MASTERARBEIT

Ontologies in Automation

ausgeführt am

Institut für Rechnergestützte Automation

Arbeitsgruppe Automatisierungssysteme

der Technischen Universität Wien

unter der Anleitung von

ao. Univ.-Prof. Dipl.-Ing. Dr. Wolfgang Kastner

durch

Rainer Müller

Hernalser Hauptstrasse 57/17

1170 Wien

Wien, January 24, 2008

Abstract

Semantic Web techniques have been proven to be an efficient method of storing information in various application areas. The intention is to enable automated processing, without the necessity of human interaction. Furthermore, semantically enriched data provides a greater flexibility due to its networked structure. In addition to this, a knowledge base (ontology) can be deployed in different ways. For example, an ontology that provides data for a web-shop, may also be integrated in industrial automation devices in order to automate communication.

The increasing tool support makes it possible to develop ontologies, even without having detailed knowledge about the underlying techniques. Data visualization tools help to keep track of the inherent complexity of Semantic Web documents.

This thesis starts with a brief overview about the Semantic Web Vision, followed by an introduction into the basic ingredients and common design issues of ontologies. Afterwards, RDF, RDFS and OWL are discussed in detail. Next, different approaches for the use of ontologies in industrial automation devices are presented. Finally, the integration of an ontology into an existing engineering tool is shown.

Kurzfassung

Semantic Web Techniken haben sich bereits in vielen Anwendungsbereichen als eine sehr effiziente Methode der Informationsspeicherung erwiesen. Das Ziel ist eine automatisierte Verarbeitung, ohne menschliches Zutun. Weiters bietet eine semantische Datenstrukturierung hohe Flexibilität auf Grund des hohen Vernetzungsgrades. Wurde eine solche *"Wissens-Datenbank"* (Ontologie) erstellt, so kann diese in unterschiedlichen Bereichen genutzt werden. Zum Beispiel kann eine Ontologie, die als Datenquelle für einen Web-Shop dient, gleichzeitig genutzt werden, um eine autonome Kommunikation zwischen Feldbusgeräten zu etablieren.

Durch die zunehmende Software-Unterstützung ist es möglich, Ontologien zu entwickeln, ohne detailiertes Wissen über die zugrundeliegenden Techniken zu haben. Daten Visualisierungsprogramme sind dabei eine nützliche und hilfreiche Methode, um den Überblick über große Datenbestände zu behalten.

Die vorliegende Diplomarbeit gibt einen Einblick in die sogenannte *Semantic Web Vision*, grundlegende Ontologie Bestandteile und allgemeine Design Richtlinen. Anschließend wird RDF, RDFS und OWL im Detail vorgestellt. Danach werden verschiedene Ansätze der Implementierung von Ontologien in Feldbusgeräten diskutiert und abschließend die Integration einer Ontologie in ein bestehendes Engineering Tool gezeigt.

Contents

List of Figures

Listings

Chapter

1 Introduction

We're drowning in information but starving for knowledge.
(John Naisbett)

In 1989, when Tim Berners-Lee invented the World Wide Web, nobody would have thought of it becoming such a powerful communication medium. Nevertheless, up to now its social contribution was limited by the user, who browses and interprets the Web. Most of its content is designed to be read by humans not by machines. This is what the Semantic Web is all about - providing structured data that can be processed by machines. For example, a hotel's website could be easily extended by annotating the hotel name, location, category or the number of rooms in a machine-processable way. By using Semantic Web languages like RDFS/OWL, common conceptualizations can be established (also referred to as *ontologies*) and information can easily be retrieved and analyzed by personal agents - even without any human interaction. Ontologies provide a common vocabulary and formal structures to organize and store knowledge. Tim Berners-Lee once stated his vision about the Semantic Web [FBL99] as follows:

I have a dream for the Web [in which computers] become capable of analyzing all the data on the Web - the content, links, and transactions between people and computers. A 'Semantic Web', which should make this possible, has yet to emerge, but when it does, the day-to-day mechanisms of trade, bureaucracy and our daily lives will be handled by machines talking to machines. The 'intelligent agents' people have touted for ages will finally materialize. (Tim Berners-Lee, 1999)

1.1 Semantic Web Vision

The word 'Semantic' is derived from the Greek word 'Semantikos' which means giving signs in a significant and symptomatic way. The idea of the 'Semantic Web' is to make the Web machine-processable by giving additional information about its content - so called Meta-data. Most of today's information is presented in a weakly structured form, e.g. text, images, audio and video. The idea is not to create a separate Web, it is rather an extension to the current one. Due to the fact that this idea remained mostly unrealized after a few years, the W3C put a lot of effort in developing and standardizing languages in order to push the development of the Semantic Web.

1.2 Ontologies

1.2.1 Definition

An ontology is a data model, which represents a set of concepts and relations within a certain domain. Further, it is possible to reason about its objects, in order to gain knowledge, which is (or is not explicitly) stored in the model.

There exist many definitions for ontologies, some of them are:

- "An ontology is an explicit specification of a conceptualization" [Gruber, 1993]
- "An ontology is a shared understanding of some domain of interest" [Uschold & Gruniger, 1996]
- "A computer model of some portion of the world" [Humns & Singh, 1997]
- "A shared and common understanding of a domain that can be communicated between people and heterogeneous and distributed systems" [Fensel, 2000]

1.2.2 Components

Most of recent ontologies share structural similarities, regardless of the used language. There are four basic components:

1.2.2.1 Instances

Instances - also referred to as Individuals - are the basic, "ground level" elements of an ontology. They can describe real objects like an electric device or a pneumatic cylinder, but also abstract terms like parameters or numbers. For example 'FB13' [1] is an instance of the class 'Fieldbus Device'.

1.2.2.2 Classes

A class or *concept* is an abstract set of objects, which may include instances or other classes. As mentioned above, 'Fieldbus Device' would be the class of the instance 'FB13'. Classes

[1] Festo Profibus Device

can also be subsets of other classes, typically the most general class is on top, usually the class 'Thing'.

A set of related classes is also called a *partition*. So, for instance, a class called 'I/O Module' may contain the partition 'Input Module' and 'Output Module'. An associated partition rule assures the correct classification of a module. If an object can only be included in one of these classes, the partition is called a *disjoint partition*. If all objects in the class 'I/O Module' are either an Input or an Output Module, which means the partition covers all objects in the super-class, the partition is also called an *exhaustive partition*.

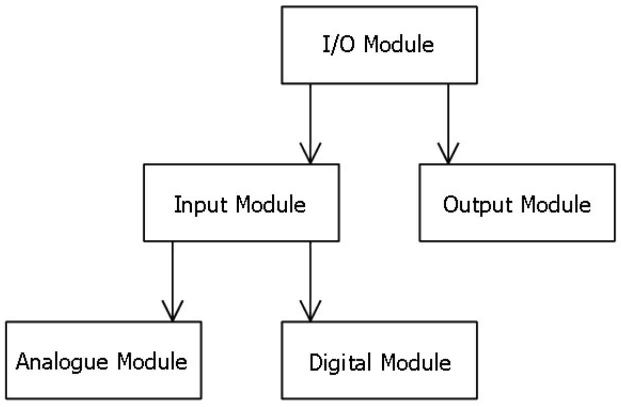


Figure 1.1: Ontology Partitions

1.2.2.3 Attributes

Ontologies usually use attributes to describe objects properties. This technique is very similar to the description of UML-classes. For example an input module can have the following attributes:

- Channels: 4

- Short Name: 4DI
- Parameters: 7

Attributes always consist of a tag and a corresponding value. The values can be of any type, even lists are possible.

1.2.2.4 Relations

Relations are used to describe interconnections between objects. In fact, a relation is an attribute and its value is another object. Regarding the input module, a possible relation might be a list of modules it can be combined with, e.g. a fieldbus module. This network-like structure is what characterizes ontologies. Simple 'is a' relationships like they are shown in Figure 1.1 are realized by using two attributes, an attribute 'is-superclass-of' and an attribute 'is-subclass-of'. Another type of relation is the 'part-of' relation, for example a valve may be part of an output module. This relation is not restricted to one parent object, which also applies to the 'is-a' relation. However, there are also other relations possible, e.g. 'connects-to', 'fits-in' and many more. As you may think, relations are highly domain-dependent.

1.2.3 Upper, Middle and Lower Ontologies

Ontology development is a very time-consuming and difficult task. That's why most ontologies cover only a certain domain of interest. Every domain-specific ontology must be based on some upper-level ontology, which contains the most general and domain-independent concepts. This includes concepts like time, space, identity, events and others. Therefore, a well-designed upper-ontology is very important, but also very difficult to develop. This complexity originates from the fact, that most problems of a top-level ontology have an abstract and highly philosophical character. Ontology development is a top-down process starting with an upper-ontology. The next step is to identify and implement some key concepts, which may be domain specific, but still are related to the upper-ontology. The last big challenge is to implement the domain-specific ontology. The major problem here is to avoid inconsistencies, which mostly come along with very detailed ontologies.

