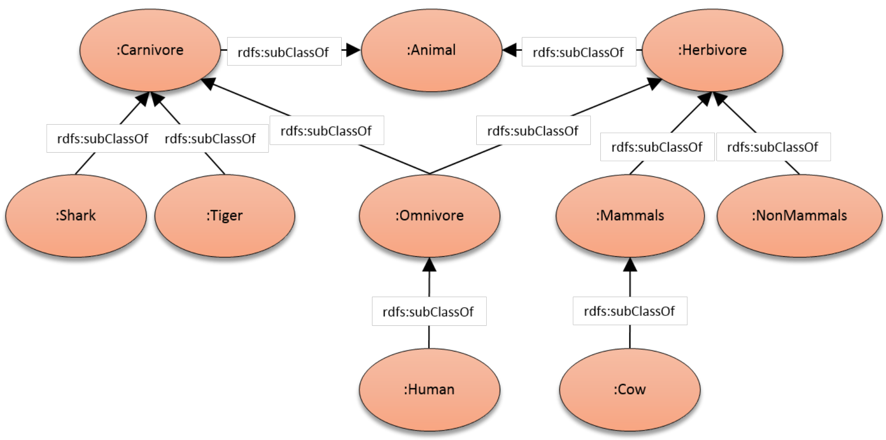

Figure 4.9: Concept Taxonomy proposed by *METHONTOLOGY*.

METHONTOLOGY proposes four different taxonomic relations which describe different concept-instance relations, namely:

Subclass-Of A concept C is a *Subclass-Of* another concept D iff every instance of C is also an instance of D.
Example: Every `Human` is born as an `Omnivore` and can digest both meat and vegetables.

Disjoint-Decomposition A *Disjoint-Decomposition* of a concept C is a set of subclasses of C that does not have common instances and does not cover C.
Example: Both `Sharks` and `Tigers` are `Carnivores`, but there are `Carnivores` which are not represented in the taxonomy (e.g. `Dogs)`

Exhaustive-Decomposition In contrast to *Disjoint-Decomposition*, an *Exhaustive-Decomposition* of C is a set of subclasses of C that covers C but may have common instances.
Example: `Animals` can be divided into `Carnivores` and `Herbivores` but there are instances which are both (cf. `Omnivore`).

Partition A *Partition* of a concept C is a set of subclasses of C that covers C but does not share common instances.
Example: The set of concepts that are `Herbivores` can be entirely split into `Mammals` and `NonMammals` and there exists no `Herbivore` which is both `Mammal` and `NonMammal`.

3. **Building Ad-hoc Binary Relation Diagrams** Once the *Concept Taxonomies* have been built, *Ad-hoc Binary Relation Diagrams* should be generated. Those diagrams represent relationships between concepts of the taxonomies and should help to identify possible imprecise or over-specified domains and ranges of properties. Figure 4.10 illustrates such an *Ad-hoc Binary Relation Diagram*.

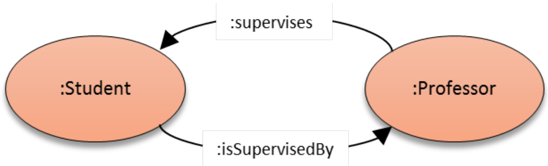

Figure 4.10: Ad-hoc Binary Relation Diagram proposed by *METHONTOLOGY*.

4. **Building a Concept Dictionary** To describe each concept of the *Concept Taxonomies* in more detail, a *Concept Dictionary* should be built. This dictionary contains all concept names, relations and instances of concepts together with their appropriate class and instance attributes and is exemplified in Table 4.2. To retrieve all required information to build such a dictionary, the results from previous steps are integrated and summarized.

Concept Dictionary				
Concept Name	Class Attributes	Instance Attributes	Instances	Relations
...	...	...		

Table 4.2: Concept Dictionary Table proposed by *METHONTOLOGY*

METHONTOLOGY does not propose any specific order to perform tasks 5 to 8, once tasks 1 to 4 are completed!

5. **Describing Ad-hoc Binary Relations** Describing the diagrams of task 3 in more detail, for each previous defined ad-hoc binary relation, its details (i.e. name,

source concept, target concept, source cardinality and inverse relation) are summarized and represented in a *Ad-hoc Binary Relation Table* as illustrated in Table 4.3.

Ad-hoc Binary Relation Table				
Relation Name	Source Concept	**Cardinality**	Target Concept	Inverse Relation
...	...	...	...	...

Table 4.3: Ad-hoc Binary Relation Table proposed by *METHONTOLOGY*

6. **Describing Instance Attributes** All instance attributes which are already included in the *Concept Dictionary* are defined more precisely during this step. Hence, a table of all instance attributes is created, each row describing its name, the concept it belongs to, its value type, its measurement unit, precision and range of values as well as its (min,max) cardinality.

   Furthermore, all instance attributes, class attributes or constants which are used to infer values of the attribute, attributes which can be inferred using values of this attribute, formulas or rules that allow inferring values of the attribute, and references used to define the attribute, can additionally be defined in the table.

Instance Attribute Table						
Attribute Name	Concept Name	Value Type	Measurement Unit	Precision	Range of Values	Cardinality
...	...	...	...	...	...	...

Table 4.4: Instance Attribute Table proposed by *METHONTOLOGY*

7. **Describing Class Attributes** Like for instances in the previous step, a table specifying class attributes more precisely is defined during this task. Unlike instance attributes, class attributes cannot be inherited by subclasses or instances, thus directly describe concepts and take their values in the class where they were defined.

   A possible incarnation of such a description, which includes information about the defined class itself, information about the concept it is defined for (i.e. its super concept), its value type, its measurement unit, its precision as well as (min,max) cardinality and values, is proposed in Table 4.5.

8. **Describing Constants** The aim of this task is to describe all constants, which were identified in the *Glossary of Terms*, in more detail. Therefore a *Constants Table* is created, containing information about the name of the constant, its value type, its measurement unit and its value as illustrated in Table 4.6.

Class Attribute Table				
Defined Concept	Super Concept	Attribute Name	Cardinality	Values
...	...	...	...	...

Table 4.5: Class Attribute Table proposed by *METHONTOLOGY*

Constant Table			
Name	Value Type	Measurement Unit	Value
...	...	...	...

Table 4.6: Constant Table proposed by *METHONTOLOGY*

Once all of the above tasks are finished, *(i) formal axioms which specify constraints in the ontology*, as well as *(ii) rules which infer additional knowledge* can be defined in their respective tables.

9. **Describing Formal Axioms** A *Formal Axioms Table* contains a definition for each formal axiom found in the ontology. Such a definition includes at least: a name, a natural language description, a logically expression which describes the axiom in first order logic, the entities to which the axiom refers and the variables it uses (cf. Table 4.7).

Formal Axiom Table						
Axiom Name	Description	Expression	Concepts	Referred Attributes	Binary Relations	Variables
...	...	...	...	...	...	...

Table 4.7: Formal Axiom Table proposed by *METHONTOLOGY*

10. **Describing Rules** Parallel to a *Formal Axioms Table*, a *Rule Table* can be defined (cf. Table 4.8), which precisely describes all rules found in the ontology.

    It basically contains the same columns as the previous defined table for formal axioms, but in contrast to a logically expression which describes the formal axiom in first order logic, an expression following the pattern `if <conditions> then <consequent>` will be used. Although `<conditions>` can be the conjunction of atoms, the `<consequent>` consists of only one atom.

11. **Describing Instances** As last step during the *Conceptualization* phase, all instances found in the the *Glossary of Terms* are described. These instances are listed once again in a tabular manner involving following columns: instance name, the concept it belongs to, its attributes and possible values, as depicted in Table 4.9.

Rule Table						
Rule Name	Description	Expression	Concepts	Referred Attributes	Binary Relations	Variables
...	...	...	...	...	...	...

Table 4.8: Rule Table proposed by *METHONTOLOGY*

Instance Table			
Instance Name	Concept Name	Attributes	Values
...	...	...	...

Table 4.9: Instance Table proposed by *METHONTOLOGY*

Remark - Based on the different needs and individual nature of ontologies, the step of *Conceptualization* may be altered (i.e. reducing or extending the level of detail of intermediate representations).

4.4.5 Formalization

Till this step, only informal descriptions of the entities of the ontology exist. Therefore, during *Formalization* all these descriptions are transformed into the target ontology language (e.g. OWL).

Although *METHONTOLOGY* proposes this step to be separately performed before the *Implementation*, it is actually mostly performed within the *Implementation* and therefore often omitted as separate step.

4.4.6 Integration

During *Integration*, all relevant ontologies which might be reusable are identified. This is especially important to

(i) save time and resources during the development process, since you do not have to redevelop already existing concepts, and

(ii) supporting the interlinking with other ontologies, since they might use the same concepts to describe their resources.

After they were identified, terms that shall be integrated into the ontology are summarized in an integration document and prepared to be further processed in the *Implementation* step.

4.4.7 Implementation

During *Implementation* the ontology finally gets serialized in an appropriate ontology language such as OWL. Although such a serialization could be carried out using a normal text editor, the authors of [23] propose to use an editor which supports at least:

- a lexical and syntactic analyzer to guarantee the absence of lexical or syntactic errors
- translators, to guarantee the portability of the definitions into other target languages
- an editor to add, remove or modify definitions
- a browser to inspect the library of ontologies and their definitions
- an evaluator to detect incompleteness, inconsistencies and redundant knowledge (i.e. an OWL reasoner)
- an automatic maintainer, to manage the inclusion, removal or modification of existing definitions

Nowadays, ontology editors like Protégé [1] or the TopBraid Composer [89] offer a large variety of features, supporting ontology developers in developing and maintaining their ontologies.

4.4.8 Evaluation

The phase of *Evaluation* takes place throughout the whole development process and assesses whether or not previously defined requirements are met. To do so, every artifact (table, diagram, serialized code) gets evaluated and compared against the *Ontology Requirement Specification Document* whenever it is altered, to prevent possible implementation errors.

After the completion of the development life cycle all functional and non-functional requirements must be met and in case of unrealizable requirements, decided whether or not that happened due to restrictions in the chosen ontology language or based on implementation mistakes.

4.4.9 Documentation

Similar to *Evaluation*, *Documentation* within the *METHONTOLOGY* approach is not performed explicitly, but during the whole development process. Following this approach of documentation, ontology engineers do not have to generate a documentation after the completion of the ontology, which might lead to incomplete or wrong information [23], but are able to finish their documentation together with the ontology.

4.4.10 Maintenance

Unfortunately, no ontology can be considered to cover the intended application domain completely and therefore might have to be modified after its release.

Such a modification usually comes hand in hand with a modification of the requirements of the ontology. If that is the case, the whole development process starts over again, starting with *Specification* and is repeated until all requirements are met.

CHAPTER 5

The Actor Preferences Ontology

Contents

5.1 Specification

The first step proposed by *METHONTOLOGY* called *Specification* imposes all requirements for the ontology which shall be developed by creating an *Ontology Requirements*

Specification Document [86]. We extended this step by identifying and discussing several use case scenarios for the *Actor Preferences Ontology* before we started to formulate the *Ontology Requirements Specification Document*.

5.1.1 Use Case Scenarios

In the following, we will motivate the choice of competency questions stated in the later defined *Ontology Requirements Specification Document* (cf. Section 5.1.2) by presenting use cases the CQs were extracted from. Each use case consists of *(i) a Story* which introduces its context of use, *(ii) a Discussion* which explains a possible realization approach of that use case in more detail, and finally *(iii) Related Competency Questions* which can be extracted from the respective *Discussion*.

UC1 - Storing Information about Actors and their Preferences

Story

> The smart home not only stores information about its actors but furthermore is able to persist a large variety of different types of preferences for its actors. These information can be used to either retrieve general statistical information about the actors and their preferences, gather a list of involved appliances/applications which are responsible for the realization of a certain preference, and most importantly to increase the user comfort of actors by automatically realizing the stored preferences. To measure user comfort, the system rates and persists the level of satisfaction of individual actors based on a predefined scale.

Discussion: To be able to store information about actors and their preferences the ontology must provide a possibility to describe these concepts. A neat definition of properties of actors (e.g. name, age, gender, level of satisfaction) as well as their preferences and properties of preferences (e.g. value, valid timeframe, valid zone), which should be classified based on their type, is mandatory. Preference profiles that can be used to provide a condensed way to link a set of preferences (and later introduced activities/schedules) to their owner shall be introduced. Additionally, specific types of actors and preferences must be automatically inferable using their characteristics (i.e. stored values of properties).

Related Competency Questions:

CQ1: Who are the actors of the smart home at hand?

CQ2: What is the average age of all (male/female) actors?

CQ3: What preferences does a specific actor have?

CQ4: What is the level of satisfaction of a specific user?

CQ7: What is the average comfort temperature of all actors?

CQ12: What applications are involved in order to realize a temperature preference in a specific room?

CQ13: Is a specific preference defined by a user and if so by whom?

UC2 - Defining Preferences for Activities

Story

> As an addition to various types of preferences which can be stored, the smart home offers the possibility to define sets of preferences which shall be active whenever a certain activity is performed. Any active or passive action (e.g. doing sports, reading a book, waking up in the morning, ...) can be described in terms of an activity, preferences like *Temperature: 20 °C, Blinds: down* defined for it, and classified as being one of the predefined activity types.

Discussion: The type of an activity shall be determined based on the action the activity describes (i.e. *nonpassive* or *passive*) and further distinguished into various predefined activity types. To link a set of preferences to a specific activity, they must be clustered within an activity preference profile and associated to the activity they were defined for. An actor shall then be able to use his activities by storing them into one of his preference profiles.

Related Competency Questions:

CQ18: Are there any activities stored for a specific actor?

CQ19: What preferences are involved in the *Zumba* activity of a specific actor?

CQ20: What kind of activity is the *Zumba* activity?

UC3 - Scheduling Preferences and Activities

Story

> Users of the smart home system are able to schedule their preferences and activities in preference/activity schedules. Previous non-scheduled preferences/activities can be associated with a certain timeframe within which they are supposed to be executed/valid. With these schedules users

can plan preferences/activities for entire time spans, which drastically decreases the amount of time necessary for users to interfere with the smart home system. Preference schedules can further be used to model timeframes certain areas (or the entire smart home) are occupied (cf. UC4).

Discussion: Preferences and activities shall be schedulable by associating a time description to their individuals. Every preference or activity which has such an associated time information must be automatically classified as scheduled preference/activity and a set of scheduled preferences/activities must be organized in preference schedules which are linked to preference profiles that are themselves associated to actors. To ensure a standardized representation of time information, existing ontologies offering the opportunity to define time definitions in a comprehensive way shall be explored.

Related Competency Questions:

CQ8: What is the preference schedule of a specific actor?

CQ9: Is there a scheduled preference on Mondays between 8am - 10am?

CQ10: Is the home occupied on Mondays at 11am?

CQ11: Is there a preference for a specific room or zone defined at a specific time?

CQ14: Will the scheduled dishwashing job of the dishwasher be finished at 1pm?

CQ15: Is it possible to re-schedule the dish washing job within its time window and still be finished at the desired end time?

CQ16: Are there concurring scheduled preferences?

CQ17: If there are concurring preferences, which one is active / shall be realized?

CQ21: What are the scheduled preferences and activities of a specific actor for Monday?

UC4 - Decrease Energy Consumption

Story

The smart home system not only focuses on providing user comfort but also permanently aims at ensuring an energy efficient execution of household appliances and applications which are responsible for realizing certain preferences. Whenever areas of the smart home (or the entire home) are unoccupied, minimum energy consumption of systems such as *lighting, air* *conditioning, ...* is ensured taking possible later scheduled preferences and their realization into account. Additionally, the executions of household appliances are re-scheduled within their timeframes to match low energy consumption peaks.

Discussion: Keeping track of the current occupancy state of the smart home is an important part of an efficient energy management. The smart home system has to store this information by introducing a new sub-type of preferences called presence preference. These presence preferences shall be linked to a specific timeframe and location, which uniquely identify the time a certain area of the smart home is occupied.

Related Competency Questions:

CQ10: Is the home occupied on Mondays at 11am?

CQ11: Is there a preference for a specific room or zone defined at a specific time?

CQ15: Is it possible to re-schedule the dish washing job within its time window and still be finished at the desired end time?

UC5 - Managing of Concurring and/or Scheduled Preferences

Story

If two or more preferences are concurring (i.e. scheduled for the same timeframe, for the same location, and are of the same type) the smart home chooses the preference with the highest importance to be realized. Preferences with a lower importance are either re-scheduled within their timeframe (if possible) or ignored at all.

Discussion: To be able to re-schedule and/or resolve issues with concurring preferences, it is necessary that the ontology is able to assign time descriptions to preferences. These time descriptions shall be realized by using an external ontology like the *OWL-Time* ontology, which offers a large variety of expressive ways to define time related concepts. Additionally, the level of importance of a certain preference must be definable by the knowledge base to offer a first and easy way to resolve concurrency issues.

Related Competency Questions:

CQ14: Will the scheduled dish washing job of the dishwasher be finished at 1pm?

CQ15: Is it possible to re-schedule the dish washing job within its time window and still be finished at the desired end time?

CQ16: Are there concurring scheduled preferences?

CQ17: If there are concurring preferences, which one is active / shall be realized?

UC6 - Definition of Standard Preferences based on Standardized Preference Values

Story

> In case there exists no defined preference value for a certain type of preference for a user, a standard preference value which is based on well-known standards like ASHRAE [1] is assumed by the system. These standard preferences are especially useful for providing guest visitors of the smart home with an initial set of preferences without the necessity to define an individual actor for them.

Discussion: A set of already predefined individuals of standard preferences shall be defined and their preference values determined based on aforementioned well-known standards. For that purpose a new sub-type of preferences, distinguishable from those preferences defined by users, has to be introduced. These predefined standard preferences have to cover the most prominent preference types such as comfort temperature, relative humidity, or lighting level.

Related Competency Questions:

CQ5: Are there any standard preferences stored?

CQ6: Are any standard preferences in use?

CQ22: Which unit of measurement does a temperature preference have?

CQ23: What is the standard relative humidity preference value and on which standard is it based on?

5.1.2 Ontology Requirements Specification Document

Name: *Actor Preferences Ontology*

[1] `https://www.ashrae.org/`

Purpose: The *Actor Preferences Ontology* is used to store, classify and schedule preferences of actors within a smart home system. Additionally activities, which are defined as sets of preferences, can be specified and scheduled in order to provide an even more sophisticated and comprehensive set of possibilities to support smart home residents in their daily lives.

Scope: The present ontology covers 11 main concepts, describing the domain of actor preferences:

Activity: Activities form groups of preferences which have to be fulfilled. They are divided into *PassiveActivities* and *NonPassiveActivities* and can be either scheduled or non-scheduled.

Activity Schedule: Groups scheduled activities.

Actor: User for which preferences are defined for. Primarily split into *HumanActors* and *SystemActors*.

Age: Every actor has an assigned age which is defined in years for *HumanActors* and hours for *SystemActors*.

Gender: Human actors can be either *Female* or *Male*.

LevelOfSatisfaction: Every human actor has a certain level of satisfaction, namely: *DisSatisfied, BarelySatisfied, Satisfied* or *VerySatisfied*.

LevelOfImportance: Every preference has a certain level of importance, namely: *LowImportance, AverageImportance* or *HighImportance*.

Preference: Describes certain preferences of actors. For example *AirFlowVelocityPreference, AirVentilationPreference, LampPreference, DryerPreference, LightingLevelPreference*, etc.

Preference Profile: Groups preferences, preference schedules, activities and/or activity schedules.

Preference Schedule: Groups scheduled preferences and thus allows the preference management of certain timeframes.

Preference Value: Every preference has an assigned preference value, which are distinguished into *BinaryPreferenceValues* and *ContinuousPreferenceValues*.

Implementation Language: The ontology is realized using Protégé [1] as ontology development platform, HermiT [82] and Pellet [71] as ontology reasoner and is implemented in OWL 2 [64].

Intended Users: Ontology-based smart home systems, especially the ThinkHome system.

Intended Use: The ontology enables the ThinkHome system to store, classify and schedule preferences and activities of smart home actors. Based on that information, other ontologies or a respective business logic can infer new knowledge and e.g. take actions to reduce energy consumption.

Ontology Requirements:

Non-Functional Requirements:

- Whenever it is possible and applicable, the ontology must reuse existing ontologies.
- The ontology must be thoroughly documented to ensure its reusability for non domain experts.

Functional Requirements:

CQ1: Who are the actors of the smart home at hand?

CQ2: What is the average age of all (male/female) actors?

CQ3: What preferences does a specific actor have?

CQ4: What is the level of satisfaction of a specific user?

CQ5: Are there any standard preferences stored?

CQ6: Are any standard preferences in use?

CQ7: What is the average comfort temperature of all actors?

CQ8: What is the preference schedule of a specific actor?

CQ9: Is there a scheduled preference on Mondays between 8am - 10am?

CQ10: Is the home occupied on Mondays at 11am?

CQ11: Is there a preference for a specific room or zone defined at a specific time?

CQ12: What applications are involved in order to realize a temperature preference in a specific room?

CQ13: Is a specific preference defined by a user and if so by whom?

CQ14: Will the scheduled dishwashing job of the dishwasher be finished at 1pm?

CQ15: Is it possible to re-schedule the dish washing job within its time window and still be finished at the desired end time?

CQ16: Are there concurring scheduled preferences?

CQ17: If there are concurring preferences, which one is active / shall be realized?

CQ18: Are there any activities stored for a specific actor?

CQ19: What preferences are involved in the *Zumba* activity of a specific actor?

CQ20: What kind of activity is the *Zumba* activity?

CQ21: What are the scheduled preferences and activities of a specific actor for Monday?

CQ22: Which unit of measurement does a temperature preference have?

CQ23: What is the standard relative humidity preference value and on which standard is it based on?