> Once the formal principles and the basic domain concepts have been assessed (a result eventually achieved in many projects) ontology engineers must face the time-consuming and expensive task of populating the ontology and making it accessible to the user of a given virtual community. [MNV02]

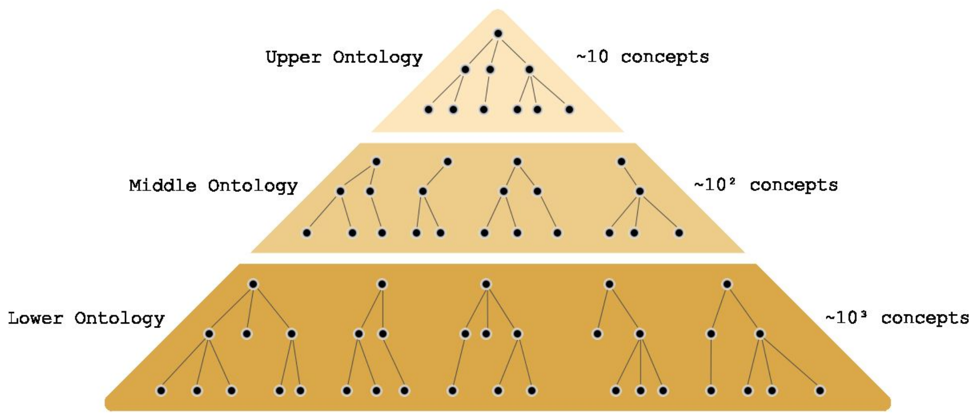


Figure 1.2: Ontology Pyramide

As Figure 1.2 shows, the number of concepts and relations increases with the level of detail. Upper-ontologies have only a few basic concepts, whereas middle-ontologies contain a few hundred and lower-ontologies thousands of concepts and relations. An example of Ontology-Layering is given in Figure 1.3.

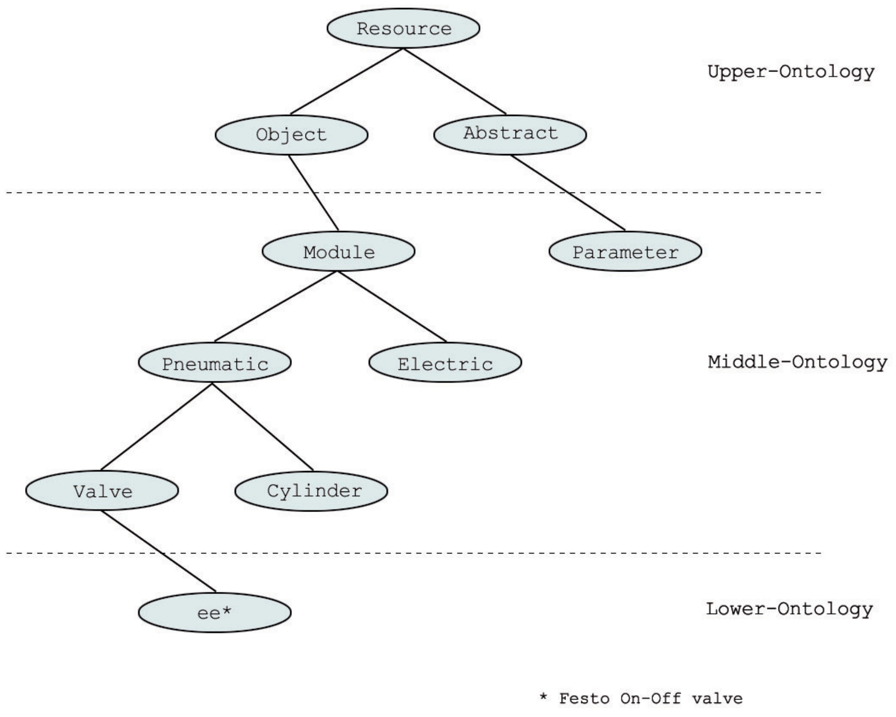


Figure 1.3: Ontology Layers

1.2.3.1 Upper Ontology (Top-level)

An upper-ontology defines very general, universal classes and properties, which are used in all domains. The most popular example in this category is the 'Cyc' Upper Ontology[2]. 'Cyc' is a commercial project, but there is also a small part available for free called 'OpenCyc'.

> Upper-level ontologies capture mostly concepts that are basic for human understanding of the world. They are 'grounded' in (supported by, wired to) the common sense that makes it hard to formalize a strict definition for them. They represent the so called prototypical knowledge. [KSD01]

[2]http://www.cyc.com/

1.2.3.2 Middle Ontology

Middle-ontologies are basically upper-ontologies, but they additionally implement concepts for a specific domain. It is quite difficult to draw a line between middle- and upper-ontologies, this depends on what the designer considers as domain independent and domain specific information.

1.2.3.3 Lower Ontology

Lower-ontologies are intended to represent specific domains, with their own classes and properties, e.g. an ontology describing field devices and modules. Developing lower ontologies is very time-consuming. The major challenge is to populate the ontology without emerging inconsistencies. Usually this layer is the biggest part of an ontology, since this is the place where the actual data is stored. Once the ontology is operational, this layer is usually subject to a lot of growth. Inference engines (software that is capable of acquiring information, which is not explicitly stated) may also store new gathered data here.

1.2.4 Ontology Specification

Ontology specification is quite a difficult process, therefore a good guideline is very important. The 'Ontology Requirements Specification Document' [3] published at Wikipedia, suggests the following approach:[4]

[3] http://ontoworld.org/wiki/Ontology_Requirements_Specification_Document

[4] This list is not intended to be complete but represents a good basis for a more detailed set of requirements.

1.2.4.1 Goal, Domain and Scope

First the field of interest should be carefully analyzed, maybe there is already an existing ontology which can be used. Afterwards a list of the intended goal, domain and scope has to be established. An example of these assertions useful for the automation domain is shown below:

- The ontology describes modules, which are used in industrial automation.
- Its aim is to distribute knowledge about their internal structure and interoperability.
- A configuration tool with a built-in reasoner (inference engine) retrieves information about these modules.

1.2.4.2 Design Guidelines

Number of Concepts and Granularity
This estimation deals with the number of concepts and the aspired accuracy of the intended ontology. In order to avoid ending up with far too many concepts after implementation, this is a necessary and useful part of the requirements specification. Furthermore, engineers might get a clearer view of what has to be done.

Naming Convention
It is also very useful to define a certain naming convention, a set of common rules for naming concepts and relations. For instance beginning all concepts with upper case, whereas relations with lower case letters, e.g. 'Valve' and 'switchOn' is a good choice. Usually the

names must not include spaces, for long names *camelBack*-notation[5] should be used. Chapter 6 of [NM01] proposes a good base for naming rules. Differing naming conventions may cause problems when merging ontologies (e.g. referring to the same concept with different names or different concepts with same name). This problem can be solved, by implementing a mapping table which translates between common naming conventions. Nevertheless, it is better to avoid different ontologies in advance.

Guidelines

There are several guidelines for ontology development, but some of them are very specific (like [Con07] - a guide for biological knowledge bases). Another more general and very interesting guide can be found at [NM01]. This guide uses Protégé[6] to illustrate the ontology design process, supported by several examples.

Knowledge Sources

- domain experts - A person with special knowledge in a particular area.
- (reusable) ontologies
- dictionaries - A list of words, including additional information and meaning of each word.
- thesauri - A list of synonyms, sometimes also including related words and antonyms.
- other sources (databases, index lists, web pages, organization charts, ...)

[5]http://en.csharp-online.net/Identifiers
[6]http://protege.stanford.edu/

1.2.4.3 Users and Scenarios

In order to successfully design an ontology, it is important to know who will be using the ontology and what is its intention. Therefore, a list of possible usage scenarios has to be created, describing single situations where an ontology may be useful or not. This analysis intends to clarify what potential users expect and what they need in order to make their work easier. In software engineering, use cases are a common method to describe usage scenarios[7].

1.2.4.4 Applications

Before implementing an ontology, it is necessary to decide which language should be used. Nowadays, there are a many applications which make ontology development a lot easier (e.g. Protégé), therefore application support may be a quite important decision guidance.

1.2.5 Ontology Implementations

Whenever possible, re-using an ontology is highly recommended. In fact re-use and sharing is a primary reason for introducing ontologies. The decision which ontology to choose depends on several factors. First the ontology has to be present in a well-known format, e.g. RDF/OWL. Additionally it should be freely available or at least affordable. Finally, it has to comply with the stated requirements or be easily adjustable.

[7]http://www.uml.org/

1.2.5.1 (Open)Cyc

The Cyc project was started in 1984. In 1994 a spin-off company called Cycorp Inc. was founded. The name is derived from the word 'encyclopedia' and is a registered trademark of Cycorp. It is the world's largest knowledge base of everyday common sense knowledge. Cycorp's website describes Cyc as following:

> The Cyc Knowledge Server is a very large, multi-contextual knowledge base and inference engine developed by Cycorp. Cycorp's goal is to break the "software brittleness bottleneck" once and for all by constructing a foundation of basic "common sense" knowledge - a semantic substratum of terms, rules, and relations - that will enable a variety of knowledge-intensive products and services.[8]

The Cyc knowledge base is proprietary, but a free excerpt of it is available called 'OpenCyc'. It contains over 1M assertions, 100K atomic concepts and 10K predicates, whereas the free version contains 60K assertions, 6k concepts and is considered as upper ontology. Concept names are called constants in Cyc, they start with "*#$*" and are case-sensitive, e.g. *#$ConveyorBelt*. The most used predicates are *#$isa* (is-instance-of) and *#$genls* (is-a-subcollection). Statements in Cyc are written using the CycL (Cyc Language). Predicates are written first followed by their attributes:

"A conveyor belt belongs to the collection of externally powered devices":

```
(#$isa #$ConveyorBelt #$ExternallyPoweredDevice)
```

"All externally powered devices are powered devices":

```
(#$genls #$ExternallyPoweredDevice #$PoweredDevice)
```

[8] http://www.cyc.com/cyc/technology/whatiscyc

1.2.5.2 (Euro)WordNet

The development of WordNet started in 1985 by the cognitive science laboratory at the Princeton University. The intention was to create a thesaurus, a semantic lexicon for the English language. English words are organized into sets of synonyms, so called *'synsets'*. These sets are linked via several relations. WordNet covers nouns, verbs, adjectives and adverbs. It is implemented in RDF (an open standard) unlike Cyc, which uses its own language. WordNet is free available and can be used in commercial or research projects.

> The initial idea was to provide an aid to use in searching dictionaries conceptually, rather than merely alphabetically. [...] WordNet can be said to be a dictionary based on psycholinguistic principles.[MBF+05]

For the time being, WordNet containes approximately 155K words, merged into 117K synsets and resulting in about 207k word-sense pairs[9]. There is also a multilingual version of WordNet called EuroWordNet, which supports different European languages. These languages are interconnected via the WordNet Interlingual Index (ILI). The project was finished in 1999. EuroWordNet is not freely available, in fact it is very expensive.

From an ontological point of view WordNet is not suitable, because its intention was to represent links in natural language. This includes problems like the mix-up of concepts and individuals, e.g. *"Mozart"* and *"songwriter"* at the same level. Nevertheless, there are already projects with the goal to turn WordNet into a correct ontology.

1.2.5.3 SUMO

SUMO - short for *Suggested Upper Merged Ontology* - is intended as "a starter document" by the IEEE. Originally created at Teknowledge Corporation, this ontology experienced

[9]http://wordnet.princeton.edu/man/wnstats.7WN

extensive input from the Standard Upper Ontology (SUO) mailing list. Developing upper ontologies is difficult and sometimes results in endless philosophical debates, e.g. should time be regarded as 4th dimension (4D) or considered separately (3D and time). These fundamental decisions can easily lead to incompatibilities between upper ontologies. SUMO is a conjunction of several existing upper ontologies, therefore it has been divided into 11 sections with carefully documented interdependencies. For example the first section deals with structural issues, containing a relation framework in order to make proper ontology development possible. Figure 1.4 shows an overview of these topical sections.

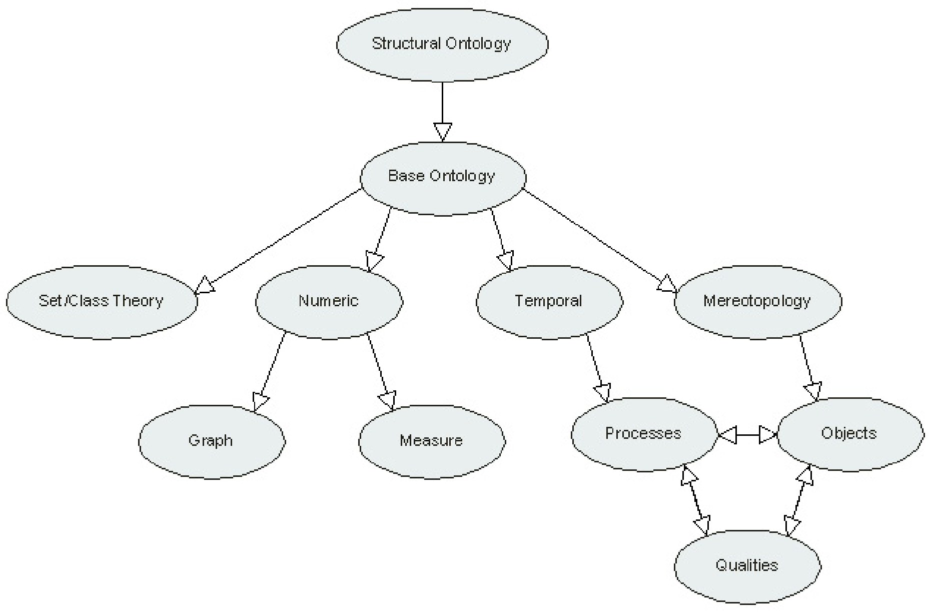


Figure 1.4: SUMO sections

The language used by SUMO is a version of the Knowledge Interchange Format (KIF) called SUO-KIF[10]. SUMO is case-sensitive: classes and individuals start with upper-case letters, whereas relations use only lower-case letters. Also multiple inheritance is allowed (concepts may have several parents).

[10]http://suo.ieee.org/SUO/KIF/index.html

SUMO contains about 1000 well-defined and documented concepts, 4000 axioms and about 800 rules. Combined with the Middle-Level Ontology (MILO) and some Lower-Ontologies, SUMO represents the largest free, formal ontology available [Cor07]. There are also open source tools for browsing and reasoning, furthermore SUMO has been mapped to WordNet 1.6 and ported to version 2.0 later on.

The SUMO was created by merging publicly available ontological content into a single, comprehensive and cohesive structure.[NP]

Chapter

2 Ontology Languages

Language is the dress of thought.
(Samual Johnson)

2.1 History

In the 1980s, the first knowledge representation projects like KL-ONE [1] and CLASSIC were started, but they had a significant drawback. All these systems used their own ontology language. Later, in the 1990s, Ontolingua[2] was developed by the Knowledge System Labs (KSL). Ontolingua is based on the Knowledge Interchange Format (KIF) and is able to translate from and to *Description Logic*[3]-based languages. At this time KIF became a standard for ontology modeling. In the late 1990s, ontologies became an interesting topic for Web applications. The first languages for semantic annotation of websites arose, e.g. SHOE [oCSUoM]. SHOE is a small extension to HTML which allows authors of websites

[1]http://en.wikipedia.org/wiki/KL-ONE
[2]http://www.ksl.stanford.edu/software/ontolingua/
[3]http://en.wikipedia.org/wiki/Description_logic

to enrich their pages with machine-processable information. There are many more projects and languages which are not mentioned here, but all of them made their contribution to recent ontology languages, like RDF.

2.2 Basic Ingredients

Many ontologies use proprietary languages, but since *re-use* is one fundamental idea of ontologies, efforts have been made to establish standards. Designing an ontology is a compromise between the expressiveness needed to properly represent human concepts, and minimal complexity needed to maintain computability. The most basic ingredients are a list of concepts and relations between them. For example, in industrial automation, *pneumatic valves*, *sensors* or *controller devices* are concepts, but even more abstract concepts are possible like *parameters*, *units* or *numbers*. Many ontologies use a hierarchical structure, a relation which states a class C is a subclass of another class C', e.g. the class of *valve terminals* is subsumed by *pneumatic devices*. Other relations implemented by most ontologies are:

- Properties (e.g. A *contains* B)
- Restrictions (e.g. only *valve terminals* can contain *valves*)
- Disjunction (e.g. *pneumatic valves* and *controller devices* are disjoint)
- Other logical relationships (e.g. a *valve terminal* must consist of at least 2 *valves*)

2.3 Design Issues

In order to be able to describe a specific domain of interest by using ontologies, the following requirements have to be kept in mind [GvH04].

1. well-defined syntax
2. well-defined semantics
3. efficient reasoning support
4. sufficient expressive power
5. convenience of expression

In order to make automatic processing possible, a *well-defined syntax* is very important. Despite of the fact that the syntax is not very user-friendly, XML has turned out to be a suitable language for the semantic web. Furthermore, development has become much easier by authoring tools and other ontology applications.

Ontology languages have to be precise and use *well-defined semantics*. This means no multiple interpretations are allowed, like it is known in mathematical logic.

One major goal of the semantic web is to allow reasoning about the stored knowledge. This includes acquiring knowledge about:

- **Class Membership**: This represents a transitive relation, which means if x is an instance of y and y an instance of z, we can derive that x must also be an instance of z.

$$\forall x, y, z \in X;\ xRy \ \wedge \ yRz \ \Rightarrow \ xRz$$

This relation can also be used to detect class equivalences. If x is equivalent to y and y to z, x is also equivalent to z.

- **Consistency**: The class membership relation allows us to check for inconsistency, e.g. if x is an instance of C and D, while C and D are disjoint classes. This indicates a possible error in the ontology:

$$x \in C, D \ \wedge \ C \cap D = \emptyset \ \Rightarrow \ Error$$

- **Classification**: Some ontologies are intended for automatic semantic annotation, like the KIM (Knowledge and Information Management) Platform. This means, by declaring certain class properties as sufficient condition for class membership, it is possible to classify unknown instances. For example if a class C has the properties c_1 and c_2, an individual x can be regarded as an instance of this class, if it implements these properties.

Automatic reasoning is very valuable, especially in large ontologies. It allows to check many interdependencies in a very short time.

Developing ontologies is always a compromise between *sufficient expressive power* and *efficient reasoning support*. From now on, we refer to the level of detail as the 'expressiveness' of an ontology.

> Generally speaking, the richer the language is, the more inefficient the reasoning support becomes, often crossing the border of non-computability. Thus we need a compromise, a language that can be supported by reasonably efficient reasoners, while being sufficiently expressive to express large classes of ontologies and knowledge. [RFS07]

2.4 Expressiveness of Ontologies

There are different ways to categorize ontologies. Besides their scope, mentioned in Section 1.2.3, we can also use their expressiveness and their internal structure respectively, to classify them [Fen07].

- **Thesaurus**: A thesaurus constitutes interconnections between related terms, the most popular example is WordNet.
- **Informal Taxonomy**: Taxonomies use hierarchical structures, which makes generalization and specialization possible, but informal taxonomies do not implement inheritance. Therefore, an instance of a class is not necessarily an instance of the superclass.
- **Formal Taxonomy**: A formal taxonomy strictly sticks to inheritance. Each instance of a class is an instance of the superclass. A prominent example is UNSPSC. The *United Nations Standard Products and Services Code* (UNSPSC) is a coding system to categorize products and services used in global e-Commerce applications.
- **Classes**: Classes also referred to as frames are characterized by a number of properties, which are inherited to subclasses and instances. Examples are ontologies implemented in RDFS.
- **Value Restrictions**: Property values are subject to restrictions, e.g. ontologies in OWL Lite.
- **General Logic Constraints**: In addition to value restriction, properties are also restricted by other logical or mathematical constraints, e.g. a *valve terminal* must consist of at least 2 *valves*. OWL DL is a typical member of this category.
- **Expressive Logic Constraints**: Ontology languages like CycL use first-order logic

constraints and other relationships, like disjoint or inverse classes.