Glossary of Terms: The *Glossary of Terms* is defined in Section 5.2.1. A preliminary set of terms, based on the present requirements specification document, can be defined as follows:
actor, age, female actor, preference, level of satisfaction, standard preference, comfort temperature, preference schedule, mondays, scheduled preference, occupancy, specific room, zone, application, temperature preference, dishwasher, time frame, activity, zumba activity, unit of measurement, standard relative humidity preference, preference value;

5.2 Conceptualization

After specifying the requirements of the to be developed ontology, its concepts, properties, and individuals are conceptualized in the present section. We start with a *Glossary of Terms* in Section 5.2.1 and follow up with a *Concept Taxonomy* (cf. Section 5.2.2), *Binary Relation Diagram* (cf. Section 5.2.3), *Concept Dictionary* (cf. Section 5.2.4), *Binary Relation Table* (cf. Section 5.2.5), *Instance Attribute Table* (cf. Section 5.2.6), *Class Attribute Table* (cf. Section 5.2.7), and an *Instance Table* (cf. Section 5.2.8).

5.2.1 Glossary of Terms

As first sub-step of *Conceptualization* a *Glossary of Terms* must be defined, which covers all important terms used within the scope of the *Actor Preferences Ontology*. We have divided this glossary into a list of *classes* (cf. 5.2.1.1), *properties* (cf. 5.2.1.2), and (cf. *individuals* 5.2.1.3). All terms represented in red refer to a more comprehensive description in the Appendix.

5.2.1.1 Classes

The concepts represented within the *Actor Preferences Ontology* are primarily distinguishable into 11 main concepts and were already mentioned in Section 5.1.2. In Figures 5.1 and 5.2, these concepts together with their sub-concepts are listed and linked to their more detailed description in the Appendix.

1. Activity
   - 1.1 NonPassiveActivity
     - 1.1.1 CleaningActivity
     - 1.1.2 CookingActivity
     - 1.1.3 SportActivity
   - 1.2 PassiveActivity
     - 1.2.1 ReadingActivity
     - 1.2.2 SleepingActivity
     - 1.2.3 WakeUpActivity
     - 1.2.4 WatchingTvActivity
     - 1.2.5 WritingActivity
   - 1.3 ScheduledActivity
   - 1.4 NonScheduledActivity
2. ActivitySchedule
3. Actor
   - 3.1 AgedHumanActor
   - 3.2 FemaleHumanActor
   - 3.3 HumanActor
   - 3.4 MaleHumanActor
   - 3.5 MatureHumanActor
   - 3.6 SatisfiedHumanActor
   - 3.7 SystemActor
   - 3.8 UnsatisfiedHumanActor
   - 3.9 UserSystemActor
   - 3.10 YoungHumanActor
4. Age
   - 4.1 AdvancedHumanActorAge
   - 4.2 HumanActorAge
   - 4.3 MatureHumanActorAge
   - 4.4 SystemActorAge
   - 4.5 YoungHumanActorAge
5. Gender
6. LevelOfSatisfaction
7. LevelOfImportance
8. PreferenceProfile
   - 8.1 ActivityPreferenceProfile
   - 8.2 HumanActorPreferenceProfile
   - 8.3 NonScheduledPreferenceProfile
   - 8.4 ScheduledPreferenceProfile
   - 8.5 StandardPreferenceProfile
9. PreferenceSchedule
   - 9.1 AppliancePreferenceSchedule
   - 9.2 ApplicationPreferenceSchedule
   - 9.3 PresencePreferenceSchedule
   - 9.4 VisualComfortPreferenceSchedule
10. PreferenceValue
    - 10.1 BinaryPreferenceValue
    - 10.2 ContinuousPreferenceValue

Figure 5.1: List of Classes (1/2)

11. Preference

- **11.1 AirFlowVelocityPreference**
- **11.2 AirQualityPreference**
- **11.3 AirVentilationPreference**
- **11.4 ApplianceCentricPreference**
- **11.5 ApplicationCentricPreference**
- **11.6 BlindsPreference**
- **11.7 DishwasherPreference**
- **11.8 DryerPreference**
- **11.9 LampPreference**
- **11.10 LightingLevelPreference**
- **11.11 PresencePreference**
- **11.12 RelativeHumidityPreference**
- **11.13 ScheduledPreference**
- **11.14 SoundPressureLevelPreference**
- **11.15 StandardPreference**
- **11.16 TemperaturePreference**
  - **11.16.1 ComfortTemperaturePreference**
  - **11.16.2 SetbackTemperaturePreference**
- **11.17 UserDefinedPreference**
- **11.18 WashingmachinePreference**

Figure 5.2: List of Classes (2/2)

5.2.1.2 Properties

All concepts listed above are connected amongst each other (or related to concepts of other ontologies) using object properties. Additionally, some concepts are linked to datatypes via datatype properties. Properties that refer to external ontologies are marked with a (*) and those having an inverse property are illustrated as (**property <-> inverse property**.
Both, all object and datatype properties are listed in Figures 5.3 and 5.4.

5.2.1.3 Individuals

There exist predefined individuals for three concepts, namely: LevelOfSatisfaction - stating the satisfaction level of an actor, LevelOfImportance - stating the importance of a preference, and Gender - specifying the gender of a human actor. These individuals are defined as follows:

LevelOfSatisfaction
: DisSatisfied, BarelySatisfied, Satisfied, VerySatisfied

LevelOfImportance
: LowImportance, AverageImportance, HighImportance

Gender
: Female, Male

forActivity <-> isActivityOf
: relates ScheduledActivities to their non-scheduled counterpart.

hasActivitySchedule <-> isActivityScheduleOf
: relates PreferenceProfiles to ActivitySchedules.

hasPreference <-> isPreferenceOf
: relates Preferences to PreferenceProfiles/PreferenceSchedules.

hasPreferenceProfile <-> isPreferenceProfileOf
: relates PreferenceProfiles to Activities/Actors.

hasPreferenceSchedule <-> isPreferenceScheduleOf
: relates PreferenceSchedules to PreferenceProfiles.

hasPreferenceValue <-> isPreferenceValueOf
: relates PreferenceValues to Preferences.

hasAge
: relates an Age to an Actor.

hasGender
: relates a Gender to an Actor.

hasImportance
: relates a LevelOfImportance to a Preference.

controlsAppliance *
: relates ero:Appliances to a Preference.

currentlyLocatedIn *
: relates a gbo:Zone or gbo:Space to an Actor.

Figure 5.3: List of Properties (1/2)

hasSatisfactionLevel
: relates a LevelOfSatisfaction to an Actor.

hasScheduledActivity <-> isScheduledActivityOf
: relates ScheduledActivities to ActivitySchedules.

representedBy <-> represents
: relates a SystemActor to an Actor.

usesApplication
: relates ppo:Applications to a Preference.

forTime *
: relates a time:TemporalEntity to ScheduledPreferences or ScheduledActivities.

forSpace *
: relates a gbo:Space to a Preference.

forState *
: relates ero:States to PreferenceValues.

forZone *
: relates a gbo:Zone to a Preference.

hasHours
: defines the Age in hours as xsd:float.

hasID
: defines the ID of an Actor as xsd:integer.

hasName
: defines the name of an Actor as xsd:string.

hasValue
: defines the value of a PreferenceValue as xsd:Literal.

hasYears
: defines the Age in years as xsd:integer.

Figure 5.4: List of Properties (2/2)

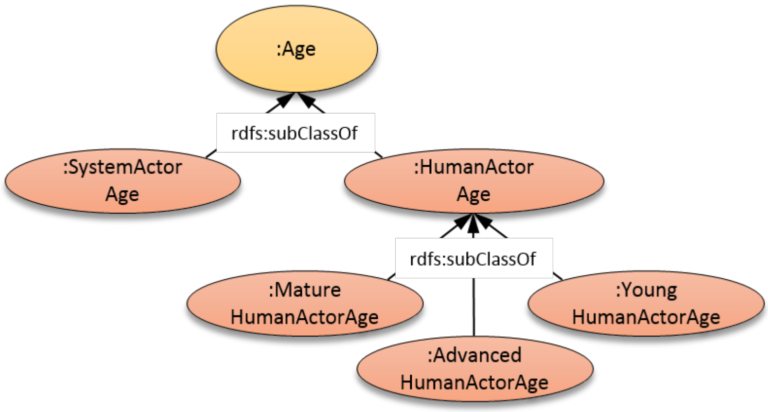

Figure 5.5: Concept taxonomy for `Age`

5.2.2 Concept Taxonomy

Age

The concept `Age` (cf. Figure 5.5) represents the age of an `Actor` either in hours for `SystemActors` or years for `HumanActors`. `HumanActors` can be further divided into three different groups based on their age, which are `YoungHumanActorAge` for an age below 14, `MatureHumanActorAge` for an age between 14 and 65, and `AdvancedHumanActorAge` for an age over 65.

Activity

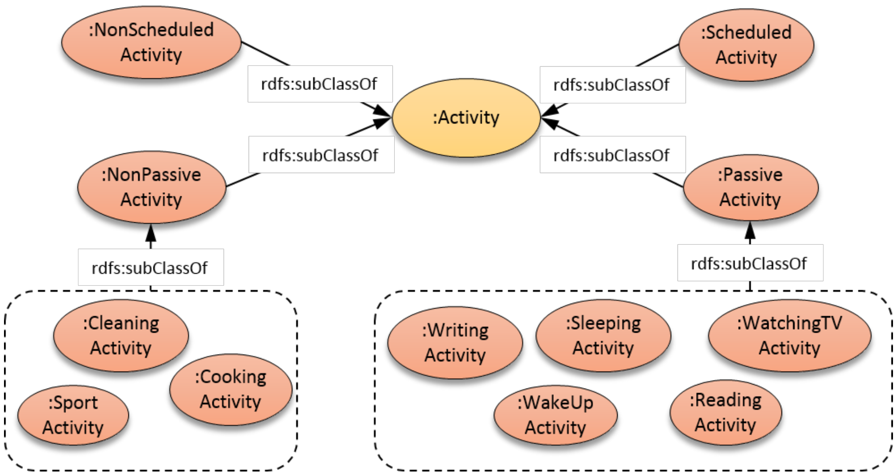

Figure 5.6: Concept taxonomy for `Activity`

The concept of `Activities` shall represent activities an `Actor` can perform within his home environment and for which he wants to define `Preferences` for. They are divided into passive and non-passive activities, each having several predefined activity types (cf. Figure 5.6 and the list below). Each `Activity` which is not associated to a specific time slot (i.e. not scheduled yet) is defined as `NonScheduledActivity` in contrast to scheduled ones which are called `ScheduledActivities`. Those `ScheduledActivities` are composed of a time description and the `NonScheduledActivity` containing the necessary `Preferences`.

NonPassiveActivity All activities which include some sort of (physical) exercise.

CookingActivity An activity which describes the action of cooking (e.g. a meal, a cake).

CleaningActivity An activity which describes any actions necessary to clean (parts of) the home.

SportActivity All activities that include some sort of workout and sport related actions (e.g. cardio training, home training).

PassiveActivity All activities which do not include any type of (physical) exercise.

ReadingActivity Activities which include actions related to reading.

SleepingActivity An activity which describes the process of sleeping.

WakeUpActivity All actions which form the process of waking up (e.g. in the morning, after a nap).

WatchingTVActivity The activity of watching television.

WritingActivity Activities which include actions related to writing (e.g. a letter, a book, a homework).

Actor

An `Actor` shall either represent `HumanActors` of a smart home, having an associated age, gender, and satisfaction level or `SystemActors` having an associated age and ID. Note that `SystemActors` are supposed to represent agents of the MAS (cf. Section 3.2.1) and shall perform actions on behalf of their corresponding users. Within the course of the present thesis we primarily focused on modeling and describing `HumanActors` rather than `SystemActors`, which we plan to do in future work. Both types of `Actors` are further divided into different sub-types based on following criteria:

AgedHumanActor is of age `AdvancedHumanActorAge`.

FemaleHumanActor has gender of type `Female`.

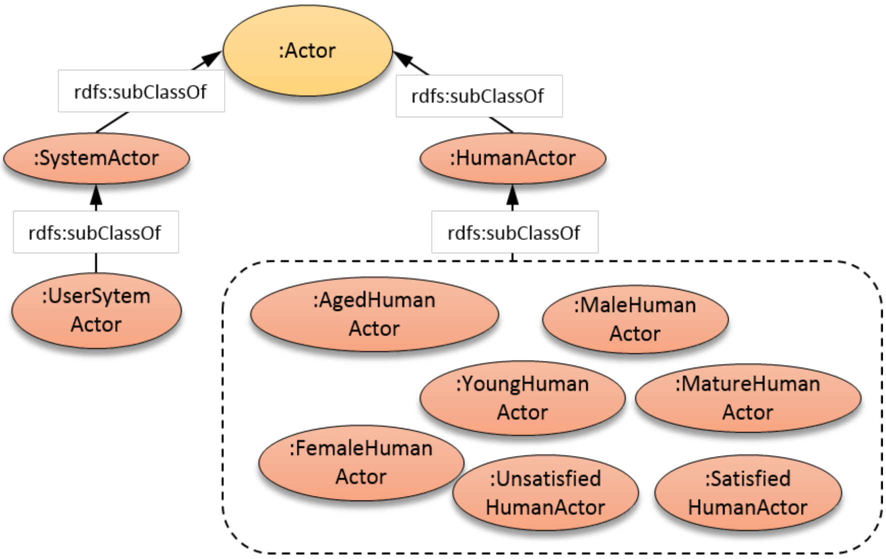

Figure 5.7: Concept taxonomy for `Actor`

MaleHumanActor has gender of type `Male`.

MatureHumanActor is of age `MatureHumanActorAge`.

SatisfiedHumanActor has satisfaction level of either `Satisfied` or `VerySatisfied`.

UnsatisfiedHumanActor has satisfaction level of either `BarelySatisfied` or `DisSatisfied`.

UserSystemActor is of age `SystemActorAge` and represents a `HumanActor` on whose behalf it performs its actions.

YoungHumanActor is of age `YoungHumanActorAge`.

Preference

A `Preference` (cf. Figure 5.8), as the major concept of the *Actor Preferences Ontology*, can be primarily defined as being a:

ScheduledPreference if it has an associated temporal entity,

ApplianceCentricPreference if it has an associated appliance which is responsible for their realization,

ApplicationCentricPreference if it has an associated application which is responsible for their realization,

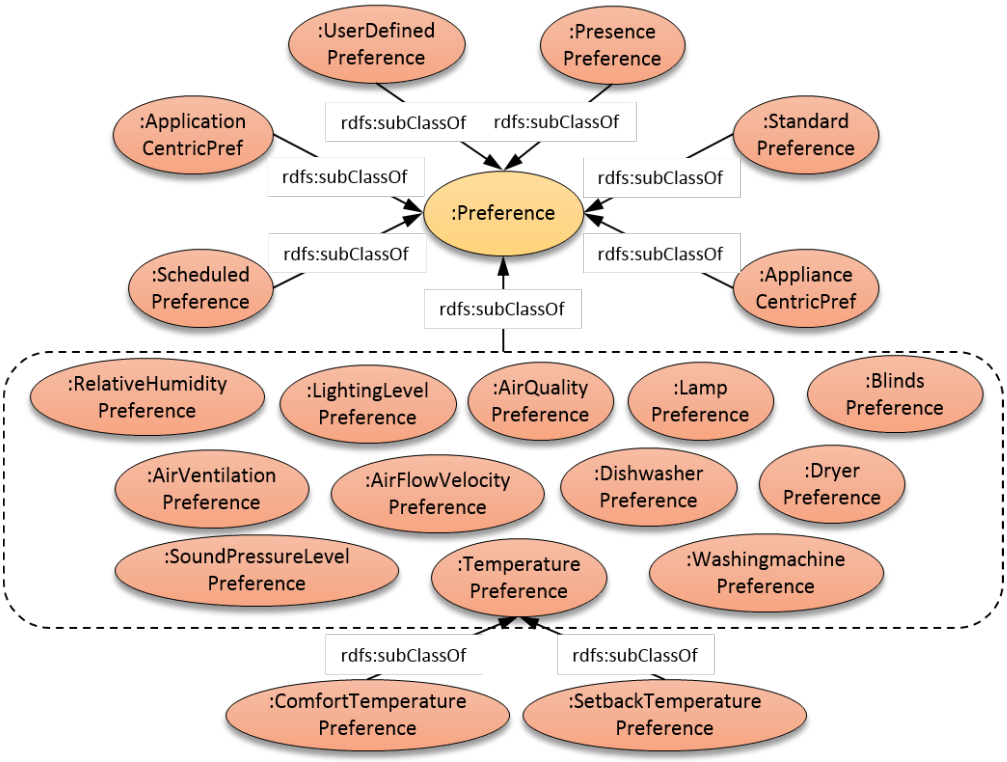

Figure 5.8: Concept taxonomy for `Preference`

PresencePreference if it is used to model occupancy of the smart home,

StandardPreference if it is part of a `StandardPreferenceProfile`,

UserDefinedPreference if it is part of a `HumanActorPreferenceProfile` and thus not a `StandardPreference`.

They can be further classified in being one of the following types:

AirFlowVelocityPreference describing the velocity of air flow in *m/s*.

AirQualityPreference describing the air quality in *parts per million*.

AirVentilationPreference describing the air ventilation frequency in *l/h*.

BlindsPreference describing the relative state of blinds.

DiswasherPreference describing the state (on/off) of a dishwasher.

DryerPreference describing the state (on/off) of a dryer.

LampPreference describing the state (on/off) and light intensity of lamps.

LightingLevelPreference describing the lighting level in *lux*.

RelativeHumidityPreference describing the relative humidity in *percent*.

SoundPressureLevelPreference describing the sound pressure in *dB*.

TemperaturePreference describing the temperature in *degrees celsius*.

ComfortTemperaturePreference describing the comfort temperature of an actor.

SetbackTemperaturePreference describing the standard setback temperature.

WashingmachinePreference describing the state (on/off) of a washing machine.

PreferenceProfile

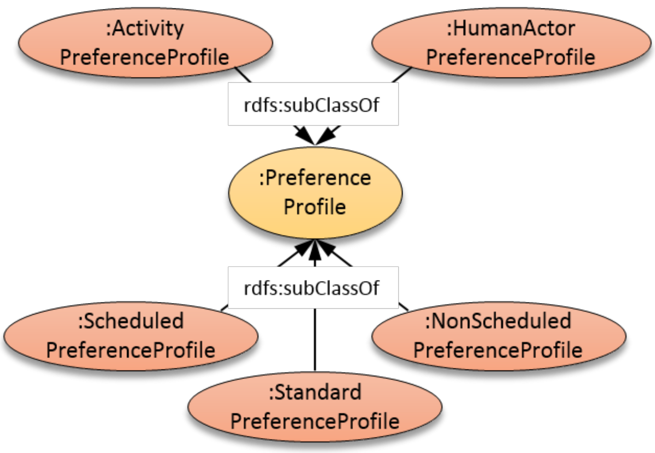

Figure 5.9: Concept taxonomy for `PreferenceProfile`

Besides **`ScheduledPreferenceProfiles`** which contain at least one `PreferenceSchedule` or `ActivitySchedule` and their respective counterparts **`NonScheduledPreferenceProfiles`** which contain at least one `Preference` or `NonScheduledActivity`, there exist three different types of `PreferenceProfiles`, namely:

ActivityPreferenceProfiles which contain `Preferences` for `Activities`,

HumanActorPreferenceProfiles which belong to a `HumanActor`, and

StandardPreferenceProfiles which contain `StandardPreferences` and do not belong to any explicit `HumanActor`.

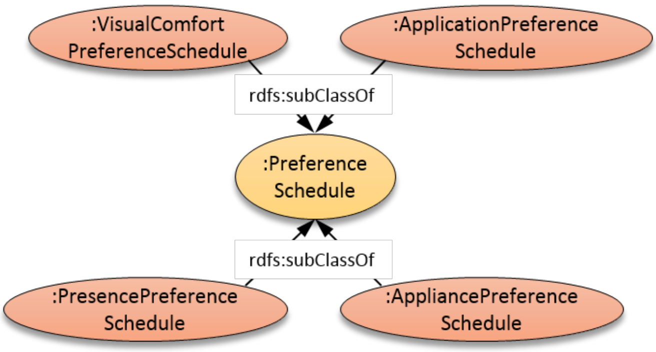

Figure 5.10: Concept taxonomy for `PreferenceSchedule`

PreferenceSchedule

`PreferenceSchedules` either contain a set of `ScheduledPreferences` or a set of `PresencePreferences` which are responsible to model the occupancy schedule of the home (hence, being a `PresencePreferenceSchedule`) or `ScheduledPreferences` which model any other type of preference. The latter ones can be defined as:

AppliancePreferenceSchedule if they contain any `ApplianceCentricPreference`,

ApplicationPreferenceSchedule if they contain any `ApplicationCentricPreference`, or

VisualComfortPreferenceSchedule as an example for a more specific `PreferenceSchedule`, serving the purpose of containing `ScheduledPreferences` which are responsible of ensuring visual comfort.

PreferenceValue

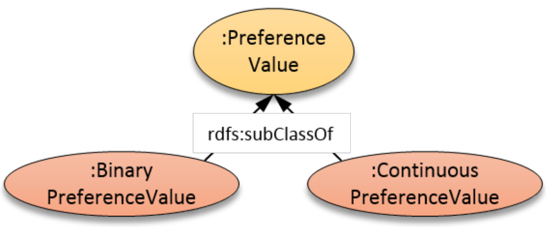

Figure 5.11: Concept taxonomy for `PreferenceValue`

A `PreferenceValue` can either contain binary values and thus be classified as `BinaryPreferenceValue` or contain continuous values and be classified as `ContinuousPreferenceValue`. The latter type uses units of measurement defined in the *Ontology of Units of Measure and Related Concepts (OM)* [79] to represent the units of its values.