Due to the fact, that relations like the disjointness of classes are also supported by OWL DL and even OWL Lite, classification of ontologies are often very difficult. Categorizing ontologies helps to get an overview about possible solutions, but in the end a well-done requirements analysis is the most important part of the design process.

CHAPTER

3 RDF

The goal is to transform data into information, and information into insight.
(Carly Fiorina)

The *Resource Description Framework* (RDF) is a data model that uses Uniform Resource Identifiers (URI) to represent nodes and edges. Even if it is often referred to as 'language' in literature, it is better to talk about it as data model - a model for storing information in graphs. There are several different syntax implementations for RDF, most people think of RDF as an XML language. In fact, this syntax is clearly geared towards automatic processing. There are other implementations that can be read much easier by humans. The RDF model is based on the idea, of making statements about resources in the form subject-predicate-object. Every subject is connected to an object by a predicate. These expressions are also called triples in RDF terminology.

3.1 History

In 1995, Ramanathan V. Guha[1] - an Indian computer scientist - created the Meta Content Framework (MCF). MCF is the closest ancestor to RDF. The first application using MCF was 'Hotsauce', a browser plug-in developed by Apple in 1996. It allows the user to navigate in a 3D environment through a website that includes MCF metadata. Later, in 1997, Ramanathan continued his work at Netscape. Together with Tim Bray[2], he worked on an XML-based version of MCF. Netscape used this technology in its 'smart browsing' feature (*What's related* search). By adding namespace support RDF got its current form. In 1999, the W3C published the first specification of RDF's data model and XML-based syntax. Five years later a new version was published. In addition to the XML syntax, there are also more readable notations like N3 (Notation 3), but this will be discussed later in this chapter.

3.2 RDF Basics

Machine-processable languages need a strong syntax, like XML. Even if XML cannot be read that easily by humans, its clear structure and form is predestined for the use in RDF and semantic web applications. Furthermore, XML provides good tool support and a big community. However, XML does not tell us anything about the meaning of data. For example, the tag ordering needs to be interpreted by an application. This circumstance is illustrated by the following example:

a valve terminal is a *pneumatic module*.

[1]http://en.wikipedia.org/wiki/Ramanathan_V._Guha
[2]http://en.wikipedia.org/wiki/Tim_Bray

This statement can be represented in different ways:

Listing 3.1: XML Example 1

```
<ModuleType="PneutmaticModule">
            <Module>ValveTerminal</Module>
</ModuleType>
```

Listing 3.2: XML Example 2

```
<Module="ValveTerminal">
            <Type>PneumaticModule</Type>
</Module>
```

Both versions state that valve terminals are pneumatic modules, although they use a different nesting. This shows that there is no unique way of assigning meaning to data. This is where RDF comes into play.

3.2.1 Resources

The most basic elements in RDF are resources. Things that we make statements about, are called *resources*. Every resource can be identified by an URI, a Universal Resource Identifier. URIs can be URLs (Unified Resource Locators) or other unique identifiers, e.g. ISBN. It is important to know, that these identifiers do not necessarily represent a way to access resources. In most cases identifiers point to locations, where descriptions of objects are stored. Strictly speaking, RDF uses an URI reference to identify a single resource, also referred to as *URIref*. URIrefs are URIs with an optional fragment identifier. For example an URIref *http://mydomain.at/festo.rdf#fb13*, can be split in the URI *http://mydomain.at/festo.rdf* and the fragment identifier *fb13*. Fragment identifiers are always separated via a '#' symbol.

In RDF, predicates are referenced by URIs, too. This method enables global (re-)use and reduces definition of equivalent terms. Predicates are used to describe relations between resources, e.g. *isPartOf*, *hasParameters*. Predicates are also called properties.

3.2.2 Statements

The basic building block of RDF is called *statement*, which consists of the subject-predicate-object triple. Subjects are resources, whereas objects can either be resources or literals. Literals are simple datatypes, like strings or integers.

3.2.3 Literals

In order to know how to interpret data, we need additional information. This information can be used to identify values, e.g. as numbers or dates. It is not strictly necessary to use literals, because this information could also be given by an URI, but the use of literals is regarded to be more convenient and intuitive. As already mentioned, objects can be resources or literals, whereas subjects and predicates are always resources. Suppose we read the information:

(http://www.domain.at/#FB13, http://www.domain.at/#hasParameters, "18")

We do not know if "18" is of string or integer type, nor if it is hexadecimal or decimal. A program reading this data, can only know how to interpret it, if the information is explicitly given. RDF solves this problem, by adding datatype URIs to literals. Literals can be *plain* or *typed*.

Plain Literals

If there is no URI asserted, the literal is called *plain*. A string datatype is used with an optional tag indicating the language.

Typed Literals

Typed literals are also strings, but they are always combined with datatype URIs. A datatype URI gives further information about the literal's type. It is not strictly necessary to use datatypes with literals. In fact, its intention is, to make it easier for vocabulary users to understand a particular RDF document. This means there is no additional semantic information attached, i.e. automated processes are not obliged to check the validity of a literal's type. The following examples illustrate the use of typed literals with XML-based syntax and N3:

Listing 3.3: Typed Literals

```
<?xml version="1.0"?>
<rdf:RDF
      xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
      xmlns:md="http://www.domain.at/">

   <rdf:Description rdf:about="#FB13">
      <md:hasParameters rdf:datatype="http://www.w3.org/2001/XMLSchema#int">
                  18
            </md:hasParameters>
   </rdf:Description>
```

<#FB13> <md:#hasParameters> <"18"^^http://www.w3.org/2001/XMLSchema#int>.

RDF provides no mechanism for defining new datatypes. XML Schema Datatypes [W3Ci] provides an extensibility framework suitable for defining new datatypes for use in RDF. [W3Cg]

3.3 RDF Views

There are different ways to formalize RDF statements. Suppose we want to state:

A valve is a part of a pneumatic module.

N3

Even if RDF/XML is the most important method to represent RDF data, there are also other techniques to serialize RDF, e.g. N3 or Notation3. N3 breaks an RDF graph into separate triples, each triple contains a subject, a predicate and an object, which are separated by spaces. Additional information outside the brackets makes statements more readable, but is ignored in automated processing. An example is given below:

<#Valve> is <#Part> of <#PneumaticModule>.

It is also possible to use full URIs or namespace-qualified XML names (QNames[3]) in N3. Further information about this serialization technique can be found at [W3Cc] published by Tim Berners-Lee.

RDF Graph

Another more visual approach is to use RDF graphs. This representation method uses directed labeled graphs, which are also called *nodes and arcs diagrams*. As already mentioned, in RDF only binary predicates are allowed. Therefore, each predicate connects two nodes, starting from a resource and pointing to another resource or literal. Nodes can be

[3]http://www.w3.org/2001/tag/doc/qnameids.html

URIrefs, blank nodes or literals. Blank nodes, also called *bnodes*, are nodes without an URI. However, in order to be able to reference the same resource within a document, blank node identifiers are used. These local identifiers are placeholders for values, which are not present at the moment. Blank nodes will be discussed in detail later on.

Graphs are a good method to keep track of growing RDF documents. URIrefs, literals, bnodes and predicates are the only components used. Therefore, graphs are easy to read for humans, but they are not suitable for automated processing. Figure 3.1 shows an excerpt of an RDF graph generated by the W3C RDF Validator [W3Ce]:

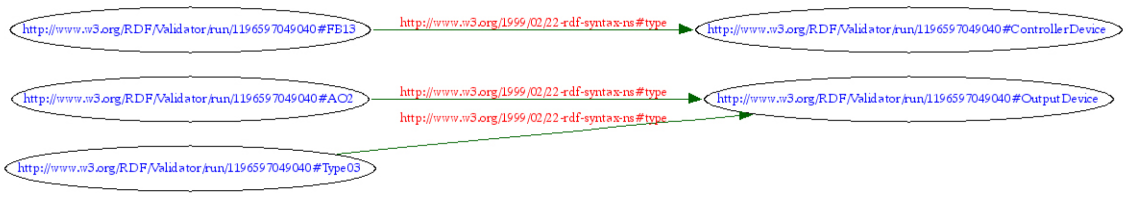

http://www.w3.org/RDF/Validator/run/1196597049040#FB13
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://www.w3.org/RDF/Validator/run/1196597049040#ControllerDevice
http://www.w3.org/RDF/Validator/run/1196597049040#AO2
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://www.w3.org/RDF/Validator/run/1196597049040#OutputDevice
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://www.w3.org/RDF/Validator/run/1196597049040#Type03

Figure 3.1: RDF Graph

RDF/XML

The third and most important method uses XML to serialize RDF data. Because of its strong structural syntax, XML is a good base for automated processing. In RDF documents, the root node is `rdf:RDF`, that surrounds the data. A big advantage of the RDF/XML syntax are namespace declarations, which are usually declared at the beginning of a document. A very simple RDF/XML file is shown in Listing 3.4.

Listing 3.4: A simple RDF document

```xml
<?xml version="1.0"?>
<rdf:RDF
          xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
          xmlns:p1="http://www.mydomain.at/">
</rdf:RDF>
```

In this example two namespaces are declared, *rdf* and a namespace called *p1*. This technique makes ontology sharing very easy, because we only need to know the specific namespace in order to (re-)use elements of an existing ontology. This enables the development of large, distributed knowledge collections.

Data is stored in `rdf:Description` tags, which are used to make statements about resources. For existing resources, we use the *about* attribute, whereas the `rdf:ID` attribute is used to create new resources.

Listing 3.5: The *about* Attribute

```xml
<rdf:Description rdf:about="p1:FB13">
          <rdf:type rdf:resource="p1:FieldbusDevice"/>
</rdf:Description>
```

The content of `rdf:Description` tags is also called *property elements*. In our example we also use the `rdf:type` element, this allows us to introduce some structure to RDF documents. Section 3.4 will discuss RDF tags in detail.

3.4 RDF/XML Syntax

As mentioned before, there are several different methods to represent RDF data. Many of them only exist in order to make RDF documents more readable for humans. Nevertheless, the most popular and important is the XML-based syntax.

Every RDF document contains an `rdf:RDF` tag. This tag encloses all other tags. A single data record starts with `rdf:Description` and contains a variable number of other tags. Listing 3.6 shows a simple RDF document.

Listing 3.6: RDF Document

```xml
<?xml version="1.0"?>
<rdf:RDF
      xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
      xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">

   <rdf:Description rdf:about="#Device">
      <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
   </rdf:Description>

   <rdf:Description rdf:about="#OutputDevice">
      <rdfs:subClassOf rdf:resource="#Device"/>
      <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
   </rdf:Description>

   <rdf:Description rdf:about="#ControllerDevice">
      <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
      <rdfs:subClassOf rdf:resource="#Device"/>
   </rdf:Description>

   <rdf:Description rdf:about="#FB13">
      <rdf:type rdf:resource="#ControllerDevice"/>
   </rdf:Description>

   <rdf:Description rdf:about="#AO2">
      <rdf:type rdf:resource="#OutputDevice"/>
   </rdf:Description>

```

```
  <rdf:Description rdf:about="#Type03">
    <rdf:type rdf:resource="#OutputDevice"/>
  </rdf:Description>
</rdf:RDF>
<!-- Created with Protege (with OWL Plugin 3.3.1, Build 430) -->
<!-- http://protege.stanford.edu -->
```

This file was automatically created by Protégé, but slightly modified afterwards in order to make it more readable. As mentioned in Section 1.2.4, it is not allowed to use whitespaces in names, instead we use *Camelback*-notation for long names. Furthermore, names must start with a character. The RDF data structure of Listing 3.6 is represented in Figure 3.2.

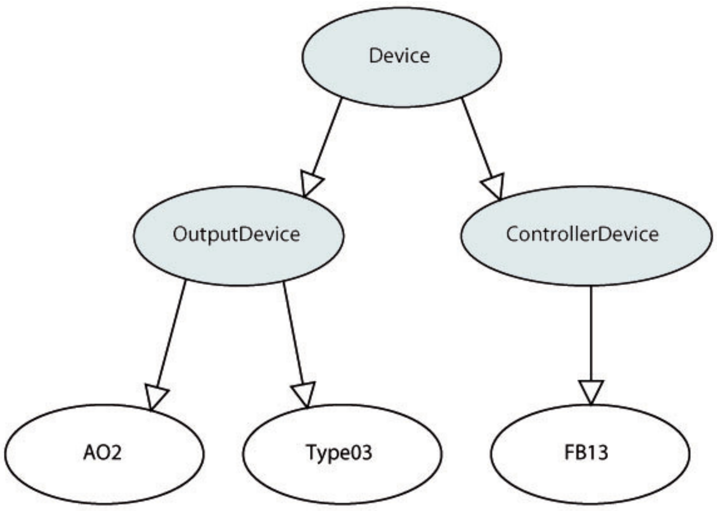


Figure 3.2: RDF Example Structure

The first line of Listing 3.6 shows the used XML version, followed by the `rdf:RDF` tag. Namespaces are an important part of RDF, usually they are declared as attributes within the `rdf:RDF` tag, but it is also possible to declare them elsewhere. This technique allows us, to reuse and extend existing ontologies, by linking to external resources. The next tag is the `rdf:Description`, which encloses one or more subject-predicate-object triples. It is also possible to use a subject in several statements within an `rdf:Description` tag. This is illustrated in Figure 3.3 [W3Ce].

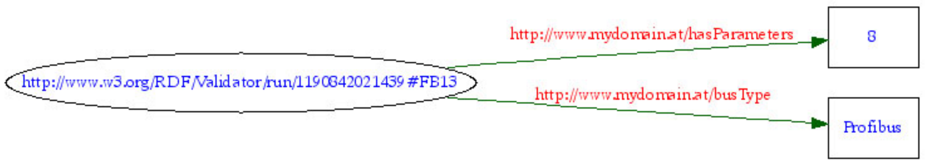

http://www.w3.org/RDF/Validator/run/1190842021439#FB13
http://www.mydomain.at/hasParameters
8
http://www.mydomain.at/busType
Profibus

Figure 3.3: Multiple Subject Usage

3.4.1 `rdf:ID` and `rdf:about`

In order to refer to a data record, we can use the `rdf:about` or `rdf:ID` tag, as seen in Listing 3.6. The meaning is quite similar, but `rdf:ID` provides an additional check, since this tag is allowed only once for each data record. Especially when creating a set of distinct, but related terms this may be a useful feature. Usually `rdf:ID` is used to create a new resource, whereas `rdf:about` can be used to reference and enrich an already existing one. In contrast to `rdf:ID`, we have to add the *'#'* symbol when using `rdf:about`. Some people use these tags to distinguish between technically accessible objects, e.g. a Web page and those that are abstract or real-world objects. More on `rdf:about` and `rdf:ID` can be found at [Ogb].

3.4.2 `rdf:resource`

Objects can be literals, but also resources. In order to indicate the usage of a resource, we need the `rdf:resource` tag. Suppose we want to state that *FB13* is member of the device family *CPX*, we can handle this as follows:

Listing 3.7: Resource As Object - XML

```xml
<?xml version="1.0"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:md="http://www.domain.at/">

  <rdf:Description rdf:about="#FB13">
    <md:hasParameters>8</md:hasParameters>
    <md:family rdf:resource="#CPX" />
  </rdf:Description>

  <rdf:Description rdf:about="#CPX">
    <md:fullName>Compact Performance Extension</md:fullName>
  </rdf:Description>
</rdf:RDF>
```

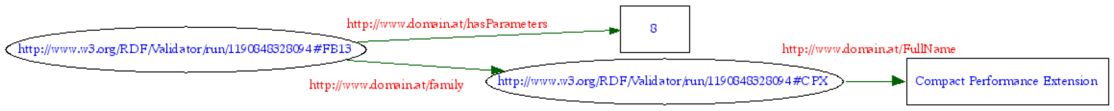

http://www.domain.at/hasParameters
8
http://www.w3.org/RDF/Validator/run/1190848328094#FB13
http://www.domain.at/FullName
http://www.domain.at/family
http://www.w3.org/RDF/Validator/run/1190848328094#CPX
Compact Performance Extension

Figure 3.4: Resource As Object - Graph

As shown in Figure 3.4, resource nodes are represented by ellipses, whereas for RDF literals rectangles are used. It is easy to see, that the *family*-relation ends in another resource. In Listing 3.7, this resource could also have been nested directly into the *FB13* description, but in order to keep things clear, it is better to leave them separated. Otherwise the code becomes unreadable, even though automated processing is not affected.

3.4.3 `rdf:parseType`

We have already mentioned RDF literals and their characteristics. Normally, RDF parsers do not need explicit information about the given object type. Nevertheless, it is possible to add an optional **`rdf:parseType`** tag, in order to tell parsers about the used type. This allows us for example, to use additional XML tags within an RDF document. A parser will interpret this as single string, which can be further processed by another application afterwards.

Listing 3.8: rdf:parseType

```
<rdf:Description rdf:about="#FB13">
        <md:hasParameters rdf:parseType="Literal">
                <h1>8</h1>
        </md:hasParameters>
    <md:family rdf:resource="#CPX" />
</rdf:Description>
```

The object can also be identified as resource, by simply adding **`rdf:parseType="Resource"`**. Using this tag tells the parser explicitly to create an anonymous object node. Furthermore, this makes the **`rdf:Description`** tag dispensable. However, the use of this technique is very controversial, because it is no complete substitution for the Description tag. There are still conditions were the Description tag must be used.

3.4.4 QNames and Namespace Support

As already mentioned earlier, namespaces are a very important part of RDF. Especially, when merging data models, namespace support is needed in order to avoid element collision. Namespaces are usually defined in the following way:

```
xmlns:name="Reference URI"
```

This is where QNames come into play. A QName consist of a namespace prefix, a colon (:) and an XML local name, e.g. `dc:creator`. Similar to the #define directive in the programming language C, a parser will replace the used QNames with the full namespace URIs. Usually the prefix *rdf* is used to reference the RDF Syntax Schema, whereas *rdfs* is used for RDF Schema and *dc* for Dublin Core schema.

3.4.5 Blank Nodes

Blank Nodes - also called *bnodes* - represent resources, that do not have a value at the moment. These bnodes can also denote, that the use of a resource URI is not advisable. For instance, when using URIs to identify humans, *bnodes* may be used.

Most of the tools assign internal identifiers to blank nodes. However, it is also possible to assign an identifier yourself, by using the `rdf:nodeID` tag. This tag has to be unique within a document, but when merging RDF data, the tools could change the identifiers used. Therefore, the `rdf:nodeID` tag, is not the right way to provide a global identifier. In this case, it is better to use an URI. It is also important to know, that `rdf:nodeID` is RDF/XML specific and does not exist in the RDF abstract model. Its intension is to support people working with RDF/XML.