ActivitySchedule, Gender, LevelOfImportance, and LevelOfSatisfaction

Those four concepts do not contain any sub-concepts. Thus, they are not illustrated as concept taxonomies.

5.2.3 Binary Relation Diagram

Figure 5.12 shows all binary relations between the main concepts of the *Actor Preferences Ontology*. Please note, that every property starting with `has<name>` has a respective inverse property named `is<name>Of`, as exemplified with the two properties `hasPreferenceSchedule` and `isPreferenceScheduleOf`[2].

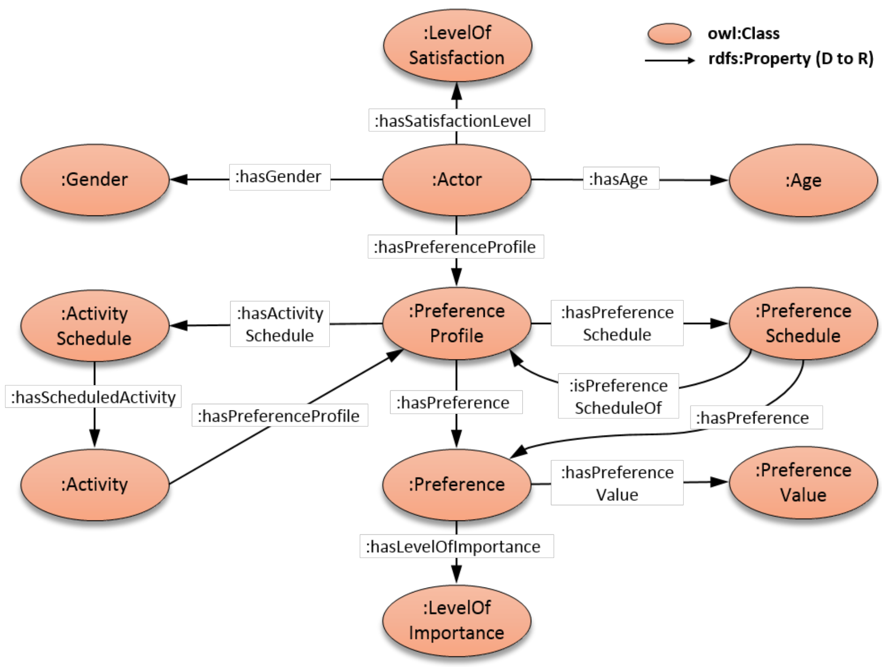

Figure 5.12: Binary relation diagram consisting of the main concepts of the *Actor Preferences Ontology*

[2]Except for the relations `hasAge`, `hasGender`, `hasSatisfactionLevel`, and `hasLevelOfImportance`.

5.2.4 Concept Dictionary

Before documenting all relations of concepts and as an addition to their verbal description in Section 5.2.2, the concepts themselves get documented in more detail within this section. We have listed all *Concept Names* together with their *Instance Attributes, Instances* and/or *Relations* if present and grouped them by their super-concept in Tables A.1 to A.11.

5.2.5 Binary Relation Table

The *Ad-hoc Binary Relation Table* (cf. Table A.12) describes and extends the relations illustrated in Section 5.2.3 in more detail. It consists of the *Relation Name, Source/Target Concept, Cardinality,* and *Inverse Relation* if available.

5.2.6 Instance Attribute Table

Instance Attributes, which are usually of type `owl:DatatypeProperties`, are those attributes having different values for each instance and therefore are responsible to describe individuals/instances of concepts in more detail. The ones used within the *Actor Preferences Ontology* are summarized in Table A.13.

5.2.7 Class Attribute Table

When defining *Class Attribute Tables* the focus usually lies in the description of properties (*Class Attributes*) which are responsible for the definition/specialization of concepts (cf. `Actor -> HumanActor`) itself rather than in the description of those attributes (*Instance Attributes*) that describe the instances of the concept and whose value(s) may be different for each instance of the concept (cf. `hasName`) [16]. In contrast to the proposed approach of designing such *Class Attribute Table*, we do not group the entries by their *Attribute Name* but by the *Defined Concept* they define, which allows a much more concise representation. We have documented all *Class Attributes* in Table A.14 to A.18.

5.2.8 Instance Table

Finally, individuals of concepts which were already predefined are documented. The *Instance Table* comprises concrete representations of the concepts `Gender, LevelOfImportance, and LevelOfSatisfaction` and is located in the appendix in Table A.19.

5.3 Integration

An essential part of ontology driven modeling is the reuse of existing ontologies whenever possible and suitable [83]. In the domain of storing and scheduling actor preference information we identified two different areas for which existing external ontologies can be used, namely (cf. Section 2.6 for a more comprehensive introduction):

Time information: The usage of *OWL-Time (time)* [39] for representing temporal information allows to schedule preferences and activities using a standardized temporal information representation.

Units of measurement information: The *Ontology of Units of Measure and Related Concepts (OM)* [79] is used to introduce units to values of preferences. This is a big advantage in comparison to a simple string representation especially if the *Actor Preferences Ontology* must be integrated with other related ontologies.

5.4 Implementation

The steps of *Specification* (cf. Section 5.1.2) and *Conceptualization* (cf. Section 5.2.8) provide the foundations to actually implement the *Actor Preferences Ontology*. We have conducted the step of *Implementation* using Protégé 4.3 [1] as ontology development platform, HermiT [82] and Pellet [71] as ontology reasoners and OWL 2 [64] as language of implementation.

The choice to use HermiT as primary ontology reasoner is based on the facts, that (i) it performs best (i.e. fastest execution time of reasoning tasks) amongst all tested ontology reasoners (cf. Section 6 for a detailed discussion on OWL reasoner evaluation), and (ii) the absence of SWRL rules to be evaluated which would have enforced the usage of Pellet as it would have been the only ontology reasoner capable of achieving this task.

5.5 Evaluation

As a last step, the developed ontology gets evaluated based on the functional and non-functional requirements which were defined in the *Ontology Requirements Specification Document* 5.1.2. Only if all requirements are met, the development process can be assumed to be completed.

5.5.1 Non-Functional Requirements

Two major non-functional requirements were identified and met, namely:

Reuse of existing ontologies: As already stated in Section 2.6, we reused the *Time Ontology (owl-time)* for representing time related concepts (i.e. linking concepts via `forTime` to specific time information) and the *Ontology of Units of Measure and Related Concepts (OM)* for representing units of measurement of `PreferenceValues`.

Thorough documentation: One major reason why we chose *METHONTOLOGY* as ontology development approach was that it enforces the creation of several documentation artifacts throughout the whole development process, which leads to a comprehensive documentation once the development process is finished.

5.5.2 Functional Requirements

In the following, we will evaluate the competency questions (CQ) introduced in the *Ontology Requirements Specification Document* 5.1.2. We show the fulfillment of each CQ by proposing SPARQL queries which are capable of answering their respective CQ and by discussing them afterwards.

CQ1: Who are the actors of the smart home at hand?

Listing 5.1:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX act: <https://www.auto.tuwien.ac.at/.../ActorOnt.owl#>
SELECT ?actor WHERE {
  ?actor rdf:type+ act:Actor .
}
```

Explanation: This query returns all individuals of type `Actor` of the smart home system at hand. By using SPARQL property paths[3] it is possible to traverse through the subclass hierarchy and retrieve all actors, even though they might not be a direct instantiation of `Actor`.

CQ2: What is the average age of all female actors?

Listing 5.2:

```
%Version 1
PREFIX act: <https://www.auto.tuwien.ac.at/.../ActorOnt.owl#>
SELECT (AVG(?ageYears) as ?avgYears) WHERE {
  ?actor act:hasGender ?gender .
  ?actor act:hasAge/act:hasYears ?ageYears .
  FILTER(?gender = act:Female)
} GROUP BY ?gender
```

[3] Newly introduced with SPARQL 1.1

```
------------- Assuming SPARQL reasoning capabilities -------------

%Version 2
PREFIX act: <https://www.auto.tuwien.ac.at/.../ActorOnt.owl#>
SELECT (AVG(?ageYears) as ?avgYears) WHERE {
  ?actor rdf:type act:FemaleHumanActor .
  ?actor act:hasAge/act:hasYears ?ageYears .
}
```

Explanation: The `Gender` of a `HumanActor` is assigned via its `hasGender` object property. Furthermore, the defined class `FemaleHumanActor` is specified as `Actor and (hasGender value Female)` which makes it possible for an OWL reasoner to derive that every individual of type `Actor` which has a `hasGender` property with value `Female` is additionally of type `FemaleHumanActor`. The queries presented in Listing 5.2 both use a SPARQL property path `hasAge/hasYears` to retrieve the `Age` of the `HumanActor` in years and either (*Version 1*) do not rely on reasoning capabilities to retrieve only `FemaleHumanActors` by filtering out all non `Female Actors` or (*Version 2*) rely on inference and query for individuals of type `FemaleHumanActor`. After that, the average age is calculated using the aggregation function AVG.

CQ3: What preferences does a specific actor have? (except those used in activities)

Listing 5.3:

```
PREFIX act: <https://www.auto.tuwien.ac.at/.../ActorOnt.owl#>
SELECT ?preferences WHERE {
  act:ActorHumanActor1 act:hasPreferenceProfile/act:hasPreferenceSchedule?
                          /act:hasPreference ?preference .
 }
```

Explanation: All `Actors` store their `Preferences` in `PreferenceProfiles`, where `Preferences` can be further distinguished into `ScheduledPreferences` and those which are not scheduled. All `Preferences` of a specific `Actor` (in this example `ActorHumanActor1`) can be retrieved by a SPARQL property path, which matches either on `Preferences` stored in a `PreferenceSchedule` and thus being a `ScheduledPreference` or on those which are not part of a `ScheduledPreference` and thus being a non-scheduled `Preference`.

CQ4: What is the level of satisfaction of a specific user?

Listing 5.4:

```
PREFIX act: <https://www.auto.tuwien.ac.at/.../ActorOnt.owl#>
SELECT ?sat WHERE {
  act:ActorHumanActor1 act:hasSatisfactionLevel ?sat .
```

```
}
```

Explanation: A `LevelOfSatisfaction` can be assigned to every `HumanActor` via the `hasSatisfactionLevel` object property. The `LevelOfSatisfaction` concept stores the actual satisfaction level of a specific user and is essential for applications which are responsible to ensure user comfort within the smart home system.

CQ5: Are there any standard preferences stored?

Listing 5.5:

```
PREFIX act: <https://www.auto.tuwien.ac.at/.../ActorOnt.owl#>
SELECT ?stdPref WHERE {
  ?stdPrefProf a act:StandardPreferenceProfile .
  ?stdPrefProf act:hasPreference ?stdPref .
}
```

Explanation: In case no `Preference` is stored for a specific `HumanActor` (e.g. for guests), `StandardPreferences` can be used. Those `StandardPreferences` are usually specified beforehand and stored within a `StandardPreferenceProfile`. They rely on certain standards for comfort temperature, relative humidity, etc. which are defined by 3^{rd} party institutions such as ASHRAE.

CQ6: Are any standard preferences in use?

Listing 5.6:

```
PREFIX act: <https://www.auto.tuwien.ac.at/.../ActorOnt.owl#>
ASK WHERE {
  ?actor a act:Actor .
  ?actor act:hasPreferenceProfile/act:hasPreferenceSchedule?
            /act:hasPreference ?preference .
  ?stdPrefProf a act:StandardPreferenceProfile .
  ?stdPrefProf act:hasPreference ?preference .
}
```

Explanation: To check whether a `StandardPreference` (cf. Listing 5.5) is currently in use, all preferences which are assigned to `Actors` and are part of a `StandardPreferenceProfile` are queried.

CQ7: What is the average comfort temperature of all actors?

Listing 5.7:

```
PREFIX act: <https://www.auto.tuwien.ac.at/.../ActorOnt.owl#>
SELECT (AVG(?val) as ?avgVal)  WHERE {
```

```
  ?comPref a act:ComfortTemperaturePreference.
  ?comPref act:hasPreferenceValue/act:hasValue ?val
  FILTER NOT EXISTS {
    ?stdPrefProf a act:StandardPreferenceProfile .
    ?stdPrefProf act:hasPreference ?comPref .
  }
}
```

Explanation: It is useful to be able to make general statements about all `Actors` of a smart home system. For example to calculate the average `PreferenceValue` of `ComfortTemperaturePreferences` of `Actors`, which could then be used to define a `StandardPreferenceValue` for comfort temperature, without relying on any 3^{rd} party standards. In Listing 5.7 the average of all `ComfortTemperaturePreferences` is calculated, excluding all `StandardPreferences` from the average calculation.

CQ8: What are the preference schedules of a specific actor?

Listing 5.8:

```
PREFIX act: <https://www.auto.tuwien.ac.at/.../ActorOnt.owl#>
SELECT ?prefSchedules WHERE {
  act:ActorHumanActor1 act:hasPreferenceProfile/act:hasPreferenceSchedule
                           ?prefSchedules .
}
```

Explanation: As already mentioned in Listing 5.3 `Preferences` can either be scheduled or non-scheduled. `ScheduledPreferences` are related to one or more `PreferenceSchedules` which are themselves accessible through `PreferenceProfiles`.

CQ9: Is there a scheduled preference which is active on Mondays between 8am - 10am?

Listing 5.9:

```
PREFIX time: <http://www.w3.org/2006/time#>
PREFIX act: <https://www.auto.tuwien.ac.at/.../ActorOnt.owl#>
ASK WHERE {
  ?prefs act:forTime ?time .
  ?time time:hasBeginning/time:inDateTime ?bDate .
  ?time time:hasEnd/time:inDateTime ?eDate .
  ?bDate time:dayOfWeek ?bDay .
  ?eDate time:dayOfWeek ?eDay .
  ?bDate time:hour ?bHour .
  ?eDate time:hour ?eHour .
  FILTER (?bDay = time:Monday && ?eDay = time:Monday)
  FILTER ((?bHour >= 8 && ?bHour <= 10) ||
```

```
            (?eHour >= 8 && ?eHour <= 10) ||
            (?bHour < 8 && ?eHour > 10))
}
```

Explanation: `ScheduledPreferences` are `Preferences` which are related to a specific time period via the `forTime` object property. They are either (i) valid between two timestamps defined with the `hasBeginning` and `hasEnd` property or (ii) valid for a certain amount of time (`hasDurationDescription`) between two timestamps. Those time descriptions make it possible to check, whether or not certain preferences are valid during a specific timeframe, which is exemplified in Listing 5.9.

CQ10: Is the home occupied on Mondays at 11am?

Listing 5.10:

```
PREFIX time: <http://www.w3.org/2006/time#>
PREFIX act: <https://www.auto.tuwien.ac.at/.../ActorOnt.owl#>
ASK WHERE {
  ?presPref a act:PresencePreference .
  ?presPref act:forTime ?time .
  ?time time:hasBeginning/time:inDateTime ?bDate .
  ?time time:hasEnd/time:inDateTime ?eDate .
  ?bDate time:dayOfWeek ?bDay .
  ?eDate time:dayOfWeek ?eDay .
  ?bDate time:hour ?bHour .
  ?eDate time:hour ?eHour .
  FILTER (?bDay = time:Monday && ?eDay = time:Monday)
  FILTER (?bHour <= 11 && ?eHour >= 11)
}
```

Explanation: Especially for energy related tasks (e.g. reducing unnecessary energy consumption), information of occupation of the smart home at hand is very important. This information is stored within `PresencePreferences`, which are `Scheduled-Preferences` that model the occupied timeframes via their `forTime` object property. The query stated above returns *true*, if the home is occupied at the specified timeframe.

CQ11: Is there a preference for a specific room or zone defined at a specific time?

Listing 5.11:

```
PREFIX gbo: <https://www.auto.tuwien.ac.at/.../BuildingOnt.owl#>
PREFIX act: <https://www.auto.tuwien.ac.at/.../ActorOnt.owl#>
ASK WHERE {
  ?preferences act:forSpace ?space .
  ?preferences act:forTime act:LR1HA1_ScheduledTemperaturePreferenceDuration_2HMo1800
  FILTER(?space = gbo:Space_ID_zon001)}
```

Explanation: `Preferences` can not only be assigned to a specific timeframe (`forTime`) but also to a specific space (`forSpace`) or zone (`forZone`). Adding space/zone information to `Preferences` offers the possibility to define their validity for defined regions within the smart home, which makes it easier to only trigger applications and appliances within that region to fulfill those `Preferences` and thus preserving energy.

CQ12: What applications are involved in order to realize a temperature preference in a specific room?

Listing 5.12:

```
PREFIX act: <https://www.auto.tuwien.ac.at/.../ActorOnt.owl#>
SELECT ?applications WHERE {
  ?pref a act:ComfortTemperaturePreference .
  ?pref act:forSpace ?space .
  ?pref act:usesApplication ?applications
  FILTER (?space = gbo:Space_ID_zon001)
}
```

Explanation: As mentioned earlier, `Preferences` can either be realized by using applications (`usesApplication`) or appliances (`controlsAppliance`). A list of applications which are involved in the realization of a certain `Preference` can be retrieved by the query exemplified in Listing 5.12.

CQ13: Is a specific preference defined by a user and if so by whom?

Listing 5.13:

```
PREFIX act: <https://www.auto.tuwien.ac.at/.../ActorOnt.owl#>
SELECT ?actor WHERE {
  act:LR1HA1_ComfortTemperaturePreference a act:UserDefinedPreference.
  act:LR1HA1_ComfortTemperaturePreference isPreferenceOf/isPreferenceScheduleOf?
                                                    /isPreferenceProfileOf ?actor .
}
```

Explanation: If we want to decide whether or not a specific `Preference` is a `UserDefinedPreference` and if that is the case, to query for the `Actor` who is assigned to that `Preference`. We first have to check if a certain `Preference` is of type `UserDefinedPreference` and then backtrack to its assigned owners by using the inverse property path
`isPreferenceOf/isPreferenceScheduleOf?/isPreferenceProfileOf`.

CQ14: Will the scheduled dishwashing job of the dishwasher be finished at 1pm?

Listing 5.14:

```
PREFIX time: <http://www.w3.org/2006/time#>
PREFIX act: <https://www.auto.tuwien.ac.at/.../ActorOnt.owl#>
ASK WHERE {
  act:HA1_ScheduledDishwasherPreference_Mo1200 act:forTime ?time .
  ?time time:hasEnd/time:inDateTime/time:hour ?eHour .
  FILTER (?eHour <= 13)
}
```

Explanation: An essential part of increasing user comfort and satisfiability within a smart home, is the possibility to get information about scheduled jobs (e.g. dishwashing/washing machine/etc.) which are represented via `ScheduledPreferences`. The query presented in Listing 5.14 returns *true* if `HA1_ScheduledDishwasherPreference_Mo1200` is scheduled to be finished before 1pm.

CQ15: Is it possible to re-schedule the dishwashing job within its time window and still be finished at the desired end time?

Listing 5.15:

```
PREFIX time: <http://www.w3.org/2006/time#>
PREFIX act: <https://www.auto.tuwien.ac.at/.../ActorOnt.owl#>
ASK WHERE {
  act:HA1_ScheduledDishwasherPreference_Mo1200 act:forTime ?time .
  ?time time:hasEnd/time:inDateTime/time:hour ?eHour .
  ?time time:hasBeginning/time:inDateTime/time:hour ?bHour .
  ?time time:hasDurationDescription/time:hours ?duration .
  FILTER ((?eHour-?bHour) > ?duration)
}
```

Explanation: As an addition to Listing 5.14 and assuming the presence of a `DurationDescription` of the task to be executed, one can even check if it is possible to re-arrange the dishwashing job within its timeframe (e.g. to shift it on an off-peak time-slot). This can be achieved by calculating the difference between the length of the timeframe and the actual duration of the task to be executed.

CQ16: Are there concurring scheduled preferences?

Listing 5.16:

```
PREFIX act: <https://www.auto.tuwien.ac.at/.../ActorOnt.owl#>
SELECT ?pref1 ?pref2 WHERE {{
  ?pref1 a ?type .
```

```
  ?pref2 a ?type .
  ?pref1 act:forTime ?time .
  ?pref2 act:forTime ?time .
  ?pref1 act:forSpace ?space .
  ?pref2 act:forSpace ?space .
}
FILTER NOT EXISTS {
 ?pref1 act:controlsAppliance ?app1 .
 ?pref2 act:controlsAppliance ?app2 .
 FILTER (?app1 != ?app2)
 }
FILTER NOT EXISTS {
 ?pref1 act:usesApplication ?appl1 .
 ?pref2 act:usesApplication ?appl2 .
 FILTER (?appl1 != ?appl2)
 }
 FILTER(?pref1 != ?pref2)
}
```

Explanation: Sometimes it may happen, that two or more `Preferences` of the same type are defined for the same timeframe, space/zone, control the same appliances, and use the same application. In order to detect such concurring preferences a query as exemplified in Listing 5.16 can be used.

CQ17: If there are concurring preferences, which one is active / shall be realized?

Listing 5.17:

```
PREFIX act: <https://www.auto.tuwien.ac.at/.../ActorOnt.owl#>
SELECT ?pref1 ?importance1 ?pref2 ?importance2 WHERE {
  %assuming ?pref1 and ?pref2 from CQ16
  ?pref1 act:hasImportance ?importance1 .
  ?pref2 act:hasImportance ?importance2 .
}
```

Explanation: To support the smart home system and/or business logic with the decision which `Preference` should be realized if they are two or more concurring ones (cf. Listing 5.16) we assigned `LevelsOfImportance` to `Preferences`. Those `LevelsOfImportance` can serve as a first decision support for a potential underlying business logic.

CQ18: Are there any activities stored for a specific actor?

Listing 5.18:

```
PREFIX act: <https://www.auto.tuwien.ac.at/.../ActorOnt.owl#>
SELECT ?activities WHERE {
  act:ActorHumanActor1 act:hasPreferenceProfile/act:hasActivitySchedule
```

```
                                /act:hasScheduledActivity ?activities .
}
```

Explanation: As an addition to `Preferences`, `Actors` can define `Activities` which contain a set of `Preferences` that shall be active whenever this `Activity` is performed. Listing 5.18 exemplifies a query which accesses all `Activities` defined by `ActorHumanActor1`.

CQ19: What preferences are involved in the *Zumba* activity of a specific actor?

Listing 5.19:

```
PREFIX act: <https://www.auto.tuwien.ac.at/.../ActorOnt.owl#>
SELECT ?preferences WHERE {
 act:LR1HA1_ZumbaActivity act:hasPreferenceProfile
                             /act:hasPreference ?preferences .
}
```

Explanation: The earlier mentioned set of `Preferences` which shall be active whenever their related `Activity` is executed, can be accessed via the `Activity's ActivityPreferenceProfile`.

CQ20: What kind of activity is the *Zumba* activity?

Listing 5.20:

```
PREFIX act: <https://www.auto.tuwien.ac.at/.../ActorOnt.owl#>
SELECT ?type WHERE {
 act:LR1HA1_ZumbaActivity a* ?type .
}
```

Explanation: `Activities` are divided into `PassiveActivities` and `NonPassiveActivities`, having additional subcategories such as `SportActivity`, `ReadingActivity` and many more. The short query in Listing 5.20 queries for those types.

CQ21: What are the scheduled preferences and activities of a specific actor for Monday?

Listing 5.21:

```
PREFIX time: <http://www.w3.org/2006/time#>
PREFIX act: <https://www.auto.tuwien.ac.at/.../ActorOnt.owl#>
SELECT DISTINCT ?activities ?preferences WHERE {
  act:ActorHumanActor1 act:hasPreferenceProfile/act:hasActivitySchedule
                        /act:hasScheduledActivity ?activities .
  act:ActorHumanActor1 act:hasPreferenceProfile/act:hasPreferenceSchedule
                        /act:hasPreference ?preferences .
```

```
  ?preferences act:forTime ?time .
  ?time (time:hasBeginning|time:hasEnd)/time:inDateTime ?date .
  ?date time:dayOfWeek ?day .
  ?activities act:forTime ?atime .
  ?atime (time:hasBeginning|time:hasEnd)/time:inDateTime ?adate .
  ?adate time:dayOfWeek ?aday .
  FILTER (?day = time:Monday && ?aday = time:Monday)
}
```

Explanation: Again, to support users with scheduling their `Preferences` and `Activities`, `PreferenceSchedules` and `ActivitySchedules` can be used. The query exemplified in Listing 5.21 illustrates the retrieval of all `Preferences` and `Activities` of `ActorHumanActor1` which are scheduled for Monday.

CQ22: Which unit of measurement does a temperature preference have?

Listing 5.22:

```
PREFIX act: <https://www.auto.tuwien.ac.at/.../ActorOnt.owl#>
PREFIX uom: <http://www.wurvoc.org/vocabularies/om-1.6#>
SELECT DISTINCT ?uom WHERE {
  ?tempPref a/rdfs:subClassOf* act:TemperaturePreference .
  ?tempPref act:hasPreferenceValue ?prefValue .
  ?prefValue act:hasUnitOfMeasure ?uom
}
```

Explanation: To increase the expressivity of our *Actor Preferences Ontology* supporting the collaboration with other related ontologies, we used the UnitsOfMeasurement Ontology[4] to relate `PreferenceValues` via the `hasUnitOfMeasure` object property to their respective unit of measurement, rather than specifying them as strings.

CQ23: What is the standard relative humidity preference value and on which standard is it based on?

Listing 5.23:

```
PREFIX act: <https://www.auto.tuwien.ac.at/.../ActorOnt.owl#>
SELECT ?prefValue ?standard WHERE {
  act:StandardRelativeHumidityPreference act:hasPreferenceValue ?prefValue.
  ?prefValue rdfs:comment ?standard .
}
```

Explanation: As already mentioned in Listings 5.5 and 5.6 we based our `StandardPreferences` on `PreferenceValues` proposed by 3^{rd} party institutions. The information of the actual standard a certain `StandardPreferenceValue` is based on, is stored within the `rdfs:comment` annotation property.

[4] `http://www.wurvoc.org/vocabularies/om-1.6/`

CHAPTER 6

OWL Reasoner Evaluation

Contents

In the present chapter, we will briefly introduce selected state-of-the-art OWL reasoners 6.1 and follow up with a discussion of planned evaluation tasks 6.2 which chosen reasoners have to perform. We continue by giving a detailed description of the achieved evaluation results 6.5 before we conclude the present chapter by summarizing the gained insights based on the evaluation we have carried out 6.6.

6.1 Selected OWL Reasoners

Pellet [71, 84] *Pellet* was introduced in 2002 and is one of the most popular representatives of OWL reasoners and thus widely used. It is based on tableaux algorithms developed for expressive Description Logics [44], supports all OWL2 profiles, and contains optimizations for incremental reasoning, nominals (including inverse properties and cardinality constraints), and conjunctive query answering.

FaCT++/JFact[1] **[90]** *FaCT++* an OWL reasoner developed in C++ as well as its Java implementation *JFact*, are based on tableaux algorithms like *Pellet* and incorporate a set of optimization strategies to decrease the processing time. *FaCT++* is a successor of the 1998 proposed FaCT reasoner [41] and is known to have a limited support of `rdfs:Datatype` definitions, i.e. it only supports those listed in the official datatype maps definition[2].

HermiT [42, 82] *HermiT* was introduced in 2008 and was one of the first OWL reasoners which were based on a *hypertableau calculus* that allows for a much more efficient reasoning than any other previously-known algorithm. It was especially developed to be used on very complex and large ontologies (e.g. within the biomedical domain) and offers interfaces to the Java *OWLAPI* as well as Protégé.

TrOWL [88] A rather new OWL reasoner which follows a completely different reasoning approach is the *Tractable reasoning infrastructure for OWL 2* called *TrOWL*. *TrOWL* supports the expressiveness of OWL2 by using language/profile transformations, i.e. it performs semantic approximation to transform OWL2-DL ontologies into OWL2-QL ones for query answering and into OWL2-EL ontologies for TBox and ABox reasoning. Additionally, *TrOWL* is one of the only OWL reasoners that supports stream reasoning.

6.2 Evaluation Tasks

To represent a certain domain of interest as ontology a set of axioms, which each makes a statement that is assumed to be true about the domain, is defined (an extensive definition of OWL2 and its axioms can be found in [65, 66]). Based on these axioms a number of interpretations (of an ontology $\mathcal{O}$) can be derived, which basically contain concrete instantiations/mappings of the domain entities, i.e. it maps object properties to elements of the object domain, data properties to pairs of elements of the object and data domain, individuals to elements of the object domain, etc.

In order to become a model of the ontology, an interpretation must fulfill several conditions which are defined by their respective OWL axioms and can be checked by OWL reasoners [65].

In the following, we will describe four different reasoning tasks the evaluated reasoners had to accomplish before we introduce the chosen evaluation approach and evaluation results.

[2] `http://www.w3.org/TR/owl2-syntax/#Datatype_Maps`

6.2.1 Classification

Definition 2. Let C and D be concepts, R and S be object or data properties, $\mathcal{O}$ an OWL2 Ontology, $C_{\mathcal{O}}$ a set of concepts of an ontology $\mathcal{O}$, and $OP_{\mathcal{O}}$ (respectively $DP_{\mathcal{O}}$) a set of object (data) properties of an ontology $\mathcal{O}$. Then a classification of an ontology $\mathcal{O}$ computes all pairs of classes (C, D) such that $\{C, D\} \subseteq C_{\mathcal{O}}$ and $\mathcal{O} \models C \sqsubseteq D$ and respectively all pairs of object/data properties (R, S) such that $\{R, S\} \subseteq OP_{\mathcal{O}}$ (or $\{R, S\} \subseteq DP_{\mathcal{O}}$) and $\mathcal{O} \models R \sqsubseteq S$ [28].

Classifying an ontology, i.e. an OWL reasoner computes the subsumption hierarchies for concepts and properties of the ontology, is one of the main reasoning tasks any OWL reasoner must be able to fulfill. Based on the characteristics of both the ontology as well as the OWL reasoner two dimensions, i.e. *(i) time efficiency*, and *(ii) quality of results*, must be considered. For example, a reasoner might be the fastest one to finish classifying the ontology but misses some of the correct subsumption relations others were able to infer.

6.2.2 Consistency Check

Definition 3. Let $\mathcal{D}$ be a datatype map, $\mathcal{V}$ a vocabulary over $\mathcal{D}$, and $\mathcal{O}$ be an OWL2 Ontology. Then $\mathcal{O}$ is considered to be consistent (or satisfiable) with respect to $\mathcal{D}$ if a model of $\mathcal{O}$ with respect to $\mathcal{D}$ and $\mathcal{V}$ exists. [65]

An ontology is considered to be consistent if there exists at least one valid model of the ontology, thus makes the task of checking the consistency of an ontology usually much faster compared to others.

Remark: An unsatisfiable class does not imply an inconsistent ontology if there is at least one satisfiable model of the ontology (such an ontology is considered to be *incoherent*). On the contrary, all classes of an inconsistent ontology are unsatisfiable.

6.2.3 Type Inference of Individuals

One major advantage of using ontologies as knowledge base is the capability to model the explicit semantics of concepts, i.e. define characteristics an individual must have in order to belong to that respective concept (e.g. a `SatisfiedHumanActor` is a `HumanActor` which has a `LevelOfSatisfaction` of `Satisfied` or `VerySatisfied`; cf. Section 3.5 for a more detailed discussion).

The related reasoning task computes all possible types for individuals of an ontology and again two dimensions, i.e. *(i) time efficiency*, and *(ii) quality of results* must be considered.

6.2.4 Query Answering

One drawback of querying ontologies without reasoning support is that information a reasoner might be able to infer (i.e. type inference, inverse property relations, ...) cannot be utilized per se. For example, if we want to accomplish the simple task of retrieving all non-scheduled activities, a SPARQL query utilizing reasoning results can be as short as shown in Listing 6.1 or much more complicated as illustrated in Listing 6.2 if reasoning support is not available.

Listing 6.1: Query with reasoning support

```
PREFIX act: <https://www.auto.tuwien.ac.at/.../ActorOntology.owl#>
SELECT DISTINCT ?activity
WHERE {
  ?activity a act:NonScheduledActivity .
}
```

Listing 6.2: Query without reasoning support

```
PREFIX act: <https://www.auto.tuwien.ac.at/.../ActorOntology.owl#>
SELECT DISTINCT ?activity
WHERE {
  ?activity a/rdfs:subClassOf* act:Activity.
  {
    ?activity act:hasPreferenceProfile/act:hasPreference ?preference.
  }
  UNION
  {
    ?activity2 a/rdfs:subClassOf* act:Activity.
    ?activity2 act:forTime ?timeInfo .
    ?activity2 act:forActivity ?activity .
  FILTER(?activity != ?activity2)
  }
}
```

Since queries which rely on reasoning support require some pre-processing done by the respective reasoning system to be able to retrieve any results at all, we defined three different query tasks (cf. Listings 6.3, 6.4, and 6.5) to be carried out on the *Actor Preferences Ontology* and measured the performance of the tested OWL reasoners to fulfill these tasks.

Listing 6.3: Which non-scheduled activities contain preferences of high importance?

```
NonScheduledActivity and
    hasPreferenceProfile some
        (hasPreference some (hasImportance value HighImportance))
```

Explanation: The particular difficulty of this query lies in the definition of `NonScheduledActivity` which has to be derived in the first place, i.e. individuals of type `Activity` must be typed to `NonScheduledActivity` if they fullfil certain conditions before the query can be executed.

Listing 6.4: Which preferences end on Monday?

```
UserDefinedPreference and
    forTime some (
        (hasEnd some (inDateTime some (dayOfWeek value Monday)))
         or
        (hasBeginning some (inDateTime some (dayOfWeek value Monday)))
    )
```

Explanation: Again, reasoners have to define individuals of type `Preference` as `UserDefinedPreference` based on rather difficult constraints (i.e. they have to follow several paths of inverse properties to reach a potential owner of a property), before they are able to further process the query. Apart from that, external entities of the OWL-TIME ontology must be processed.

Listing 6.5: Which preference values were defined by ActorHumanActor1?

```
PreferenceValue and
    isPreferenceValueOf some (
        isPreferenceOf some (
            (isPreferenceScheduleOf some (
                isPreferenceProfileOf value ActorHumanActor1)
            ) or
            (isPreferenceProfileOf some (
                isActivityOf some (
                    isScheduledActivityOf some (
                        isPreferenceProfileOf value ActorHumanActor1)
                    )
                )
            )
        )
)
```

Explanation: Some reasoners have difficulties to resolve inverse property relations, which should be tested using the present query that only consists of not explicitly defined inverse properties.

6.3 Evaluation System

The system specifications of the machine that has been used to perform the reasoning tasks are listed in Table 6.1. Note that this machine was not particularly optimized for evaluation purposes, thus evaluation results might differ from those perceived by others.

System Specifications	
Processor	2x Dual-Core i5-4200U
Processor Details	1.60 GHz
RAM	12 GB
Allocated RAM	4 GB
Operating System	Windows 7 - 64 Bit

Table 6.1: System specifications of the evaluation machine

6.4 Evaluation Approach

All OWL reasoners which we have chosen to evaluate based on reasoning tasks related to the *ThinkHome* system, offer an interface to *OWLAPI 3.5.0*[3] implemented in Java. Thus, we developed a reasoner testing framework which performs the previously introduced reasoning tasks using the respective OWL reasoner defined as parameter as exemplified in Listing 6.6 which contains the method, responsible for computing type inferences.

```
public Set<OWLAxiom> computeTypeInference(OWLReasoner reasoner) {
  InferredClassAssertionAxiomGenerator classAssertionGenerator = new
      InferredClassAssertionAxiomGenerator();
  OWLOntologyManager manager = this.ont.getOWLOntologyManager();

  // Start measuring the computation time
  long start = System.currentTimeMillis();
  Set<? extends OWLAxiom> resultAxioms = new HashSet();
  try {
    // start computing type inferences
    reasoner.precomputeInferences(new InferenceType[] { InferenceType.
        CLASS_ASSERTIONS });
    // generate axioms for inferred class assertions
    resultAxioms = classAssertionGenerator.createAxioms(manager,
        reasoner);
  } catch (InconsistentOntologyException e) {
    // return Nothing in case of inconsistent ontology
```

[3]http://owlapi.sourceforge.net/