3.4.6 URI abbreviation

It is also possible to abbreviate resource URIs. The simplest way is to use `rdf:about` with an absolute URI as already seen in previous examples, but in order to shorten this procedure, we can also use `xml:base`. This XML attribute allows to define a base, used when resolving URIs. This affects all RDF attributes like `rdf:about`, `rdf:ID`, `rdf:resource` and `rdf:datatype`. Without `xml:base`, RDF uses the document's URI as its base. Listing 3.9 shows an example using `xml:base` to abbreviate URIs:

Listing 3.9: URI abbreviation

```xml
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
                 xmlns:festo="http://mydomain.at/festo#"
                 xml:base="http://mydomain.at/festo.rdf">
  <rdf:Description rdf:ID="Device">
    <festo:vendor> Festo </festo:vendor>
  </rdf:Description>
</rdf:RDF>
```

The `xml:base` is usually added to the namespace list. Nevertheless, it can also be placed somewhere else. Without the `xml:base` tag, the documents URI is used. The corresponding RDF graphs are shown in Figure 3.5 and Figure 3.6.

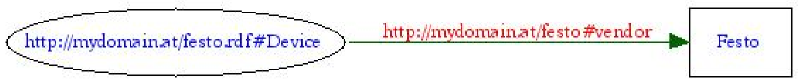

http://mydomain.at/festo.rdf#Device
http://mydomain.at/festo#vendor
Festo

Figure 3.5: URI abbreviation using xml:base [W3Ce]

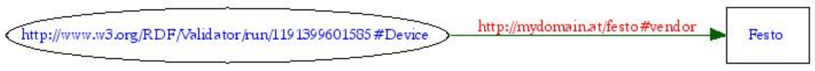

http://www.w3.org/RDF/Validator/run/1191399601585#Device
http://mydomain.at/festo#vendor
Festo

Figure 3.6: URI abbreviation without xml:base [W3Ce]

3.4.7 `rdf:type`

The `rdf:type` tag is used to emphasize that a resource is a member of a specific group. For example, Listing 3.6 states, that 'FB13' is of type *ControllerDevice*, a resource may also belong to more than one single group. In detail, the value of the `rdf:type` property identifies an `rdfs:Class`, but this will be discussed later on.

3.5 RDF/XML Abbreviation Techniques

There are several ways to shorten RDF/XML syntax. This is also what makes understanding RDF quite challenging at the beginning, because hardly anybody uses pure RDF. However, it is very important to get used to the basic concepts of the RDF abbreviation. Some shortcuts are intuitive and easy to understand, others are a bit more sophisticated.

Merging multiple predicates

This shortcut is very common and simple. We have already mentioned the `rdf:about` attribute and its usage to enrich already existing resources. When serializing an RDF graph, we would write a separate `rdf:Description` block for each subject-predicate-object triple. Therefore, subjects with multiple predicate relations, would also pop up in multiple `rdf:about` tags.

Listing 3.10: Multiple rdf:Description Tags

```
<rdf:Description rdf:about="#FB13">
    <festo:vendor> Festo </festo:vendor>
</rdf:Description>

<rdf:Description rdf:about="#FB13">
    <festo:bus> Fieldbus </festo:bus>
```

```
</rdf:Description>
```

In order to avoid this, we can merge separate predicates within one `rdf:Description` block. This technique is shown in Listing 3.11.

Listing 3.11: Merging rdf:Description Tags

```
<rdf:Description rdf:about="#FB13">
    <festo:vendor> Festo </festo:vendor>
    <festo:bus> Fieldbus </festo:bus>
</rdf:Description>
```

Childless Predicate Tags

Another abbreviation method is to place childless predicates into the description tag, but the involved objects must be literal values in order to do this. The corresponding listing is shown below.

Listing 3.12: Childless Predicates

```
  <rdf:Description rdf:about="#FB13"
                   festo:vendor="Festo"
                   festo:bus="Fieldbus"/>
```

This example represents the same data as Listing 3.11, but uses a significantly shorter syntax.

Omitting `rdf:Description` Blocks

We already mentioned the `rdf:type` property before. This tag can also be used for abbreviation purposes. Instead of enclosing the data by `rdf:Description` blocks, we put the type property directly into the XML structure. This technique will be used later on in RDF

Schema, since RDF itself does not provide any mechanism to define application specific classes or groups. However, a prior example showing this abbreviation is listed below.

Listing 3.13: Omitting rdf:Description

```
<festo:ControllerDevice rdf:about="#FB13"
                        festo:vendor="Festo"
                        festo:bus="Fieldbus"/>
```

3.6 RDF concepts

There are a few concepts that need special attention. First, because they are very controversial, but also because of their inherent complexity. These concepts are containers, collections and reification.

3.6.1 Containers

Sometimes it is useful to describe groups of related things, e.g. a specific product category, like Festo's CPX series. Containers have been intensely discussed by the RDF Working Group, because the information they provide, can also be achieved by the `rdf:type` property. However, RDF containers are still included in the RDF/XML specification, perhaps because their usage is more convenient than `rdf:type`. Container items are called *members*, which may be resources or literals. There are three predefined container types, which will now be discussed in detail.

`rdf:Bag`

A *Bag* contains resources or literals without a significant ordering, which may also include duplicates. An example for the `rdf:Bag` structure could be the set of Festo's CPX modules. There is no need to have a certain internal order within the CPX series, so `rdf:Bag` would be the right choice. Listing 3.14 shows the corresponding syntax. Similar to the HTML list item tag <LI>, RDF container elements have a prefixed `rdf:li` tag.

Listing 3.14: The rdf:Bag container

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
                    xmlns:festo="http://mydomain.at/festo#">

  <rdf:Description rdf:about="http://mydomain.at/festo_devices.htm">
    <festo:modules>
          <rdf:Bag>
            <rdf:li rdf:resource="http://mydomain.at/modules#FB13"/>
            <rdf:li rdf:resource="http://mydomain.at/modules#FB06"/>
            <rdf:li rdf:resource="http://mydomain.at/modules#FB14"/>
          </rdf:Bag>
        </festo:modules>
  </rdf:Description>
</rdf:RDF>
```

As shown in Figure 3.7, `rdf:Bag` elements are identified by automatically generated numbers. Furthermore, we can see the internal `rdf:Bag` representation using the `rdf:type` property.

`rdf:Seq`

The RDF *Sequence* contains a group of resources or literals, maybe including duplicates, where the ordering is of special interest. An example could be an alphabetically sorted list of module parameters. The graphical structure of `rdf:Seq` is similar to those of `rdf:Bag` containers. The only difference is the `rdf:type` pointing to RDF Sequence instead.

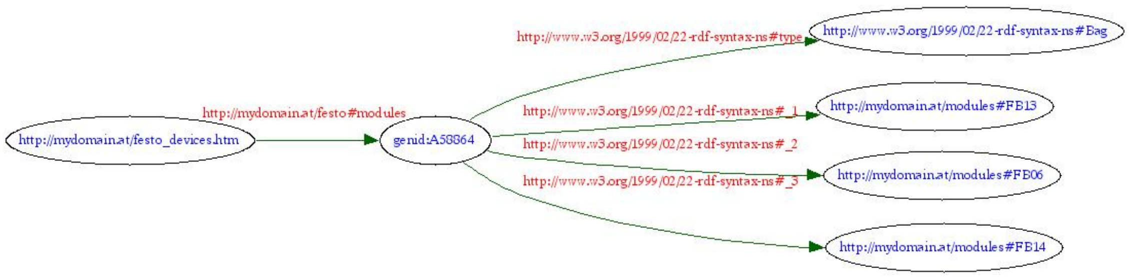

http://mydomain.at/festo_devices.htm
http://mydomain.at/festo#modules
genid:A58864
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://www.w3.org/1999/02/22-rdf-syntax-ns#Bag
http://www.w3.org/1999/02/22-rdf-syntax-ns#_1
http://mydomain.at/modules#FB13
http://www.w3.org/1999/02/22-rdf-syntax-ns#_2
http://mydomain.at/modules#FB06
http://www.w3.org/1999/02/22-rdf-syntax-ns#_3
http://mydomain.at/modules#FB14

Figure 3.7: The rdf:Bag graph [W3Ce]

`rdf:Alt`

An RDF *Alternative* contains a group of resources or literals, that represent alternative elements. For example, a set of different pneumatic modules, which are identical in construction, but built by different vendors. The first item listed in `rdf:Alt` is normally used as default item, as long as there is no other criteria for selection. RDF Alternative containers are frequently used for translation purposes, e.g. a pneumatic module's name property pointing to an `rdf:Alt` container holding all different name representations.

3.6.2 Collections

In contrast to *Containers*, a *Collection* is a finite list of objects. RDF already provides the needed vocabulary to implement this list structure. These predefined predicates are `rdf:first`, `rdf:rest` and `rdf:nil`. In order to state that we are describing a collection, the tag `parseType="Collection"` has to be used. The corresponding RDF/XML syntax is quite simple, as shown in Listing 3.15.

Listing 3.15: RDF Collections

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
                    xmlns:festo="http://mydomain.at/festo#">
```

```

  <rdf:Description rdf:about="http://mydomain.at/ControllerDevice">
    <festo:modules rdf:parseType="Collection">
            <rdf:Description rdf:about="http://mydomain.at/modules#FB13"/>
            <rdf:Description rdf:about="http://mydomain.at/modules#FB06"/>
            <rdf:Description rdf:about="http://mydomain.at/modules#FB14"/>
          </festo:modules>
  </rdf:Description>
</rdf:RDF>
```

To illustrate the use of RDF *Collections*, Figure 3.8 is shown below. As this example shows, it is not necessary to use the former mentioned predefined predicates within the code. The parser will automatically generate these relations. This graph has been created with RDF Gravity[4] - a very comfortable RDF visualization tool, which allows the user to rearrange nodes in order to make the graph more readable.

As the graph demonstrates, each element of the RDF collection is represented by an object whose corresponding subject is a blank node. The linking predicate is `rdf:first`. These RDF triples are connected via `rdf:rest` predicates. The end of the collection is indicated by the object `rdf:nil`. This example shows that even very simple RDF/XML constructs may result in a rather complex RDF graph.