```
      OWLDataFactory factory = manager.getOWLDataFactory();
      resultAxioms = Collections.singleton(factory.getOWLSubClassOfAxiom(
          factory.getOWLThing(), factory.getOWLNothing()));
    }
    // End measuring the computation time
    long end = System.currentTimeMillis();
    // return the inferred axioms
    return (Set<OWLAxiom>) resultAxioms;
  }
```

Listing 6.6: Method computing all type inferences and returning them as set of OWLAxioms.

For evaluation purposes, we store the time a reasoner needs to complete its task together with the results it produces. This is primarily based on the fact that you cannot measure the performance of an OWL reasoner solely based on the time it needs to complete a tasks but additionally on the quality and completeness of the calculated/inferred results (e.g. a reasoner might classify an ontology twice as fast as all other reasoners but misses some of the implications the ontology would offer which other reasoners could potentially infer).

To provide representative evaluation results, every reasoner had to perform every reasoning task 100 times before we calculate the average time taken for each reasoner and gather the computed results.

6.5 Evaluation Results

In addition to the *Actor Preferences Ontology*, we focused on three different ontologies of the *ThinkHome* system which are related to (i.e. are used within) the *Actor Preferences Ontology* for our reasoner evaluation, namely: *Energy & Resources Ontology (ero)*, *User Behavior & Building Processes Ontology (ppo)*, and *Architecture & Building Physics Ontology (gbo)* [52]. Each of these ontologies has slightly different characteristics (cf. Table 6.2) and thus is either easier or more difficult to be reasoned over.

In order to obtain representative reasoning results we have used those versions of the ontologies which contain already a number of sample instantiations and thus emulate a real world scenario.

6.5.1 Actor Preferences Ontology (:act)

Discussion of the Results

Pellet: As one of the most matured OWL reasoners available, *Pellet* was able to accomplish all reasoning tasks correctly and to retrieve all expected query results for Q1-Q3.

	Number of ...			
Ontology	Concepts	Data Properties	Object Properties	Individuals
gbo	232	207	233	427
ero	253	39	71	327
ppo	156	9	29	532
act	69	6	28	484

Table 6.2: Characteristics of used ontologies.

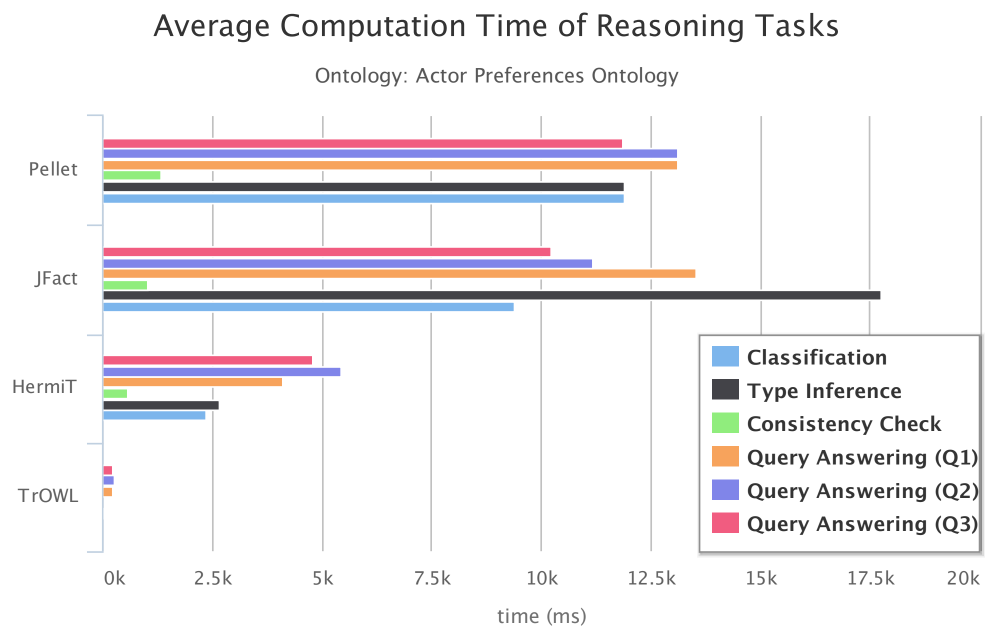

Figure 6.1: Average processing time of each reasoning tasks for all reasoners.

Reasoner	**Classification**	**Type Inference**	**Consistency Check**	**Query Answering**		
				Q1	**Q2**	**Q3**
Pellet	11913 ms	11883 ms	1329 ms	13132 ms	13119 ms	11872 ms
JFact	9390 ms	17749 ms	1039 ms	13536 ms	11196 ms	10215 ms
HermiT	2364 ms	2643 ms	4109 ms	4054 ms	5443 ms	4795 ms
TrOWL	5 ms	24 ms	1 ms	221 ms	256 ms	218 ms

Table 6.3: Average processing time of each reasoning tasks for all reasoners.

JFact: One major drawback of FaCT++/JFact is the limited support of datatypes as they only accept those which are part of the OWL2 datatypes map[4]. As a result, custom `rdfs:Datatype` definitions as e.g. used within the OWL-TIME ontology lead to errors and the termination of the reasoning process.

HermiT: Like *Pellet*, *HermiT* was able to fulfill all reasoning as well as query tasks correctly whilst at the same time being 3-5 times faster as *Pellet*.

TrOWL: Two major shortcomings of *TrOWL* regarding the quality and correctness of its inferred results were identified, namely: *(i) inability to process value range restrictions* and *(ii) missing support of type inference using inverse property relations*. *TrOWL* was not able to process queries containing any `is...Of` relations which are solely defined as the inverse of their related property (e.g. `(ii) Preference and (isPreferenceOf min 1 StandardPreferenceProfile` to define standard preferences) or queries like `(i) Age and (hasYears some int[>=66]`, thus was not able to retrieve results for Q2 and Q3. Besides those shortcomings, *TrOWL* had remarkable results regarding time efficiency being up to 500 times faster than the second fastest reasoner *HermiT*.

6.5.2 Energy & Resources Ontology (:ero)

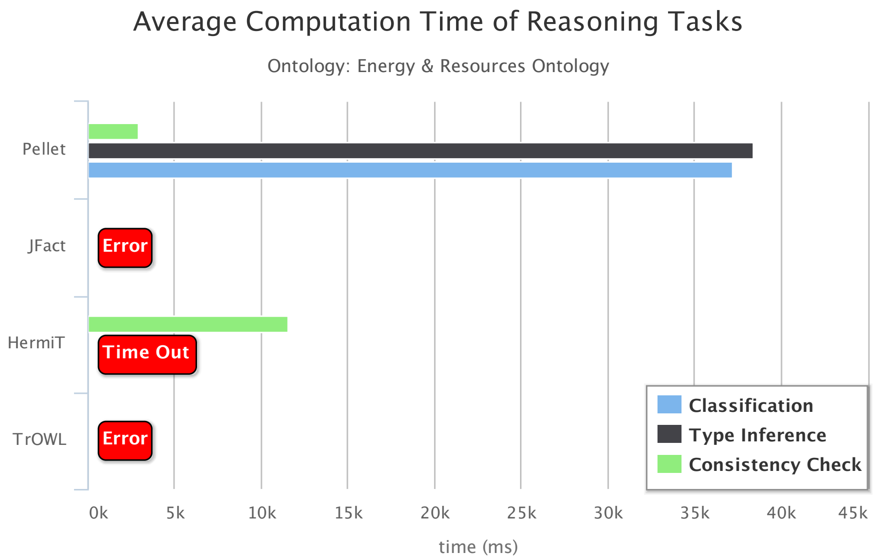

Figure 6.2: Average processing time of each reasoning tasks for all reasoners.

[4] `http://www.w3.org/TR/owl2-syntax/#Datatype_Maps`

Discussion of the Results

Pellet: *Pellet* was able to correctly and completely infer all required inferences, thus finishing all reasoning tasks.

JFact: Threw exception due to problems with customly defined `rdfs:Datatypes`.

HermiT: *HermiT* was only able to check if the present ontology is consistent but failed in computing the other tasks in a timely manner (i.e. under 10 hours).

TrOWL: Threw exception due to problems with customly defined `rdfs:Datatypes`.

6.5.3 User Behavior & Building Processes Ontology (:ppo)

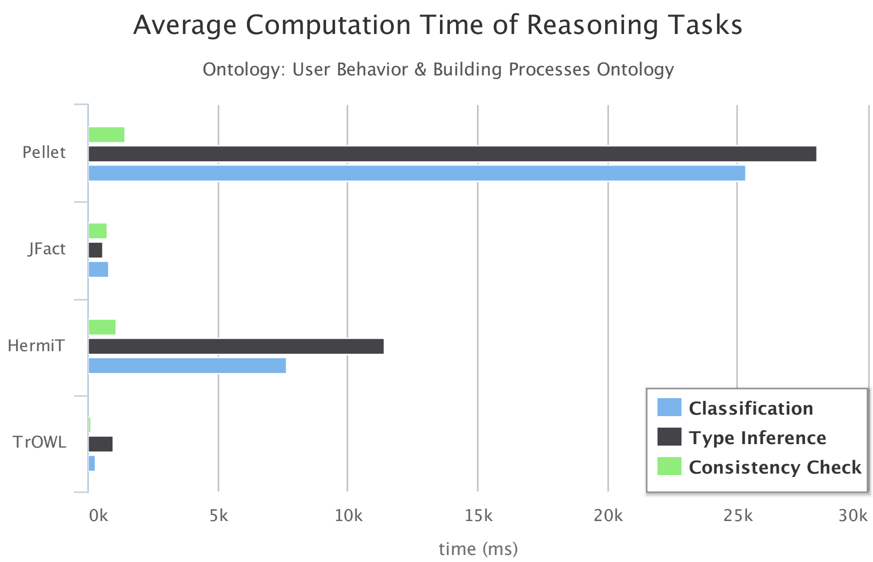

Figure 6.3: Average processing time of each reasoning tasks for all reasoners.

Reasoner	Classification	Type Inference	Consistency Check
Pellet	37207 ms	38385 ms	2949 ms
JFact	✗	✗	✗
HermiT	time out	time out	11551 ms
TrOWL	✗	✗	✗

Table 6.4: Average processing time of each reasoning tasks for all reasoners.

Discussion of the Results

Pellet: *Pellet* was able to correctly and completely infer all required inferences, thus finishing all reasoning tasks.

JFact: *JFact* was able to finish all tasks but it was not able to compute any results. Although this might be related to problems with customly defined `rdfs:Datatypes` this cannot be guaranteed and would require further investigations.

HermiT: *HermiT* was able to finish all reasoning tasks and derived the same inferences as *Pellet*, whilst at the same time being 2-3 times faster than *Pellet*.

TrOWL: We tested *TrOWL* on both our Java implementation as well as directly in Protégé where it was only able to run properly in the latter case. The results presented in Table 6.5 as well as Figure 6.3 for *TrOWL* are based on the computation time provided by Protégé and therefore might differ from potential results which we would have been able to obtain from our reasoner evaluation framework if *TrOWL* would not have thrown an internal `NullPointerException`.

6.5.4 Architecture & Building Physics Ontology (:gbo)

Discussion of the Results

Pellet: Again, only *Pellet* was able to compute results whilst at the same time being the only freely available OWL reasoner supporting SWRL Rules.

Reasoner	Classification	Type Inference	Consistency Check
Pellet	25305 ms	28055 ms	1409 ms
JFact	803 ms	573 ms	764 ms
HermiT	7671 ms	11409 ms	1083 ms
TrOWL	284 ms	951 ms	129 ms

Table 6.5: Average processing time of each reasoning tasks for all reasoners.

Reasoner	Classification	Type Inference	Consistency Check
Pellet	23376 ms	23494 ms	4922 ms
JFact	✗	✗	✗
HermiT	time out	time out	time out
TrOWL	✗	✗	✗

Table 6.6: Average processing time of each reasoning tasks for all reasoners.

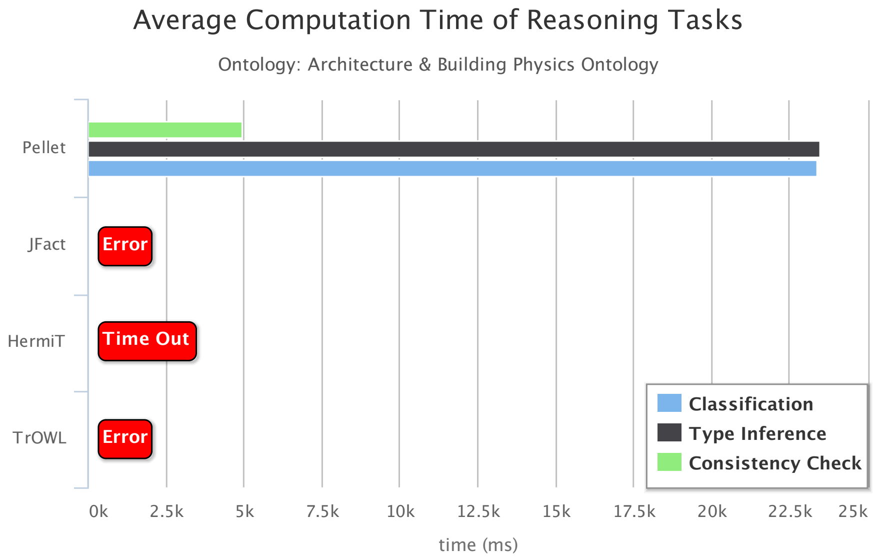

Figure 6.4: Average processing time of each reasoning tasks for all reasoners.

JFact: Threw exception due to problems with customly defined `rdfs:Datatypes`.

HermiT: Although *HermiT* provides very basic support of SWRL rules (i.e. currently no support of built-ins), they were removed for the evaluation, since only *Pellet* would have been able to process them properly. *HermiT* was not able to compute any results in a reasonable time[5].

TrOWL: Threw exception due to problems with customly defined `rdfs:Datatypes`.

6.6 Conclusion

The evaluation conducted within this chapter offers several interesting insights and information which might be helpful when choosing an appropriate OWL reasoner for performing reasoning tasks. It has been shown that although single reasoners might perform very well on specific reasoning tasks which are performed on ontologies having specific characteristics, they might have a very bad performance (in both *time efficiency* as well as *quality of reasoning results*) for reasoning over ontologies with slightly different features.

Generally speaking, especially for the domain of an ontology-based knowledge base which integrates a large variety of different ontologies, a stable OWL reasoner, i.e. one that is able to fulfill all reasoning tasks on a large variety of different ontologies,

[5]We aborted the reasoning process after 10 hours.

should always be preferred over one that cannot guarantee the correctness and to a certain degree completeness of its results. Of course, this *stableness* usually comes in hand with a decrease of time efficiency.

Regarding the evaluated OWL reasoners several key characteristics were derivable, namely:

Pellet: The most stable OWL reasoner, with extensive reasoning support including custom `rdfs:Datatype` definitions and SWRL rules with built-ins, but unfortunately it was also the slowest one amongst all evaluated reasoners.

FaCT++/JFact: The most unstable OWL reasoner which was only able to properly process one of the tested ontologies and even that quite slowly.

HermiT: A very promising OWL reasoner which (in case it does not get stuck in a time-out) usually performs reasoning tasks 3-5 times faster than *Pellet*, whilst at the same time providing equally good reasoning results and even supporting basic SWRL rules.

TrOWL: Unfortunately, *TrOWL* has similar shortcomings in terms of processing custom `rdfs:Datatype` definitions as *FaCT++/JFact* and inverse property relations, thus, it was not able to process all tested ontologies properly. Besides these shortcomings, if *TrOWL* was able to process an ontology it was up to 500 times faster than *HermiT*.

CHAPTER 7

Conclusion

In the present thesis, we describe the concept of smart homes in general by introducing the benefits of an ontology-based smart home system, discuss various ontology development approaches, define a comprehensive ontology which is capable of persisting information about actors and their preferences, and give an evaluation of current state-of-the-art OWL reasoners based on ontologies of the *ThinkHome* system.

After motivating the necessity of an ontology which covers the domain of actors and their preferences within a smart home domain and discussing the concept of a smart home in general, we discuss several different methodologies for developing ontologies and emphasize our choice of using *METHONTOLOGY* by analysing all of the said approaches taking their advantages and disadvantages regarding our requirements into account.

In the main chapter of the present thesis, we discuss the development process of the *Actor Preferences Ontology* structured in a way that follows the development steps proposed by *METHONTOLOGY*. To summarize, the *Actor Preferences Ontology* represents a convenient way for residents of a smart home to persist their personal information, to store information about their general preferences and preferences for activities, as well as to schedule preferences and activities. To achieve that, the *Actor Preferences Ontology* contains eleven main concepts, namely: `Activity`, `ActivitySchedule`, `Actor`, `Age`, `Gender`, `LevelOfImportance`, `LevelOfSatisfaction`, `Preference`, `PreferenceProfile`, `PreferenceSchedule`, and `PreferenceValue`. Instantiations of these concepts can be used to precisely model actors of a smart home system together with their preferences and activities which are grouped within logically coherent preference profiles and might be scheduled within preference schedules. Taking advantage of the reasoning capabilities an ontology-based knowledge base comes with, we exemplify the usability of our ontology by answering a large set of competency questions with SPARQL.

As the previously mentioned reasoning capabilities are of major importance to any ontology-based system, the choice of the right reasoner for carrying out these reasoning tasks is no less important. For that purpose, we evaluated current state-of-the-art ontology reasoners based on different reasoning tasks which have to be performed on ontologies of the *ThinkHome* system and discussed their achieved results.

7.1 Further Work

Concerning future work that could be done to improve the *Actor Preferences Ontology* primarily revolves around *(i) improving reasoning capabilities it provides*, and *(ii) extending its predefined set of concepts*:

Improving Reasoning Capabilities: Although the *Actor Preferences Ontology* was built having extensive reasoning support in mind not all possibilities Semantic Web technologies would offer in that regard were considered. One example of such a technology would be the *Semantic Web Rule Language (SWRL)* [43] which offers the possibility to define general valid and more complex rules which would have been difficult to describe using OWL and RDFS alone.

Listing 7.1: SWRL rule that assigns scheduled preferences to type `ErrorClass` if their end time starts before their start time.

```
Preference(?x), forTime(?x,?y), hasBeginning(?y,?B), hasEnd(?y,?E),
inDateTime(?B,?dtB), inDateTime(?E,?dtE), hour(?dtB,?hB), hour(?dtE,?hE),
swrlb:greaterThan(?hB, ?hE) -> ErrorClass(?x)
```

For example, consider the SWRL rule exemplified in Listing 7.1 using Protégé syntax. By using that rule, it is possible to assign individuals of type `Preference` to type `ErrorClass`, which is disjoint with all other concepts in the ontology, if they have a scheduled end time which starts before their scheduled starting time (under the assumption, that preferences can only be scheduled within one day, i.e. their timeframe does not span over midnight).

Extending Predefined Concepts: Regarding the extension of predefined concepts three concepts especially stand out:

- **`Activities`** - We currently only cover three different types of `NonPassiveActivities` and five different types of `PassiveActivities`, which should have served as an example on how to define activities but definitely do not cover the area of activities completely.
- **`Preferences`** - Although we already conceptualized a large amount of possible preferences which could be set in a smart home environment, there definitely exist many more which were not already considered.

`StandardPreferences` - Similar to `Activities` we only defined standard preference values for a small set of preferences, which should of course be extended if the *ThinkHome* system would be deployed in practice.

Introducing and Describing `SystemActors` As already briefly mentioned in previous sections (cf. Section 5.2.2), the concept of `SystemActors` which represent agents of the multi-agent system is not thoroughly defined in the present thesis. For future work, we plan to investigate and extend this concept extensively to provide an even better integration of the MAS and corresponding KB.

APPENDIX A

Conceptualization Tables

In the present appendix we *'out-sourced'* some tables of Section 5.2 to be able to stick to a concise representation regime, namely:

Concept Dictionary Table: cf. Section 5.2.4 for more details.

- `Activity` A.1,
- `ActivitySchedule` A.2,
- `Actor` A.3,
- `Age` A.4,
- `LevelOfImportance` A.6,
- `LevelOfSatisfaction` A.7,
- `Preference` A.8,
- `PreferenceProfile` A.9,
- `PreferenceSchedule` A.10,
- `PreferenceValue` A.11;

Binary Relation Tables: A.12; cf. Section 5.2.5 for more details.

Instance Attribute Tables: A.13; cf. Section 5.2.6 for more details.

Class Attribute Tables: cf. Section 5.2.7 for more details.

- `Actor` A.14
- `Age` A.15
- `Preference` A.16

`PreferenceProfile` A.17

`PreferenceSchedule` A.18

Instance Tables: A.19; cf. Section 5.2.8 for more details.

Concept Name	Instance Attributes	Relations
CleaningActivity	–	hasPreferenceProfile
CookingActivity	–	hasPreferenceProfile
NonPassiveActivity	–	hasPreferenceProfile
NonScheduledActivity	–	isActivityOf, hasPreferenceProfile
PassiveActivity	–	hasPreferenceProfile
ReadingActivity	–	hasPreferenceProfile
ScheduledActivity	–	forTime, forActivity
SleepingActivity	–	hasPreferenceProfile
SportActivity	–	hasPreferenceProfile
WakeUpActivity	–	hasPreferenceProfile
WatchingTVActivity	–	hasPreferenceProfile
WritingActivity	–	hasPreferenceProfile

Table A.1: Concept Dictionary for `Activity`

Concept Name	Instance Attributes	Relations
ActivitySchedule	–	isActivityScheduleOf

Table A.2: Concept Dictionary for `ActivitySchedule`

Concept Name	Instance Attributes	Relations
Actor	–	hasPreferenceProfile
AgedHumanActor	hasName	hasAge, hasGender, hasPreferenceProfile
FemaleHumanActor	hasName	hasGender, hasPreferenceProfile
HumanActor	hasName	hasAge, hasGender, hasSatisfactionLevel, hasPreferenceProfile
MaleHumanActor	hasName	hasGender, hasPreferenceProfile
MatureHumanActor	hasName	hasAge, hasGender, hasPreferenceProfile
SatisfiedHumanActor	hasName	hasSatisfactionLevel, hasPreferenceProfile
SystemActor	hasID	hasAge
UnsatisfiedHumanActor	hasName	hasSatisfactionLevel, hasPreferenceProfile
UserSystemActor	hasName	hasAge, represents, hasPreferenceProfile
YoungHumanActor	hasName	hasAge, hasGender, hasPreferenceProfile

Table A.3: Concept Dictionary for Actor

Concept Name	Instance Attributes	Relations
AdvancedHumanActorAge	hasYears	–
Age	–	–
HumanActorAge	hasYears	–
MatureHumanActorAge	hasYears	–
SystemActorAge	hasHours	–
YoungHumanActorAge	hasYears	–

Table A.4: Concept Dictionary for Age

Concept Name	Instances	Relations
Gender	Female Male	-

Table A.5: Concept Dictionary for Gender

Concept Name	Instance Attributes	Relations

`LevelOfImportance`	`LowImportance`
	`AverageImportance`
	`HighImportance`
		-

Table A.6: Concept Dictionary for `LevelOfImportance`

Concept Name	Instance Attributes	Relations

`LevelOfSatisfaction`	`DisSatisfied`
	`BarelySatisfied`
	`Satisfied`
	`VerySatisfied`
		-

Table A.7: Concept Dictionary for `LevelOfSatisfaction`

Concept Name	Relations
AirFlowVelocityPreference	hasImportance, hasPreferenceValue, forSpace, forZone, isPreferenceOf
AirQualityPreference	hasImportance, hasPreferenceValue, forSpace, forZone, isPreferenceOf
AirVentilationPreference	hasImportance, hasPreferenceValue, forSpace, forZone, isPreferenceOf
ApplianceCentricPreference	controlsAppliance, hasImportance, hasPreferenceValue, forSpace, forZone, isPreferenceOf
ApplicationCentricPreference	hasImportance, hasPreferenceValue, forSpace, forZone, isPreferenceOf, usesApplication
BlindsPreference	hasImportance, hasPreferenceValue, forSpace, forZone, isPreferenceOf
ComfortTemperaturePreference	hasImportance, hasPreferenceValue, forSpace, forZone, isPreferenceOf
DiswasherPreference	hasImportance, hasPreferenceValue, forSpace, forZone, isPreferenceOf
DryerPreference	hasImportance, hasPreferenceValue, forSpace, forZone, isPreferenceOf
LampPreference	hasImportance, hasPreferenceValue, forSpace, forZone, isPreferenceOf
LightingLevelPreference	hasImportance, hasPreferenceValue, forSpace, forZone, isPreferenceOf
PresencePreference	hasImportance, hasPreferenceValue, forSpace, forZone, isPreferenceOf
RelativeHumidityPreference	hasImportance, hasPreferenceValue, forSpace, forZone, isPreferenceOf
ScheduledPreference	hasImportance, hasPreferenceValue, forSpace, forTime, forZone, isPreferenceOf
SetbackTemperaturePreference	hasImportance, hasPreferenceValue, forSpace, forZone, isPreferenceOf
SoundPressureLevelPreference	hasImportance, hasPreferenceValue, forSpace, forZone, isPreferenceOf
StandardPreference	hasImportance, hasPreferenceValue, forSpace, forZone, isPreferenceOf
TemperaturePreference	hasImportance, hasPreferenceValue, forSpace, forZone, isPreferenceOf
UserDefinedPreference	hasImportance, hasPreferenceValue, forSpace, forZone, isPreferenceOf
WashingmaschinePreference	hasImportance, hasPreferenceValue, forSpace, forZone, isPreferenceOf

Table A.8: Concept Dictionary for `Preference`

Concept Name	Relations
ActivityPreferenceProfile	hasActivitySchedule, hasPreference, hasPreferenceSchedule, isPreferenceProfileOf
HumanActorPreferenceProfile	hasActivitySchedule, hasPreference, hasPreferenceSchedule, isPreferenceProfileOf
NonScheduledPreferenceProfile	hasActivitySchedule, hasPreference, hasPreferenceSchedule, isPreferenceProfileOf
ScheduledPreferenceProfile	hasActivitySchedule, hasPreference, hasPreferenceSchedule, isPreferenceProfileOf
StandardPreferenceProfile	hasActivitySchedule, hasPreference, hasPreferenceSchedule, isPreferenceProfileOf

Table A.9: Concept Dictionary for `PreferenceProfile`

Concept Name	Relations
AppliancePreferenceSchedule	hasPreference
ApplicationPreferenceSchedule	hasPreference
PresencePreferenceSchedule	hasPreference
PreferenceSchedule	hasPreference
VisualComfortPreferenceSchedule	hasPreference

Table A.10: Concept Dictionary for `PreferenceSchedule`

Concept Name	Instance Attributes	Relations
BinaryPreferenceValue	hasValue	forState
ContinuousPreferenceValue	hasValue	forState, hasUnitOfMeasure
PreferenceValue	hasValue	forState

Table A.11: Concept Dictionary for `PreferenceValue`

Relation Name	Source Concept	Cardinality	Target Concept	Inverse Relation
controlsAppliance	ApplianceCentricPreference	min 1	ero:Appliance	-
currentlyLocatedIn	HumanActor	exactly 1	gbo:Space **or** gbo:Zone	- -
forActivity	ScheduledActivity	min 1	NonScheduledActivity	isActivityOf
forSpace	Preference	min 1	gbo:Space	-
forState	PreferenceValue	min 1	ero:State	-
forTime	ScheduledPreference ScheduledActivity	min 1	time:TemporalEntity	-
forZone	Preference	min 1	gbo:Zone	-
hasActivitySchedule	PreferenceProfile	only	ActivitySchedule	isActivityScheduleOf
hasAge	Actor	exactly 1	Age	-
hasGender	HumanActor	exactly 1	Gender	-
hasImportance	Preference	exactly 1	LevelOfImportance	-
hasPreference	PreferenceProfile PreferenceSchedule	only min 1	Preference ScheduledPreference	isPreferenceOf
hasPreferenceProfile	Activity Actor	min 1 only	ActivityPreferenceProfile PreferenceProfile	isPreferenceProfileOf
hasPreferenceSchedule	PreferenceProfile	only	PreferenceSchedule	isPreferenceScheduleOf
hasPreferenceValue	Preference	min 1	PreferenceValue	isPreferenceValueOf
hasSatisfactionLevel	HumanActor	exactly 1	LevelOfSatisfaction	-
hasScheduledActivity	ActivitySchedule	min 1	ScheduledActivity	isScheduledActivityOf
hasUnitOfMeasure	ContinuousPreferenceValue	exactly 1	om:Unit_of_measure	-
represents	UserSystemActor	exactly 1	HumanActor	representedBy
usesApplication	HumanActor	exactly 1	Gender	-

Table A.12: Ad-hoc Binary Relation Table

Attribute Name	Concept Name	Value Type	Measurement Unit	Range of Values	Cardinality
`hasName`	`Actor`	`xsd:string`	-	no restriction	(1,1)
`hasYears`	`Age`	`xsd:integer`	$[0, \infty]$	(1,1)	
`hasHours`	`Age`	`xsd:float`	$[0, \infty]$	(1,1)	
`hasID`	`SystemActor`	`xsd:integer`	-	$[0, \infty]$	(1,1)
`hasValue`	`PreferenceValue`	`xsd:Literal`	-	$[-\infty, \infty]$	(1,1)

Table A.13: Instance Attribute Table

Defined Concept	Attribute Name	Cardinality	Values
HumanActor	hasAge	exactly 1	HumanActorAge
	hasGender	exactly 1	Gender
	hasSatisfactionLevel	exactly 1	LevelOfSatisfaction
SystemActor	hasAge	exactly 1	SystemActorAge
	hasID	exactly 1	xsd:integer
UserSystemActor	hasAge	exactly 1	SystemActorAge
	represents	exactly 1	HumanActor
AgedHumanActor	hasAge	exactly 1	AdvancedHumanActorAge
	hasGender	exactly 1	Gender
MatureHumanActor	hasAge	exactly 1	MatureHumanActorAge
	hasGender	exactly 1	Gender
YoungHumanActor	hasAge	exactly 1	YoungHumanActorAge
	hasGender	exactly 1	Gender
SatisfiedHumanActor	hasSatisfactionLevel	value	Satisfied
	or hasSatisfactionLevel	value	VerySatisfied
UnsatisfiedHumanActor	hasSatisfactionLevel	value	BarelySatisfied
	or hasSatisfactionLevel	value	DisSatisfied
FemaleHumanActor	hasGender	value	Female
MaleHumanActor	hasGender	value	Male

Table A.14: Class Attribute Table for concept `Actor`

Defined Concept	**Attribute Name**	**Cardinality**	**Values**
`HumanActorAge`	`hasYears`	`exactly 1`	`xsd:integer`
`SystemActorAge`	`hasHours`	`exactly 1`	`xsd:float`
`YoungHumanActorAge`	`hasYears`	`some xsd:integer`	$[0, 14)$
`MatureHumanActorAge`	`hasYears`	`some xsd:integer`	$[14, 66)$
`AdvancedHumanActorAge`	`hasYears`	`some xsd:integer`	$[66, 120)$

Table A.15: Class Attribute Table for concept `Age`

Defined Concept	Attribute Name	Cardinality	Values
`ApplianceCentricPreference`	`controlsAppliance`	`min 1`	`ero:Appliance`
`ApplicationCentricPreference`	`usesApplication`	`min 1`	`ppo:Application`
`ScheduledPreference`	`forTime`	`min 1`	`time:TemporalEntity`
`StandardPreference`	`isPreferenceOf`	`min 1`	`StandardPreferenceProfile`

`UserDefinedPreference`	`isPreferenceOf`
	`or` `isPreferenceOf`
		`min 1`
		`min 1`
			`HumanActorPreferenceProfile`
			`(isPreferenceScheduleOf`
			`min 1 HumanActorPreferenceProf)`

Table A.16: Class Attribute Table for concept `Preference`

Defined Concept	Attribute Name	Cardinality	Values
`ActivityPreferenceProfile`	`isPreferenceProfile`	`min 1`	`Activity`
`NonScheduledPreferenceProfile`	`hasPreference` **`or`** `isPreferenceProfOf`	`min 1` `min 1`	`Preference` `NonScheduleActivity`
`ScheduledPreferenceProfile`	`hasActSchedule` **`or`** `hasPreferenceSchedule`	`min 1` `min 1`	`ActSchedule` `PreferenceSchedule`

Table A.17: Class Attribute Table for concept `PreferenceProfile`

Defined Concept	Attribute Name	Cardinality	Values
`AppliancePreferenceSchedule`	`hasPreference`	`min 1`	`(ApplianceCentricPreference` **`and`** `ScheduledPreference)`
`ApplicationPreferenceSchedule`	`hasPreference`	`min 1`	`(ApplicationCentricPreference` **`and`** `ScheduledPreference)`
`PresencePreferenceSchedule`	`hasPreference`	`min 1`	`((PresencePreference` **`and`** `ScheduledPreference)`
`VisualComfortPreferenceSchedule`	`hasPreference`	`some`	`(LampPreference` **`or`** `LightingLevelPreference)`

Table A.18: Class Attribute Table for concept `PreferenceSchedule`

Instance Name	Concept Name
Female	Gender
Male	Gender
LowImportance	LevelOfImportance
AverageImportance	LevelOfImportance
HighImportance	LevelOfImportance
DisSatisfied	LevelOfSatisfaction
BarelySatisfied	LevelOfSatisfaction
Satisfied	LevelOfSatisfaction
VerySatisfied	LevelOfSatisfaction

Table A.19: Instance Table

APPENDIX B

Detailed Evaluation Results

In this part of the appendix, detailed results of the measured processing time of reasoning tasks are illustrated.

Actor Preferences Ontology

- **Classification B.1**
- **Consistency Check B.2**
- **Type Inference B.3**
- **Query Answering B.4 B.5**

Energy & Resources Ontology

- **Classification B.7**
- **Consistency Check B.8**
- **Type Inference B.9**

User Behavior & Building Processes Ontology

- **Classification B.10**
- **Consistency Check B.11**
- **Type Inference B.12**

Architecture & Building Physics Ontology

- **Classification B.13**
- **Consistency Check B.14**
- **Type Inference B.15**

Actor Preferences Ontology (:act)

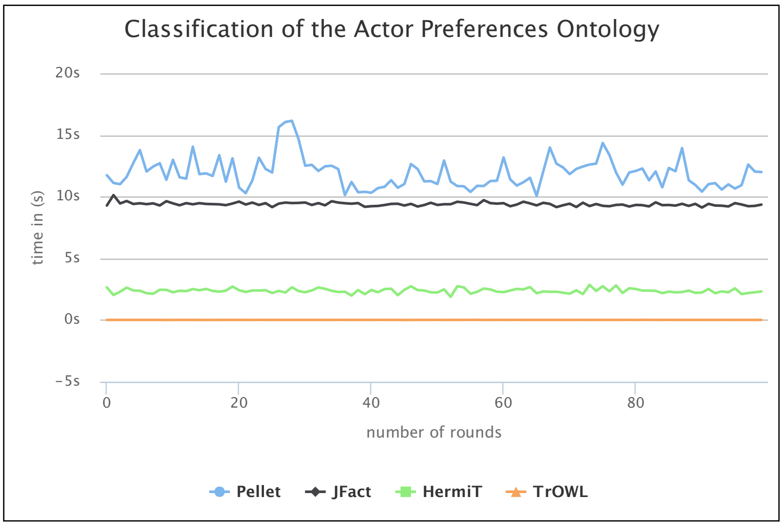

Figure B.1: Comparison of time needed to classify the *Actor Preferences Ontology*.

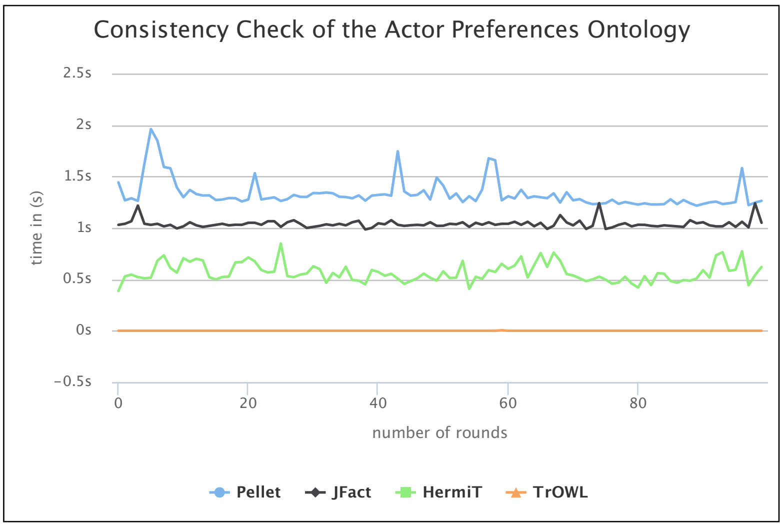

Figure B.2: Comparison of time needed to perform a consistency check for the *Actor Preferences Ontology*.

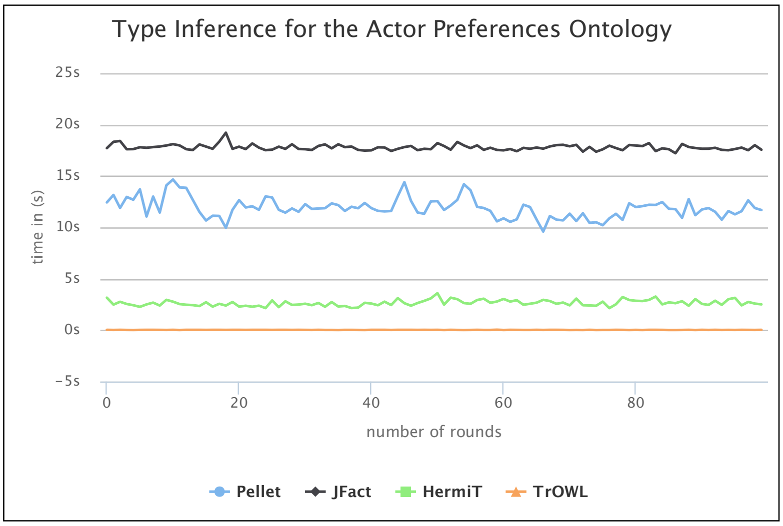

Figure B.3: Comparison of time needed to calculate type inferences for the *Actor Preferences Ontology*.

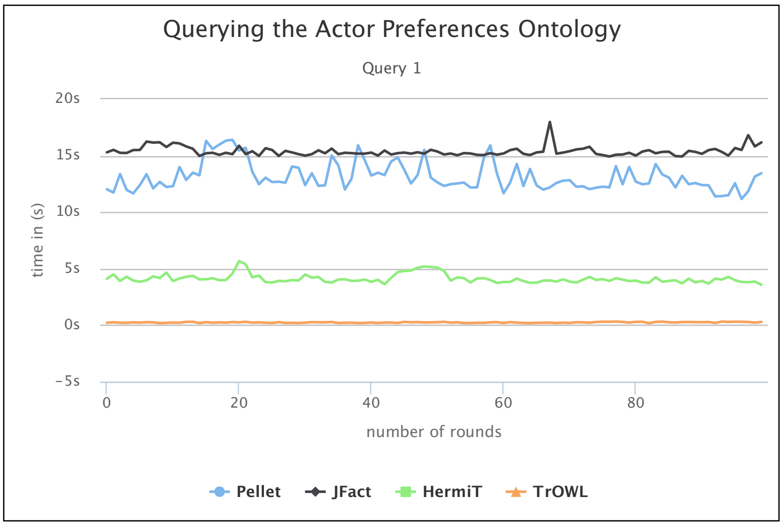

Figure B.4: Comparison of time needed to process query 6.3 on the *Actor Preferences Ontology*.

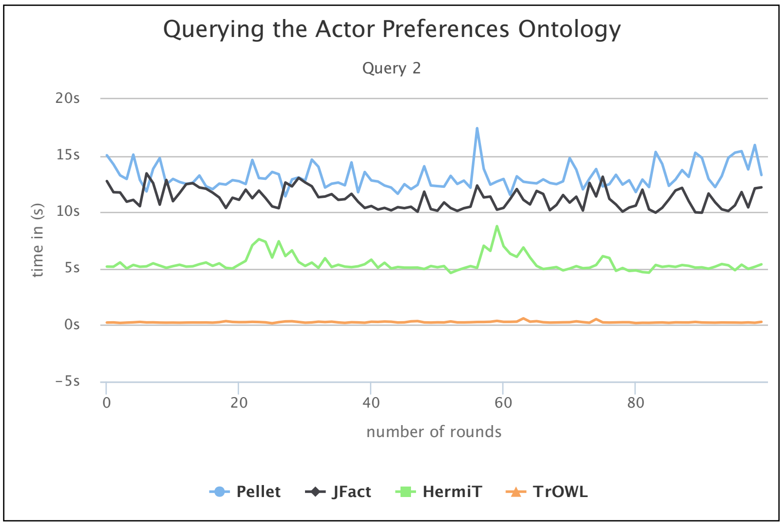

Figure B.5: Comparison of time needed to process query 6.4 on the *Actor Preferences Ontology*.

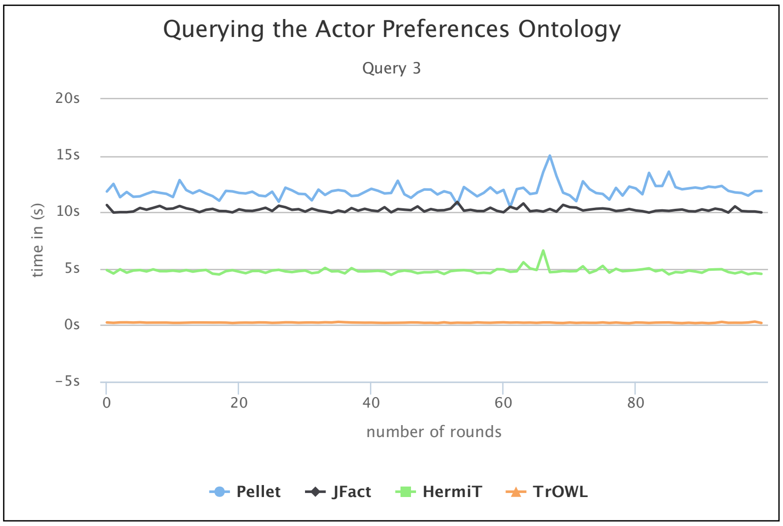

Figure B.6: Comparison of time needed to process query 6.5 on the *Actor Preferences Ontology*.

Energy & Resources Ontology (:ero)

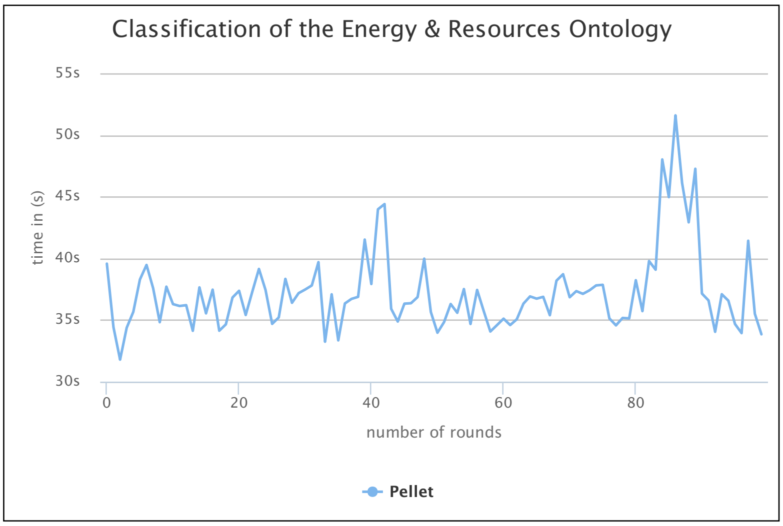

Figure B.7: Comparison of time needed to classify the *Energy & Resources Ontology*.

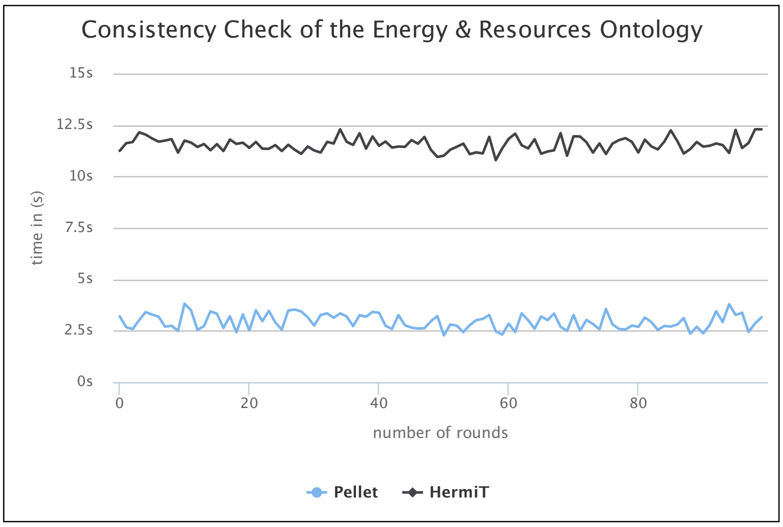

Figure B.8: Comparison of time needed to perform a consistency check for the *Energy & Resources Ontology*.

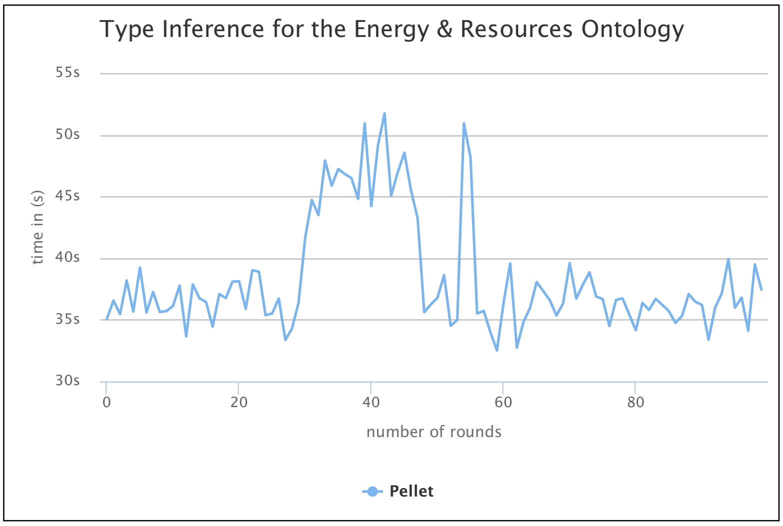

Figure B.9: Comparison of time needed to calculate type inferences for the *Energy & Resources Ontology*.

User Behavior & Building Processes Ontology (:ppo)

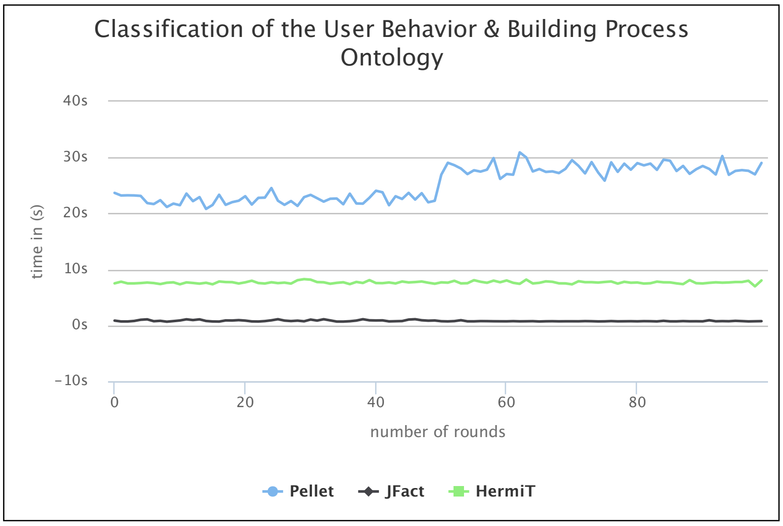

Figure B.10: Comparison of time needed to classify the *User Behavior & Building Processes Ontology*.

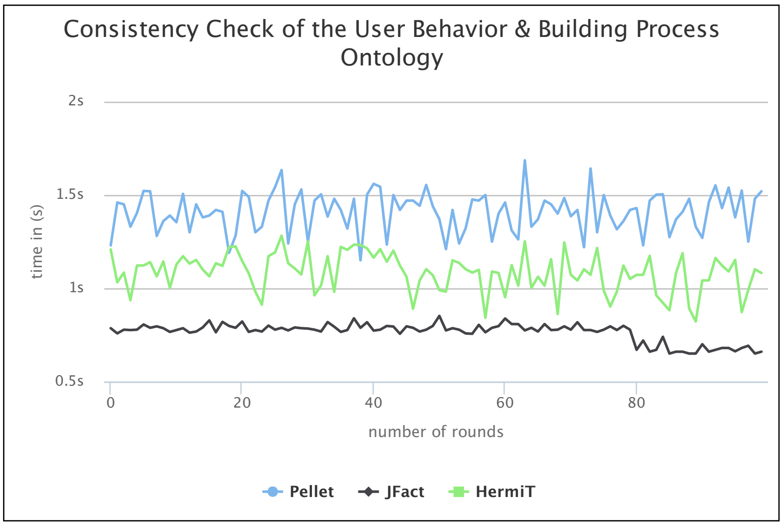

Figure B.11: Comparison of time needed to perform a consistency check for the *User Behavior & Building Processes Ontology*.

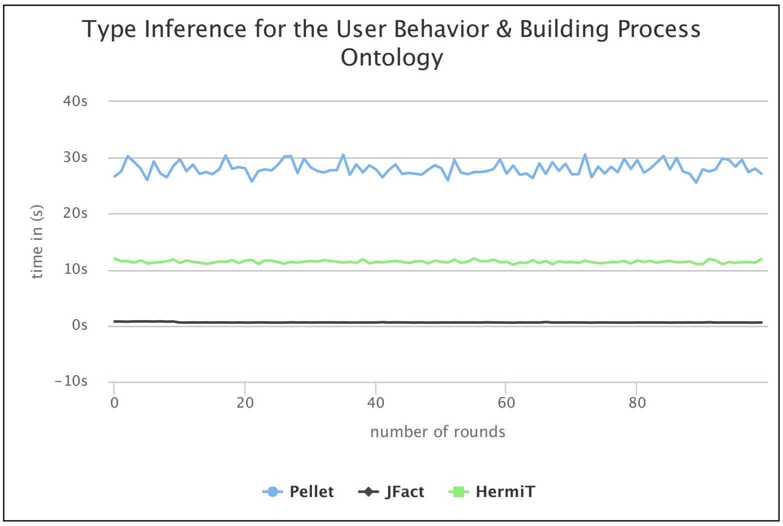

Figure B.12: Comparison of time needed to calculate type inferences for the *User Behavior & Building Processes Ontology*.

Architecture & Building Physics Ontology (:gbo)

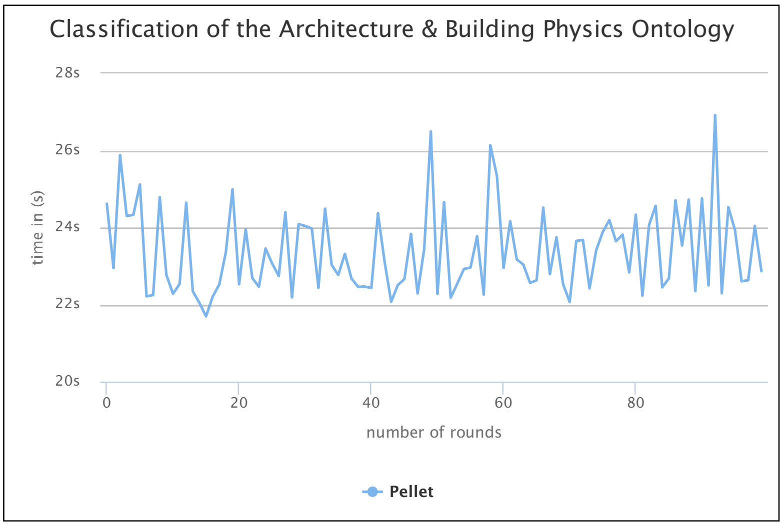

Figure B.13: Comparison of time needed to classify the *Architecture & Building Physics Ontology*.

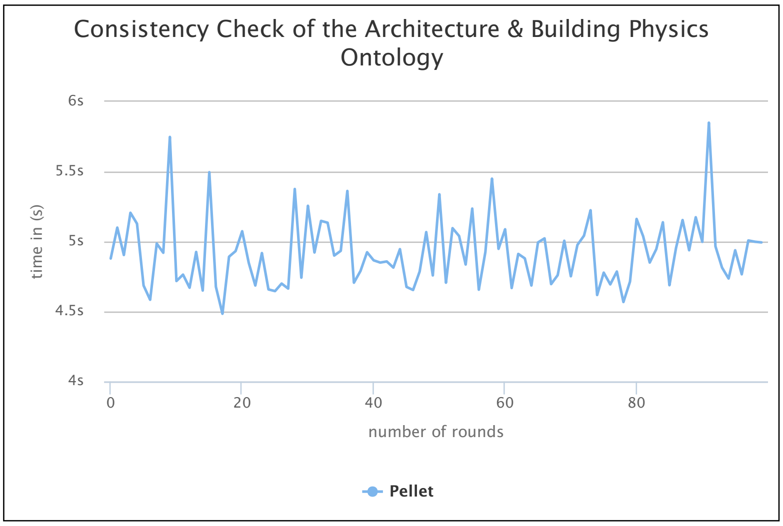

Figure B.14: Comparison of time needed to perform a consistency check for the *Architecture & Building Physics Ontology*.

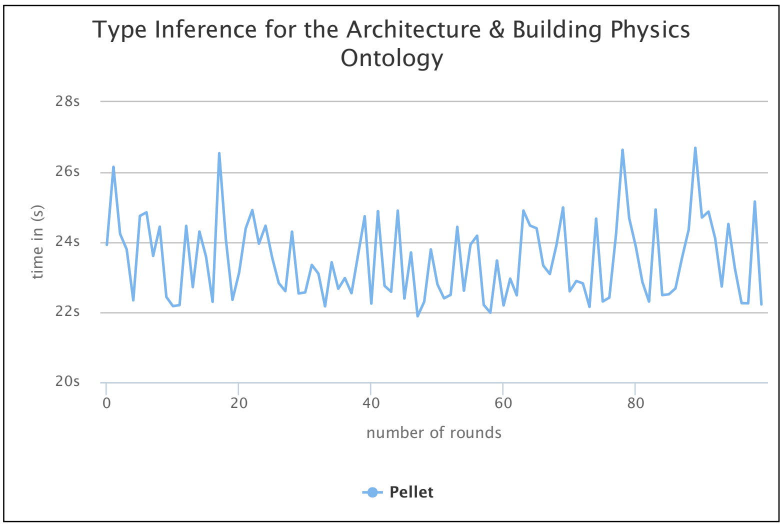

Figure B.15: Comparison of time needed to calculate type inferences for the *Architecture & Building Physics Ontology*.

APPENDIX C

List of Properties

C | F | H | I | R | U

C

controlsAppliance
relates individuals of `ero:Appliance` to individuals of `ApplianceCentricPreference`.
Inverse Property: –

currentlyLocatedIn
relates individuals of `gbo:Zone or gbo:Space` to individuals of `HumanActor`.
Inverse Property: –

F

forActivity
relates individuals of `NonScheduledActivity` to individuals of `ScheduledActivity`.
Inverse Property: `isActivityOf`

forSpace
relates individuals of `gbo:Space` to individuals of `Preference`.
Inverse Property: –

forState
relates individuals of `ero:State` to individuals of `PreferenceValue`.
Inverse Property: –

forTime
relates individuals of `time:TemporalEntity` to individuals of `PreferenceSchedule or ScheduledActivity or ScheduledPreference`.
Inverse Property: –

forZone
relates individuals of `gbo:Zone` to individuals of `Preference`.
Inverse Property: –

H

hasActivitySchedule
relates individuals of `ActivitySchedule` to individuals of `PreferenceProfile`.
Inverse Property: `isActivityScheduleOf`

hasAge
relates individuals of `Age` to individuals of `Actor`.
Inverse Property: –

hasGender
relates individuals of `Gender` to individuals of `HumanActor`.
Inverse Property: –

hasHours