3.6.3 Reification

In RDF it is possible to make statements about statements. This feature is called *Reification*, but since this mechanism is quite negligible in industrial automation, this section will give only a brief overview about this technique.

An example for a reified statement is shown below:

[4]http://semweb.salzburgresearch.at/apps/rdf-gravity/index.html

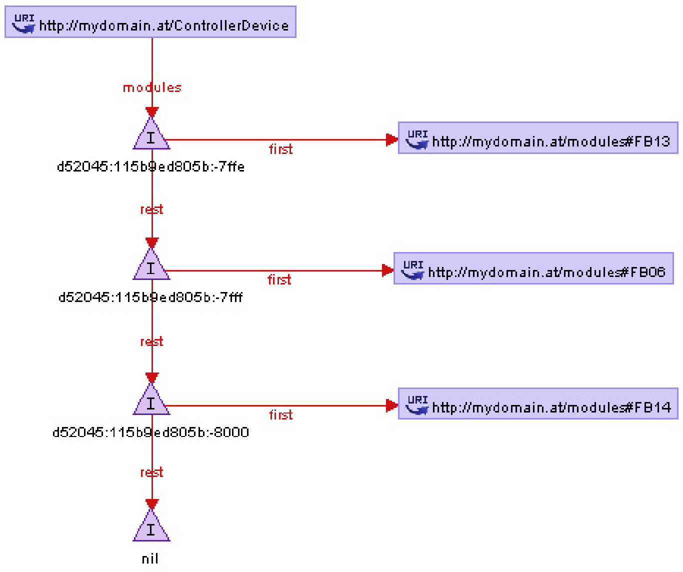

URI http://mydomain.at/ControllerDevice
modules
d52045:115b9ed805b:-7ffe
first
URI http://mydomain.at/modules#FB13
rest
d52045:115b9ed805b:-7fff
first
URI http://mydomain.at/modules#FB06
rest
d52045:115b9ed805b:-8000
first
URI http://mydomain.at/modules#FB14
rest
nil

Figure 3.8: RDF Collections

Tim states that Festo is the vendor of FB13.

As this statement implies, *Reification* introduces a certain level of trust. More on *'Reification implying trust'* can be found at [Pow03].

RDF already provides built-in vocabulary for reification in RDF/XML, including the type `rdf:Statement` and the properties `rdf:subject`, `rdf:predicate` and `rdf:object`. It is important to understand that the original statement cannot be omitted, since Reification only represents a model of it. The example mentioned above is represented by Listing 3.16.

Listing 3.16: Reification Example

```xml
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
                 xmlns:festo="http://mydomain.at/festo#"
                 xml:base="http://mydomain.at/festo.rdf">

  <rdf:Description rdf:ID="FB13">
    <festo:vendor> Festo </festo:vendor>
  </rdf:Description>

  <rdf:Statement rdf:about="#TimFB13">
        <rdf:subject rdf:resource="#FB13" />
        <rdf:predicate rdf:resource="http://mydomain.at/festo#vendor" />
        <rdf:object>Festo </rdf:object>

        <festo:statedBy rdf:resource="Tim" />
  </rdf:Statement>
</rdf:RDF>
```

This way of reifying a statement is very cumbersome, especially when we want to use reification multiple times. Nevertheless, this example provides some basic understanding on how reification works. Additionally, RDF provides a shorter and more comfortable syntax to create reified statements.

Instead of specifying subject, predicate, object and type, reification can also be achieved by using the `rdf:ID` property. This technique is shown in Listing 3.17. The corresponding RDF graph is represented by Figure 3.9.

Listing 3.17: Reification Shortcut

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
                  xmlns:festo="http://mydomain.at/festo#">

  <rdf:Description rdf:about="#FB13">
    <festo:vendor rdf:ID="TimFB13"> Festo </festo:vendor>
  </rdf:Description>

  <rdf:Description rdf:about="#TimFB13">
      <festo:statedBy rdf:resource="Tim" />
  </rdf:Description>
</rdf:RDF>
```

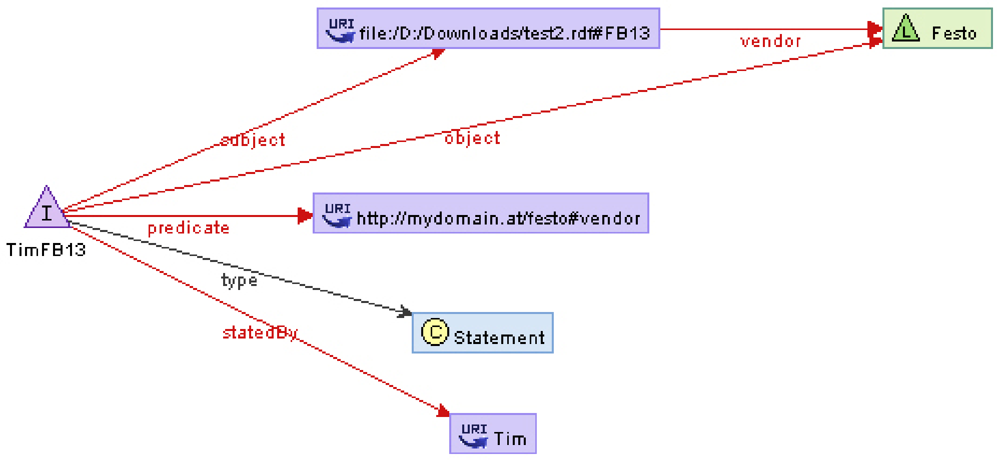

URI file:/D:/Downloads/test2.rdf#FB13
vendor
Festo
subject
object
TimFB13
predicate
URI http://mydomain.at/festo#vendor
type
statedBy
Statement
URI Tim

Figure 3.9: RDF Reification

Chapter

4 RDF Schema

Conference attendees want to know what's stable enough that they can seriously put investments toward, ... XML clearly is ready...and now RDF is a solid spec.
(Eric Miller)

As already mentioned earlier, RDF provides no mechanisms to define application-specific vocabulary. Instead, we can describe classes and properties using the RDF description language RDF Schema, also referred to as RDFS.

RDFS itself does not provide any predefined classes or properties. In fact, it offers the possibility to describe these data structures and their internal relations. For example, it allows to organize classes in a hierarchical way by using subclass relations, e.g. the class of pneumatic cylinders is a subclass of pneumatic devices.

RDFS documents represent valid RDF graphs, but the additional meaning requires software, that is capable to process this information. Therefore, RDF software must include both, the `rdf:` and the `rdfs:` vocabulary. Like RDF, RDFS uses its own namespace *http://www.w3.org/2000/01/rdf-schema#*, usually abbreviated with the prefix `rdfs:`.

4.1 Classes in RDFS

In RDF Schema, a class represents the concept of a category or a type. They are defined using the RDF vocabulary `rdfs:Class`, `rdfs:Resource` and the properties `rdf:type` and `rdfs:subClassOf`. Class members are also called *instances*, similar to object-oriented programming languages. Resources may also be instances of more than a single class, e.g. a member of the class PneumaticDevice may also be a member of the class MechanicDevice.

In RDF/XML a class is defined using the following syntax:

Listing 4.1: Describing Classes

```xml
<?xml version="1.0"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xml:base="http://stud3.tuwien.ac.at/~e0125551/RDF/schema.rdfs">

  <rdf:Description rdf:ID="PneumaticDevice">
    <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
  </rdf:Description>
</rdf:RDF>
```

Generally, classes are written with an initial uppercase letter, whereas properties usually start with a lowercase letter. This convention is not strictly necessary, but makes reading RDF documents a lot easier.

4.1.1 `rdfs:subClassOf`

In order to describe how classes are related to each other, we can use the predefined property `rdfs:subClassOf`. Suppose, we want to create a class PneumaticCylinder, which is a subclass of PneumaticDevice. We can extend Listing 4.1 by adding the following few lines:

Listing 4.2: Subclass Relation

```
<rdf:Description rdf:ID="PneumaticCylinder">
  <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
  <rdfs:subClassOf rdf:resource="#PneumaticDevice"/>
</rdf:Description>
```

This means, that any member of the class PneumaticCylinder is also a member of the class PneumaticDevice. As already stated before, RDF software that is not capable of reading RDF Schema data, will not understand this additional meaning. Instead, it will interpret this terms as simple statements with predicate `rdfs:subClassOf`, but it will not be able to understand its special meaning.

Another important characteristic of RDFS is transitivity. As already mentioned in Section 2.3, class membership represents a transitive relation, i.e. if class x is a subclass of class y, and class y is subclass of class z, this implies that class x is also subclass of class z.

$$xRy \;\wedge\; yRz \;\Rightarrow\; xRz \qquad R...subClassOf$$

Therefore, instances of class x are also instances of class y and z. All classes in RDFS are defined as subclass of `rdf:Resource`, since all instances must be resources.

It is also possible to omit the `rdf:type` definitions when using the `rdfs:subClassOf` relation, because this information can be inferred by the application. So, Listing 4.3 validates successfully in [W3Ce] and results in the graph shown in Figure 4.1. Nevertheless, it makes reading RDFS documents easier, if this additional information is explicitly given.

Listing 4.3: Omitting `rdf:type`

```
<rdf:Description rdf:ID="PneumaticCylinder">
  <rdfs:subClassOf rdf:resource="#PneumaticDevice"/>
</rdf:Description>
```

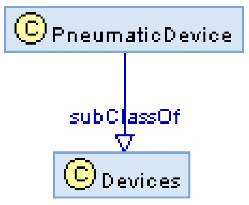


Figure 4.1: Omitting `rdf:type`

Additionally, RDF provides an abbreviation technique for description tags containing the `rdf:type` property. So, RDF classes can easily be described the way shown in Listing 4.4.

Listing 4.4: RDFS Class Abbreviation