relates individuals of `xsd:float` to individuals of `Age`.
Inverse Property: –

hasID
relates individuals of `xsd:integer` to individuals of `SystemActor`.
Inverse Property: –

hasImportance
relates individuals of `LevelOfImportance` to individuals of `Preference`.
Inverse Property: –

hasName
relates individuals of `xsd:string` to individuals of `Actor`.
Inverse Property: –

hasPreference
relates individuals of `Preference` to individuals of `PreferenceProfile or PreferenceSchedule`.
Inverse Property: `isPreferenceOf`

hasPreferenceProfile
relates individuals of `PreferenceProfile` to individuals of `Activity or Actor`.
Inverse Property: `isPreferenceProfileOf`

hasPreferenceSchedule
relates individuals of `PreferenceSchedule` to individuals of `PreferenceProfile`.
Inverse Property: `isPreferenceScheduleOf`

hasPreferenceValue
relates individuals of `PreferenceValue` to individuals of `Preference`.
Inverse Property: `isPreferenceValueOf`

hasSatisfactionLevel
relates individuals of `LevelOfSatisfaction` to individuals of `HumanActor`.
Inverse Property: –

hasScheduledActivity
relates individuals of `ScheduledActivity` to individuals of `ActivitySchedule`.
Inverse Property: `isScheduledActivityOf`

hasValue
relates individuals of `xsd:Literal` to individuals of `PreferenceValue`.
Inverse Property: –

hasYears
relates individuals of `xsd:integer` to individuals of `Age`.
Inverse Property: –

I

isActivityOf
relates individuals of `ScheduledActivity` to individuals of `NonScheduledActivity`.
Inverse Property: `forActivity`

isActivityScheduleOf
relates individuals of `PreferenceProfile` to individuals of `ActivitySchedule`.
Inverse Property: `hasActivitySchedule`

isPreferenceOf
relates individuals of `PreferenceProfile or PreferenceSchedule` to individuals of `Preference`.
Inverse Property: `hasPreference`

isPreferenceProfileOf

relates individuals of `Activity or Actor` to individuals of `PreferenceProfile`.

Inverse Property: `hasPreferenceProfile`

isPreferenceScheduleOf

relates individuals of `PreferenceProfile` to individuals of `PreferenceSchedule`.

Inverse Property: `hasPreferenceSchedule`

isPreferenceValueOf

relates individuals of `Preference` to individuals of `PreferenceValue`.

Inverse Property: `hasPreferenceValue`

isScheduledActivityOf

relates individuals of `ActivitySchedule` to individuals of `ScheduledActivity`.

Inverse Property: `hasScheduledActivity`

R

representedBy

relates individuals of `SystemActor` to individuals of `HumanActor`.

Inverse Property: `represents`

represents

relates individuals of `HumanActor` to individuals of `SystemActor`.

Inverse Property: `representedBy`

U

usesApplication

relates individuals of `ppo:Application` to individuals of `ApplicationCentricPreference`.

Inverse Property: –

APPENDIX D

List of Classes

A | B | C | D | F | G | H | L | M | N | P | R | S | T | U | V | W | Y

A

Activity

An Activity contains an ActivityPreferenceProfile, which stores Preferences for that specific Activity.

Equivalent To:
–
SubClass Of:
`hasPreferenceProfile min 1 ActivityPreferenceProfile`

ActivityPreferenceProfile

A PreferenceProfile which is used to store Preferences belonging to a specific Activity.

Equivalent To:
`PreferenceProfile and (isPreferenceProfileOf min 1 Activity)`
SubClass Of:
`PreferenceProfile`

ActivitySchedule

An ActivitySchedule clusters several ScheduledActivities together and is part of a HumanActorPreferenceProfile.

Equivalent To:

–

SubClass Of:

`hasScheduledActivity` **min 1** `ScheduledActivity`

Actor

Actors are all individuals (human or system), which interact with the system.

Equivalent To:

–

SubClass Of:

`hasPreferenceProfile` **only** `PreferenceProfile`

AdvancedHumanActorAge

An AdvancedHumanActorAge represents an age of a HumanActor of at least 66.

Equivalent To:

`Age` **and** `(hasYears` **some** `int[>= 66])`

SubClass Of:

`HumanActorAge`

Age

An Age is used to represent ages of Actors of the smart home system.

Equivalent To:

–

SubClass Of:

–

AgedHumanActor

An AgedHumanActor is a HumanActor older than 65.

Equivalent To:

`Actor` **and** `(hasAge` **exactly 1** `AdvancedHumanActorAge)` **and** `(hasGender` **exactly 1** `Gender)`

SubClass Of:

`HumanActor`

AirFlowVelocityPreference

AirFlowVelocityPreferences are used to define Preferences regarding the velocity of the air flow.

Equivalent To:

–

SubClass Of:

`Preference`

AirQualityPreference

AirQualityPreferences are used to define Preferences regarding the quality of the air.

Equivalent To:

–

SubClass Of:

`Preference`

AirVentilationPreference

AirVentilationPreferences are used to define Preferences regarding the amount of ventilated air.

Equivalent To:

–

SubClass Of:

`Preference`

ApplianceCentricPreference

ApplianceCentricPreferences are those Preferences whose realization require the interaction with `ero:Appliances`.

Equivalent To:

`Preference and (controlsAppliance min 1 ero:Appliance)`

SubClass Of:

`Preference`

AppliancePreferenceSchedule

This concept represents schedules that contain at least one ScheduledPreference of type ApplianceCentricPreference.

Equivalent To:

`PreferenceSchedule and (hasPreference min 1 (ApplianceCentricPreference and ScheduledPreference))`

SubClass Of:

`PreferenceSchedule`

ApplicationCentricPreference

Preferences are those Preferences whose realization require the interaction with `ppo:Applications`.

Equivalent To:
`Preference and (usesApplication min 1 ppo:Application)`
SubClass Of:
`Preference`

ApplicationPreferenceSchedule

This concept represents schedules that contain at least one ScheduledPreference of type ApplicationCentricPreference.

Equivalent To:
`PreferenceSchedule and (hasPreference min 1 (ApplicationCentricPreference and ScheduledPreference))`
SubClass Of:
`PreferenceSchedule`

B

BinaryPreferenceValue

A BinaryPreferenceValue contains only 0 or 1 as possible values.

Equivalent To:
–
SubClass Of:
`PreferenceValue`

BlindsPreference

BlindsPreferences are used to define Preferences for blinds.

Equivalent To:
–
SubClass Of:
`Preference`

C

CleaningActivity

A CleaningActivity represents any action related to cleaning.

Equivalent To:

–

SubClass Of:

`NonPassiveActivity`

ComfortTemperaturePreference

ComfortTemperaturePreferences are used to define Preferences regarding the comfort temperature.

Equivalent To:

–

SubClass Of:

`TemperaturePreference`

ContinuousPreferenceValue

A ContinuousPreferenceValue combines a PreferenceValue with its respective unit and is used to represent any type of PreferenceValue except binary ones.

Equivalent To:

–

SubClass Of:

`PreferenceValue` **`and`** `(hasUnitOfMeasure` **`exactly 1`** `om:Unit_of_measure)`

CookingActivity

A CookingActivity represents any action related to cooking.

Equivalent To:

–

SubClass Of:

`NonPassiveActivity`

D

DishwasherPreference

BlindsPreferences are used to define Preferences for dishwashers.

Equivalent To:

–

SubClass Of:

`Preference`

DryerPreference

BlindsPreferences are used to define Preferences for dryers.

Equivalent To:
`-`
SubClass Of:
`Preference`

F

FemaleHumanActor

A FemaleHumanActor is a HumanActor of Gender female.

Equivalent To:
`Actor` **`and`** `(hasGender` **`value`** `Female)`
SubClass Of:
`HumanActor`

G

Gender

The concept Gender represents the sex of a HumanActor.

Equivalent To:
`{Female, Male}`
SubClass Of:
`-`

H

HumanActor

A HumanActor represents a human Actor of the smart home system.

Equivalent To:
`Actor` **`and`** `(hasAge` **`exactly 1`** `HumanActorAge)` **`and`** `(hasGender` **`exactly 1`** `Gender)` **`and`** `(hasSatisfactionLevel` **`exactly 1`** `LevelOfSatisfaction)`
SubClass Of:
`Actor`

HumanActorAge

A HumanActorAge represents an age of a HumanActor in years.

Equivalent To:

`Age and (hasYears exactly 1 int)`

SubClass Of:

`Age`

HumanActorPreferenceProfile

A HumanActorPreferenceProfile is a profile that belongs to exactly one HumanActor.

Equivalent To:

–

SubClass Of:

`PreferenceProfile and (isPreferenceProfileOf exactly 1 HumanActor)`

L

LampPreference

BlindsPreferences are used to define Preferences for lamps.

Equivalent To:

–

SubClass Of:

`Preference`

LevelOfImportance

The concept LevelOfImportance describes the importance of a Preference.

Equivalent To:

`{LowImportance, AverageImportance, HighImportance}`

SubClass Of:

–

LevelOfSatisfaction

The concept LevelOfSatisfaction describes the level of satisfaction of a HumanActor.

Equivalent To:

`{DisSatisfied, BarelySatisfied, Satisfied, VerySatisfied}`

SubClass Of:

–

LightingLevelPreference

LightingLevelPreferences are used to define Preferences regarding the lighting level.

Equivalent To:

–

SubClass Of:

`Preference`

M

MaleHumanActor

A MaleHumanActor is a HumanActor of Gender male.

Equivalent To:

`Actor and (hasGender value Male)`

SubClass Of:

`HumanActor`

MatureHumanActor

An AgedHumanActor is a HumanActor older than 13 and younger than 66.

Equivalent To:

`Actor and (hasAge exactly 1 MatureHumanActorAge) and (hasGender exactly 1 Gender)`

SubClass Of:

`HumanActor`

MatureHumanActorAge

An MatureHumanActorAge represents an age of a HumanActor of at least 14 and at most 65.

Equivalent To:

`Age and (hasYears some int[>= 14]) and (hasYears some int[< 66])`

SubClass Of:

`HumanActorAge`

N

NonPassiveActivity

NonPassiveActivities are used to represent Activities, which involve physical labor.

Equivalent To:

–

SubClass Of:

`Activity`

NonScheduledActivity

An Activity, which is not defined for a specific time period.

Equivalent To:

`Activity` **`and`** `((hasPreferenceProfile` **`min 1`** `NonScheduledPreferenceProfile)` **`or`** `(isActivityOf` **`min 1`** `ScheduledActivity))`

SubClass Of:

`Activity`

NonScheduledPreferenceProfile

A NonScheduledPreferenceProfile is a profile which contains non scheduled Preferences or a profile which is defined for a NonScheduledActivity.

Equivalent To:

`PreferenceProfile` **`and`** `((hasPreference` **`min 1`** `Preference)` **`or`** `(isPreferenceProfileOf` **`min 1`** `NonScheduledActivity))`

SubClass Of:

`PreferenceProfile`

P

PassiveActivity

PassiveActivities are used to represent Activities, which does not involve any physical labor.

Equivalent To:

–

SubClass Of:

`Activity`

Preference

A Preference stores specific PreferenceValues which are realized by applications or appliances for an Actor.

Equivalent To:

–

SubClass Of:
`(forSpace only gbo:Space) and (forTime only time:TemporalEntity) and (forZone only gbo:Zone) and (hasImportance exactly 1 LevelOfImportance) and (hasPreferenceValue min 1 PreferenceValue)`

PreferenceProfile

PreferenceProfiles store Preferences, PreferenceSchedules or ActivitySchedules and can belong to either HumanActors or NonScheduledActivities.

Equivalent To:
–

SubClass Of:
`(hasActivitySchedule only ActivitySchedule) or (hasPreference only Preference) or (hasPreferenceSchedule only PreferenceSchedule)`

PreferenceSchedule

PreferenceSchedules can be used to cluster more than one ScheduledPreference of similar time together.

Equivalent To:
–

SubClass Of:
`hasPreference min 1 ScheduledPreference`

PreferenceValue

The PreferenceValue of a Preference has an assigned value and can store a specific state (of applications or appliances).

Equivalent To:
–

SubClass Of:
`(forState only Thing) and (hasValue exactly 1 Literal)`

PresencePreference

Preferences are used to model the occupancy of the smart home. If the smart home is occupied its PreferenceValue is set to 1.

Equivalent To:
–

SubClass Of:
`Preference and ((hasPreferenceValue min 1 BinaryPreferenceValue) and (isPreferenceOf min 1 PresencePreferenceSchedule))`

PresencePreferenceSchedule

This concept represents schedules that contain at least one ScheduledPreference of type PresencePreference.

Equivalent To:
`PreferenceSchedule and (hasPreference min 1 (PresencePreference and ScheduledPreference))`
SubClass Of:
`PreferenceSchedule`

R

ReadingActivity

A ReadingActivity represents any action related to reading (e.g. a book, a newspaper, . . .).

Equivalent To:
–
SubClass Of:
`PassiveActivity`

RelativeHumidityPreference

RelativeHumidityPreferences are used to define Preferences regarding the percentage of relative humidity.

Equivalent To:
–
SubClass Of:
`Preference`

S

SatisfiedHumanActor

A SatisfiedHumanActor is a HumanActor having a LevelOfSatisfaction of at least satified.

Equivalent To:
`Actor and ((hasSatisfactionLevel value Satisfied) or (hasSatisfactionLevel value VerySatisfied))`
SubClass Of:
`HumanActor`

ScheduledActivity

A ScheduledActivity is a NonScheduledActivity which is assigned to a certain time frame.

Equivalent To:
`Activity` **and** `(forTime` **min 1** `time:TemporalEntity)` **and** `(forActivity` **exactly 1** `NonScheduledActivity)`
SubClass Of:
`Activity`

ScheduledPreference

This concept represents Preferences which are scheduled for specific time frames.

Equivalent To:
`Preference` **and** `(forTime` **min 1** `time:TemporalEntity)`
SubClass Of:
`Preference`

ScheduledPreferenceProfile

A ScheduledPreferenceProfile is a profile which contains ActivitySchedules and PreferenceSchedules.

Equivalent To:
`PreferenceProfile` **and** `((hasActivitySchedule` **min 1** `ActivitySchedule)` **or** `(hasPreferenceSchedule` **min 1** `PreferenceSchedule))`
SubClass Of:
`PreferenceProfile`

SetbackTemperaturePreference

AirFlowVelocityPreferences are used to define Preferences regarding the set back temperature.

Equivalent To:
–
SubClass Of:
`TemperaturePreference`

SleepingActivity

A SleepingActivity represents any action related to sleeping.

Equivalent To:
–

SubClass Of:
`PassiveActivity`

SoundPressureLevelPreference

SoundPressureLevelPreferences are used to define Preferences regarding the level of the sound pressure.

Equivalent To:
–
SubClass Of:
`Preference`

SportActivity

A SportActivity represents any action related to doing sports.

Equivalent To:
–
SubClass Of:
`NonPassiveActivity`

StandardPreference

This concept represents Preferences which store predefined and standardized PreferenceValues.

Equivalent To:
`Preference` **`and`** `(isPreferenceOf` **`min 1`** `StandardPreferenceProfile)`
SubClass Of:
`Preference`

StandardPreferenceProfile

A StandardPreferenceProfile is a profile which only contains predefined Preferences.

Equivalent To:
–
SubClass Of:
`PreferenceProfile`

SystemActor

A SystemActor represents a system agent of the smart home system.

Equivalent To:

`Actor and (hasAge exactly 1 SystemActorAge) and (hasID exactly 1 int)`
SubClass Of:
`Actor`

SystemActorAge

A SystemActorAge represents an age of a SystemActor in hours.

Equivalent To:
`Age and (hasHours exactly 1 float)`
SubClass Of:
`Age`

T

TemperaturePreference

TemperaturePreference are used to define Preferences regarding the temperature.

Equivalent To:
–
SubClass Of:
`Preference`

U

UnsatisfiedHumanActor

A SatisfiedHumanActor is a HumanActor having a LevelOfSatisfaction of at most barely satified.

Equivalent To:
`Actor and ((hasSatisfactionLevel value BarelySatisfied) or (hasSatisfactionLevel value DisSatisfied))`
SubClass Of:
`HumanActor`

UserDefinedPreference

This concept represents Preferences which were defined by a certain Actor.

Equivalent To:
`Preference and (((isPreferenceOf min 1 HumanActorPreferenceProfile) or (isPreferenceOf min 1 (isPreferenceScheduleOf min 1 HumanActorPreferenceProfile))) or (isPreferenceOf min 1 (isPreferenceProfileOf`

`min 1 (isScheduledActivityOf min 1 (isActivityScheduleOf min 1 HumanActorPreferenceProfile)))))`

SubClass Of:

`Preference`

UserSystemActor

A UserSystemActor is a system agent which represents a HumanActor of the smart home system.

Equivalent To:

`(hasAge exactly 1 SystemActorAge) and (represents exactly 1 HumanActor)`

SubClass Of:

`SystemActor`

V

VisualComfortPreferenceSchedule

This concept is primarily used to illustrate the usage of scope-dependent schedules (i.e. schedules that group preferences serving a similar purpose together). In this case, this concept represents schedules that contain exclusively LampPreferences and LightingLevelPreferences.

Equivalent To:

`PreferenceSchedule and (hasPreference some (LampPreference or LightingLevelPreference))`

SubClass Of:

`PreferenceSchedule`

W

WakeUpActivity

A WakeUpActivity represents any action related to the process of waking up (e.g. after a nap, in the morning, . . .).

Equivalent To:

–

SubClass Of:

`PassiveActivity`

WashingmachinePreference

BlindsPreferences are used to define Preferences for washing machines.

Equivalent To:

–

SubClass Of:

`Preference`

WatchingTvActivity

A WatchingTvActivity represents any action related to watching TV.

Equivalent To:

–

SubClass Of:

`PassiveActivity`

WritingActivity

A WritingActivity represents any action related to writing (e.g. a letter, a blog entry, ...).

Equivalent To:

–

SubClass Of:

`PassiveActivity`

Y

YoungHumanActor

An AgedHumanActor is a HumanActor younger than 14.

Equivalent To:

`Actor` **`and`** `(hasAge` **`exactly 1`** `YoungHumanActorAge)` **`and`** `(hasGender` **`exactly 1`** `Gender)`

SubClass Of:

`HumanActor`

YoungHumanActorAge

An YoungHumanActorAge represents an age of a HumanActor of at most 13.

Equivalent To:

`Age` **`and`** `(hasYears` **`some`** `int[< 14])`

SubClass Of:

`HumanActorAge`

Bibliography

[1] The Protégé Ontology Editor and Knowledge Acquisition System. `http://protege.stanford.edu/`.

[2] Jans Aasman. Allegro graph: RDF triple database. Technical report, Franz Incorporated, 2006.

[3] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[4] Dave Beckett and Brian McBride. RDF/XML syntax specification (revised). *W3C recommendation*, 10, 2004.

[5] David Beckett, Tim Berners-Lee, Eric Prud'hommeaux, and Gavin Carothers. Turtle – Terse RDF Triple Language. W3C Candidate Recommendation, February 2013. `http://www.w3.org/TR/2013/CR-turtle-20130219/`.

[6] Kuderna-Iulian Benta, Amalia Hoszu, Lucia Văcariu, and Octavian Creţ. Agent based smart house platform with affective control. In *Proceedings of the 2009 Euro American Conference on Telematics and Information Systems: New Opportunities to increase Digital Citizenship*, page 18. ACM, 2009.

[7] Tim Berners-Lee and Dan Connolly. Notation3 (N3): A readable RDF syntax. *W3C Team Submission (January 2008) http://www. w3. org/TeamSubmission*, 1998.

[8] Paolo Bertoldi and Bogdan Atanasiu. Electricity consumption and efficiency trends in the enlarged European Union. *IES–JRC. European Union*, 2007.

[9] Steve Bratt. Semantic Web, and Other Technologies to Watch. W3C, 2007. `http://www.w3.org/2007/Talks/0130-sb-W3CTechSemWeb/#%2824%29`.

[10] Dan Brickley. RDF: Understanding the Striped RDF/XML Syntax, 2002. `http://www.w3.org/2001/10/stripes/`.

[11] Dan Brickley and Ramanathan V. Guha. RDF vocabulary description language 1.0: RDF schema. W3C Recommendation, February 2004. `http://www.w3.org/TR/rdf-schema/`.

[12] Jeen Broekstra, Arjohn Kampman, and Frank Van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF Schema. In *The Semantic Web—ISWC 2002*, pages 54–68. Springer, 2002.

[13] Paul Buitelaar, Philipp Cimiano, and Bernardo Magnini, editors. *Ontology Learning from Text: Methods, Evaluation and Applications*. Amsterdam: IOS Press, 2005.

[14] Paul Buitelaar, Daniel Olejnik, and Michael Sintek. A protégé plug-in for ontology extraction from text based on linguistic analysis. In *The Semantic Web: Research and Applications*, pages 31–44. Springer, 2004.

[15] Philipp Cimiano, Alexander Mädche, Steffen Staab, and Johanna Völker. Ontology learning. In *Handbook on ontologies*, pages 245–267. Springer, 2009.

[16] Oscar Corcho, Mariano Fernández-López, Asunción Gómez-Pérez, and Angel López-Cima. Building legal ontologies with METHONTOLOGY and WebODE. In *Law and the Semantic Web*, pages 142–157. Springer, 2005.

[17] Chris J. Date and Hugh Darwen. *SQL. Der Standard.: SQL/92 mit den Erweiterungen CLI und PSM.* Pearson Deutschland GmbH, 1998.

[18] Antonio De Nicola, Michele Missikoff, and Roberto Navigli. A software engineering approach to ontology building. *Information systems*, 34(2):258–275, 2009.

[19] Dejene Ejigu, Marian Scuturici, and Lionel Brunie. An ontology-based approach to context modeling and reasoning in pervasive computing. In *Pervasive Computing and Communications Workshops, 2007. PerCom Workshops' 07. Fifth Annual IEEE International Conference on*, pages 14–19. IEEE, 2007.

[20] Jérôme Euzenat and Pavel Shvaiko. *Ontology Matching*. Springer, 2007.

[21] Dieter Fensel. *Ontologies: A Silver Bullet for Knowledge Management and Electronic Commerce*. Springer, 2001.

[22] Jacques Ferber. *Multi-agent systems: an introduction to distributed artificial intelligence*, volume 1. Addison-Wesley Reading, 1999.

[23] Mariano Fernández-López, Asunción Gómez-Pérez, and Natalia Juristo. METHONTOLOGY: from ontological art towards ontological engineering. 1997.

[24] Jean Vincent Fonou-Dombeu and Magda Huisman. Semantic-Driven e-Government: Application of Uschold and King Ontology Building Methodology for Semantic Ontology Models Development. *arXiv preprint arXiv:1111.1941*, 2011.

[25] Aldo Gangemi. Ontology design patterns for semantic web content. In *The Semantic Web–ISWC 2005*, pages 262–276. Springer, 2005.

[26] Aldo Gangemi and Valentina Presutti. Ontology design patterns. In *Handbook on Ontologies*, pages 221–243. Springer, 2009.

[27] Aldo Gangemi and Valentina Presutti. Ontology Design Patterns.org(ODP). `http://www.ontologydesignpatterns.org/`, 2010. Accessed: 2014-05-27.

[28] Birte Glimm, Ian Horrocks, Boris Motik, and Giorgos Stoilos. Optimising ontology classification. In *The Semantic Web–ISWC 2010*, pages 225–240. Springer, 2010.

[29] Asuncion Gomez-Perez, Mariano Fernández-López, and Oscar Corcho. *Ontological engineering*, volume 139. Springer Heidelberg, 2004.

[30] Asunción Gómez-Pérez, Natalia Juristo, and Juan Pazos. Evaluation and assessment of the knowledge sharing technology. *Towards very large knowledge bases*, pages 289–296, 1995.

[31] Ian R. Grosse, John M. Milton-benoit, and Jack C. Wileden. Ontologies for supporting engineering analysis models. *AIE EDAM*, 19(01):1–18, 2005.

[32] OWL Working Group. OWL 2 Web Ontology Language. *W3C recommendation*, 2012.

[33] Thomas R. Gruber. Toward principles for the design of ontologies used for knowledge sharing? *International journal of human-computer studies*, 43(5):907–928, 1995.

[34] Thomas R. Gruber et al. A translation approach to portable ontology specifications. *Knowledge acquisition*, 5(2):199–220, 1993.

[35] Tao Gu, Xiao Hang Wang, Hung Keng Pung, and Da Qing Zhang. An ontology-based context model in intelligent environments. In *Proceedings of communication networks and distributed systems modeling and simulation conference*, volume 2004, pages 270–275, 2004.

[36] ASHRAE Handbook. HVAC systems and equipment. *American Society of Heating, Refrigerating, and Air Conditioning Engineers, Atlanta, GA*, 1996.

[37] Steve Harris and Andy Seaborne. SPARQL 1.1 query language. *W3C Reccomendation*, 14, 2013.

[38] Patrick Hayes and Brian McBride. RDF semantics. W3C Recommendation, February 2004. `http://www.w3.org/TR/rdf-mt/`.

[39] Jerry R. Hobbs and Feng Pan. Time ontology in OWL. *W3C working draft*, 27, 2006.

[40] Matthew Horridge and Peter F. Patel-Schneider. OWL 2 Web Ontology Language Manchester Syntax. W3C Working Group Note, December 2012. `http://www.w3.org/TR/owl2-manchester-syntax/`.

[41] Ian Horrocks. Using an expressive description logic: Fact or fiction? *KR*, 98:636–645, 1998.

[42] Ian Horrocks, Boris Motik, and Zhe Wang. The HermiT OWL Reasoner. In *Proceedings of the 1st International Workshop on OWL Reasoner Evaluation (ORE-2012), Manchester, UK*, 2012.

[43] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, Mike Dean, et al. SWRL: A semantic web rule language combining OWL and RuleML. *W3C Member submission*, 21:79, 2004.

[44] Ian Horrocks, Ulrike Sattler, and Stephan Tobies. Practical reasoning for very expressive description logics. *Logic Journal of IGPL*, 8(3):239–263, 2000.

[45] Herman J. Horst. Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2):79–115, 2005.

[46] Stephen S. Intille. Designing a home of the future. *IEEE pervasive computing*, 1(2):76–82, 2002.

[47] Ivar Jacobson, Grady Booch, James Rumbaugh, James Rumbaugh, and Grady Booch. *The unified software development process*, volume 1. Addison-Wesley Reading, 1999.

[48] Wolfgang Kastner, Mario J. Kofler, and Christian Reinisch. Using AI to realize energy efficient yet comfortable smart homes. In *Factory Communication Systems (WFCS), 2010 8th IEEE International Workshop on*, pages 169–172. IEEE, 2010.

[49] Jongwan Kim, Dejing Dou, Haishan Liu, and Donghwi Kwak. Constructing a user preference ontology for anti-spam mail systems. In *Advances in Artificial Intelligence*, pages 272–283. Springer, 2007.

[50] Nicola King. Smart Home – a Definition. `http://www.housingcare.org/downloads/kbase/2545.pdf`, 2003. Accessed: 2014-09-02.

[51] Graham Klyne, Jeremy J. Carroll, and Brian McBride. Resource description framework (RDF): Concepts and abstract syntax. *W3C recommendation*, 10, 2004.

[52] Mario J. Kofler. *An Ontology as Shared Vocabulary for Distributed Intelligence in Smart Homes*. PhD thesis, Vienna University of Technology, 2014.

[53] Mario J. Kofler and Wolfgang Kastner. A knowledge base for energy-efficient smart homes. In *Energy Conference and Exhibition (EnergyCon), 2010 IEEE International*, pages 85–90. IEEE, 2010.

[54] Mario J. Kofler, Christian Reinisch, and Wolfgang Kastner. An intelligent knowledge representation of smart home energy parameters. In *Proceedings of the World Renewable Energy Congress (WREC 2011), Linköping, Sweden*, 2011.

[55] Mario J. Kofler, Christian Reinisch, and Wolfgang Kastner. A semantic representation of energy-related information in future smart homes. *Energy and Buildings*, 47:169–179, 2012.

[56] Philippe Kruchten. *The rational unified process: an introduction*. Addison-Wesley Professional, 2004.

[57] Ora Lassila and Ralph R. Swick. Resource description framework (RDF) model and syntax specification, 1999.

[58] Jaewook Lee. Conflict resolution in multi-agent based intelligent environments. *Building and Environment*, 45(3):574 – 585, 2010.

[59] Henrik Leopold, Jan Mendling, and Artem Polyvyanyy. Generating natural language texts from business process models. In *Advanced Information Systems Engineering*, pages 64–79. Springer, 2012.

[60] Man Li, Xiao-Yong Du, and Shan Wang. Learning ontology from relational database. In *Proceedings of the International Conference on Machine Learning and Cybernetics*, volume 6, pages 3410–3415. IEEE, 2005.

[61] Kaihong Liu, William R. Hogan, and Rebecca S. Crowley. Natural language processing methods and systems for biomedical ontology learning. *Journal of biomedical informatics*, 44(1):163–179, 2011.

[62] Mariano Fernández López, Asunción Gómez-Pérez, Juan Pazos Sierra, and Alejandro Pazos Sierra. Building a Chemical Ontology Using Methontology and the Ontology Design Environment. *IEEE Intelligent Systems*, 14(1):37–46, 1999.

[63] Alexander Maedche and Steffen Staab. Ontology learning for the semantic web. *IEEE Intelligent systems*, 16(2):72–79, 2001.

[64] Deborah L McGuinness, Frank Van Harmelen, et al. OWL web ontology language overview. *W3C recommendation*, 10(2004-03):10, 2004.

[65] Boris Motik, Peter F. Patel-Schneider, and Bernardo Cuenca Grau. OWL 2 - Direct Semantics. *W3C Recommendation*, 27, 2009.

[66] Boris Motik, Peter F. Patel-Schneider, Bijan Parsia, Conrad Bock, Achille Fokoue, Peter Haase, Rinke Hoekstra, Ian Horrocks, Alan Ruttenberg, Uli Sattler, et al. OWL 2 web ontology language: Structural specification and functional-style syntax. *W3C Recommendation*, 27:17, 2009.

[67] Lam Trung Ngo and Makoto Mizukawa. RT Ontology development and human preference learning for assistive robotic service system. In *Control Automation and Systems (ICCAS), 2010 International Conference on*, pages 385–388. IEEE, 2010.

[68] Natalya Noy and Alan Rector. Defining n-ary relations on the semantic web: Use with individuals. `http://www.w3.org/TR/swbp-n-aryRelations`, 2007. Accessed: 2014-05-27.

[69] Natalya F Noy, Deborah L McGuinness, et al. Ontology development 101: A guide to creating your first ontology: Knowldege systems laboratory, stanford university. *Stanford Knowledge Systems Laboratory Technical Report KSL-01-05 and Stanford Medical Informatics Technical Report SMI-2001-0880*, 2001.

[70] Jinsoo Park, Kimoon Sung, and Sewon Moon. Developing graduation screen ontology based on the METHONTOLOGY approach. In *Networked Computing and Advanced Information Management, 2008. NCM'08. Fourth International Conference on*, volume 2, pages 375–380. IEEE, 2008.

[71] Bijan Parsia and Evren Sirin. Pellet: An OWL DL reasoner. In *Third International Semantic Web Conference-Poster*, page 18, 2004.

[72] Helena Sofia Pinto and João P. Martins. Ontologies: How can they be built? *Knowledge and Information Systems*, 6(4):441–464, 2004.

[73] Axel Polleres. SPARQL 1.1: New features and friends (OWL2, RIF). In *Web Reasoning and Rule Systems*, pages 23–26. Springer, 2010.

[74] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. W3C Recommendation, January 2008, 2008.

[75] Tirdad Rahmani, Daniel Oberle, and Marco Dahms. An adjustable transformation from OWL to Ecore. In *Proc. of MODELS*, pages 243–257, 2010.

[76] Christian Reinisch. *A Framework for Distributed Intelligence for Energy Efficient Operation of Smart Homes*. PhD thesis, Vienna University of Technology, 2012.

[77] Christian Reinisch, Mario J. Kofler, Félix Iglesias, and Wolfgang Kastner. ThinkHome: Energy Efficiency in Future Smart Homes. *EURASIP Journal on Embedded Systems*, 2011:18, 2011.

[78] Christian Reinisch, Mario J. Kofler, and Wolfgang Kastner. ThinkHome: A Smart Home as Digital Ecosystem. In *Proceedings of 4th IEEE International Conference on Digital Ecosystems and Technologies (DEST '10)*, pages 256–261, April 2010.

[79] Hajo Rijgersberg, Mark van Assem, and Jan Top. Ontology of units of measure and related concepts. *Semantic Web*, 4(1):3–13, 2013.

[80] Kurt Rohloff, Mike Dean, Ian Emmons, Dorene Ryder, and John Sumner. An evaluation of triple-store technologies for large data stores. In *On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops*, pages 1105–1114. Springer, 2007.

[81] Stefan Runde, Henrik Dibowski, Alexander Fay, and Klaus Kabitzsch. A semantic requirement ontology for the engineering of building automation systems by means of OWL. In *Emerging Technologies & Factory Automation, 2009. ETFA 2009. IEEE Conference on*, pages 1–8. IEEE, 2009.

[82] Rob Shearer, Boris Motik, and Ian Horrocks. HermiT: A Highly-Efficient OWL Reasoner. In *OWLED*, volume 432, 2008.

[83] Elena Simperl. Reusing ontologies on the Semantic Web: A feasibility study. *Data & Knowledge Engineering*, 68(10):905–925, 2009.

[84] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *Web Semantics: science, services and agents on the World Wide Web*, 5(2):51–53, 2007.

[85] Simon Steyskal. Towards an Alternative Approach for Combining Ontology Matchers. Master's thesis, Vienna University of Technology, 2013.

[86] Mari Carmen Suárez-Figueroa, Asunción Gómez-Pérez, and Boris Villazón-Terrazas. How to write and use the Ontology Requirements Specification Document. In *On the move to meaningful internet systems: OTM 2009*, pages 966–982. Springer, 2009.

[87] York Sure, Steffen Staab, and Rudi Studer. Ontology engineering methodology. In *Handbook on ontologies*, pages 135–152. Springer, 2009.

[88] Edward Thomas, Jeff Z Pan, and Yuan Ren. TrOWL: Tractable OWL 2 reasoning infrastructure. In *The Semantic Web: Research and Applications*, pages 431–435. Springer, 2010.

[89] TopQuadrant. TopBraid Composer Website. Product Website, April 2014. `http://www.topquadrant.com/tools/IDE-topbraid-composer-maestro-edition/`.

[90] Dmitry Tsarkov and Ian Horrocks. Fact++ description logic reasoner: System description. In *Automated reasoning*, pages 292–297. Springer, 2006.

[91] Ken Ukai, Yoshinobu Ando, and Makoto Mizukawa. Robot Technology Ontology Targeting Robot Technology Services in Kukanchi. *Journal of Robotics and Mechatronics*, 21(4):489, 2009.

[92] Michael Uschold and Martin King. *Towards a methodology for building ontologies*. Citeseer, 1995.

[93] Mike Uschold and Michael Gruninger. Ontologies: Principles, methods and applications. *The knowledge engineering review*, 11(02):93–136, 1996.

[94] Tam Van Nguyen, Wontaek Lim, Huy Nguyen, Deokjai Choi, and Chilwoo Lee. Context Ontology Implementation for Smart Home. *arXiv preprint arXiv:1007.1273*, 2010.

[95] Félix Iglesias Vazquez. *Smart Home Control Based on Behavioural Profiles*. PhD thesis, E384, Vienna University of Technology, 2012.

[96] Félix Iglesias Vázquez and Wolfgang Kastner. Usage profiles for sustainable buildings. In *Emerging Technologies and Factory Automation (ETFA), 2010 IEEE Conference on*, pages 1–8. IEEE, 2010.

[97] Félix Iglesias Vázquez and Wolfgang Kastner. Clustering methods for occupancy prediction in smart home control. In *Industrial Electronics (ISIE), 2011 IEEE International Symposium on*, pages 1321–1328. IEEE, 2011.

[98] Félix Iglesias Vázquez, Wolfgang Kastner, Sergio Cantos Gaceo, and Christian Reinisch. Electricity load management in smart home control. In *12th Conference of International Building Performance Simulation Association*, pages 957–964, 2011.

[99] Félix Iglesias Vázquez, Wolfgang Kastner, and Christian Reinisch. Impact of user habits in smart home control. In *Emerging Technologies & Factory Automation (ETFA), 2011 IEEE 16th Conference on*, pages 1–8. IEEE, 2011.

[100] R. Hevner von Alan, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS quarterly*, 28(1):75–105, 2004.

[101] Félix Iglesias Vázquez, Sergio Cantos Gaceo, Wolfgang Kastner, and José A. Montero Morales. Behavioral Profiles for Building Energy Performance Using eXclusive SOM. In Lazaros Iliadis and Chrisina Jayne, editors, *Engineering Applications of Neural Networks*, volume 363 of *IFIP Advances in Information and Communication Technology*, pages 31–40. Springer Berlin Heidelberg, 2011.

[102] Holger Wache, Thomas Voegele, Ubbo Visser, Heiner Stuckenschmidt, Gerhard Schuster, Holger Neumann, and Sebastian Hübner. Ontology-based integration of information-a survey of existing approaches. In *IJCAI-01 workshop: ontologies and information sharing*, volume 2001, pages 108–117. Citeseer, 2001.