```
<rdfs:Class rdf:ID="PneumaticCylinder"/>
```

When describing classes, it is good practice to use an explicit xml:base declaration. This guarantees that a class's URIref remains unchanged, even after relocating or copying the document.

4.2 Properties in RDFS

In order to characterize classes, we can also define specific properties. Therefore, RDF provides the class `rdf:Property` and the RDFS properties `rdfs:domain`, `rdfs:range` and `rdfs:subPropertyOf`.

Similar to class descriptions, properties are defined by the `rdf:Property` tag. In addition, RDFS provides vocabulary to define how these properties should be used within RDF data.

4.2.1 `rdfs:range`

With `rdfs:range`, it is possible to restrict the values of properties to a specific class. Suppose, we want to define a property *hasParameter*, whose value must be of class *Parameter*. This can be achieved the following way:

Listing 4.5: `rdfs:range` Property

```
<rdfs:Class rdf:ID="Parameter"/>

<rdf:Property rdf:ID="hasParameter">
  <rdfs:range rdf:resource="#Parameter"/>
</rdf:Property>
```

Properties can also have multiple `rdfs:range` attributes, but it is important to know, that a given value must be an instance of *all* specified classes.

4.2.2 `rdfs:domain`

The `rdfs:domain` attribute indicates, which classes the property can be used with. For example, it may be useful to restrict the property *hasParameter* to the class *Device* only, since there are no other classes that have parameters. Like `rdfs:range`, properties may also implement more than a single `rdfs:domain` attribute. Therefore, a class that uses this property must be an instance of *all* specified classes.

Listing 4.6 and 4.7 represent two files, which implement a small example to illustrate these RDFS techniques.

Listing 4.6: devices.rdf

```xml
<?xml version="1.0"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:ext="http://stud3.tuwien.ac.at/~e0125551/RDF/schema.rdfs#"
  xml:base="http://stud3.tuwien.ac.at/~e0125551/RDF/devices.rdf">

  <ext:Parameter rdf:ID="Channels"/>
  <ext:PneumaticDevice rdf:ID="MPA1G">
    <ext:hasParameter rdf:resource="#Channels"/>
  </ext:PneumaticDevice>
</rdf:RDF>
```

Listing 4.7: schema.rdfs

```xml
<?xml version="1.0"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xml:base="http://stud3.tuwien.ac.at/~e0125551/RDF/schema.rdfs">

  <rdfs:Class rdf:ID="Device"/>
  <rdfs:Class rdf:ID="Parameter"/>
    <rdfs:Class rdf:ID="PneumaticDevice">
    <rdfs:subClassOf rdf:resource="#Device"/>
  </rdfs:Class>
  <rdf:Property rdf:ID="hasParameter">
    <rdfs:range rdf:resource="#Parameter"/>
    <rdfs:domain rdf:resource="#Device"/>
  </rdf:Property>
</rdf:RDF>
```

4.2.3 `rdfs:subPropertyOf`

Similar to `rdfs:subClassOf`, RDF Schema provides also a way to define subproperties by using the predefined `rdfs:subPropertyOf` attribute. For example, a property *hasParameter* may have a subproperty called *hasSystemParameter*. A *Device* having this property, is by definition also a *Device*, that has a parameter.

$$xRy \;\Rightarrow\; xSy \qquad R...hasSystemParameter \quad S...hasParameter$$

A single subproperty may have one or multiple parent-properties. RDFS attributes like `rdfs:domain` and `rdfs:range` are inherited. Therefore, the value of *hasSystemParameter* must be of class *Parameter* and the property can only be used within the domain of *Device*.

4.2.4 Other Properties

In addition to the properties mentioned before, RDFS also provides predefined vocabulary that makes RDF documents more readable and easier to understand. These properties are `rdfs:Label`, `rdfs:comment`, `rdfs:seeAlso` and `rdfs:isDefinedBy`.

The `rdfs:Label` property is used to provide a more readable resource name. The `rdf:range` of this property is the class `rdfs:Literal`. As stated before, resource's names have to meet certain conditions, like the absence of whitespaces, hence it may be useful to add a more expressive name by using `rdfs:Label`.

Additionally, `rdfs:comment` can be used to provide more detailed descriptions of resources.

The property `rdfs:seeAlso` refers to additional information about the subject resource. Domain and range are both of class `rdfs:Resource`. Furthermore, RDFS provides a subproperty of `rdfs:seeAlso` called `rdfs:isDefinedBy`, which may be used to point to RDF data describing a resource.

The Sections B.1, B.2 and B.3 give detailed information about all predefined RDF classes and properties, including their domain and range.

With RDFS we can describe basic RDF vocabulary. Nevertheless, sometimes it may be useful to have a schema language with more capabilities, e.g. cardinality, transitivity. These and other additional features, are provided by the *Web Ontology Language*, discussed in the following chapter.

Chapter

5 OWL

OWL and RDF are much of the same thing, but OWL is a stronger language with greater machine interpretability than RDF. (W3 Schools)

In the previous chapter RDFS introduced a way to define domain-specific vocabulary, but as already mentioned, there is still demand for more semantic capabilities. For example, it is not possible to restrict a property's cardinality or to state that two classes are disjoint. With RDFS, class relationships can be defined, but in order to add additional information, an ontology language like the Web Ontology Language (OWL) is needed.

In fact, this is also a question of how precise a vocabulary should be, because OWL provides a lot of techniques to add constraints to data. These constraints may not be essential, but the more precise the ontology is, the more information can be inferred later on.

Some people may claim that the correct abbreviation should be WOL, but since OWL is easier to pronounce and suggests wisdom, OWL has been selected as acronym for the Web

Ontology Language.

5.1 History

In February 2004, OWL has been endorsed by the W3C as *Recommendation* in order to draw attention to the OWL specification. OWL originates from the DAML (DARPA Agent Markup Language) project, strictly speaking its language DAML+OIL (Ontology Inference Layer). The OIL development started in 1997 in Amsterdam and is based on SHOE (Simple HTML Ontology Extensions). In March 2000 DAML 0.5 has been released, followed by the 1.0 specification of DAML+OIL in 2001.

> The reference description document characterizes DAML+OIL as "a semantic markup language for Web resources." It builds on earlier W3C standards such as RDF and RDF Schema, and extends these languages with richer modeling primitives. [...] The language has a clean and well defined semantics. [Cov]

In November 2001 the W3C Web Ontology (WebOnt) Working Group has been founded. Their aim was to develop a broadly accepted ontology language - the Web Ontology Language, OWL.

The first OWL document released, was *OWL Use Cases and Requirements*, first time published in 2002 and updated in February 2004. This document [W3Cb] identifies three concepts that characterize an ontology language:

- Classes (general things) in the many domains of interest
- The relationships that can exist among things
- The properties (or attributes) those things may have

This may be quite confusing, because there are already concepts in RDF/RDFS that fulfill these requirements, but this will be discussed later on. However, the succeeding document released by the WebOnt Working Group, contained test cases to check whether the specified requirements are met.

The W3C Recommendation of OWL consists of six separate documents listed below.

- **OWL Web Ontology Language Use Cases and Requirements [W3Cb]**
  contains a set of use cases and requirements for an ontology language.
- **OWL Web Ontology Language Overview**
  provides a simple introduction to OWL, including a very brief feature list.
- **OWL Web Ontology Language Guide [W3Ca]**
  similar to the *RDF Primer* [W3Cd], this document gives a very good and complete overview with several examples.
- **OWL Web Ontology Language Reference**
  represents a systematic and compact document containing information about all modeling primitives of OWL.
- **OWL Web Ontology Language Semantics and Abstract Syntax**
  represents the formal definition of OWL.
- **OWL Web Ontology Language Test Cases**
  contains several test cases to validate the language.

All of these documents have been reviewed several times and were finally published in February 2004. As already mentioned, the OWL Language Guide provides a very good way to get used to the Web Ontology Language and its characteristics. Therefore, for those who are unfamiliar with OWL, this document would be the right choice to start with.

5.2 OWL Types

There are three different types of OWL, which differ in their complexity and expressiveness. Ontology development is a very difficult process, users have to decide which of the three types suits their needs best. Basically, the used language should possess as much expressiveness as needed and as few complexity as possible.

OWL Lite

The intention of this type is to provide mechanisms for developing classification hierarchies and enrich them with simple constraints. OWL Lite already supports cardinality constraints, but permits only values of 0 or 1. As the name indicates, OWL Lite is the 'poorest' member of the three OWL types.

OWL DL

In contrast to OWL Lite this type also supports more complex ontologies, without losing the computational completeness, i.e. processing finishes in finite time. OWL DL includes all language constructs, except type separation. This means a class can not be an individual or property and a property can not be an individual or class. The name OWL DL originates from its relation to *Description Logic*, a decidable subset of first order logic (FOL).

OWL Full

OWL Full provides maximum expressiveness, with no computational guarantees and a possibly infinite processing time. For example, it is feasible to enrich pre-defined RDF/OWL vocabulary by adding new information. Therefore, it is quite unlikely, that every inference software will support OWL Full.

OWL Lite is a subset of OWL DL, whereas OWL DL is again a subset of OWL Full. The Language Guide uses the following set of relations to illustrate this:

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

OWL also supports distributed ontologies, which is what the semantic web is about. Therefore, ontologies can gather additional information by importing data from other ontologies.

Reasoning in OWL is based on the *open world* assumption. This has various meanings. First, a reasoner will not assume something as being wrong, as long as this is not explicitly stated. Furthermore, resource descriptions may be distributed, e.g. a resource R is defined in ontology O_1 and extended in O_2. Information can only be added, it is not possible to retract previously made statements. Nevertheless, these statements may also be opposing.

5.3 OWL Documents

In Chapter 3, the advantage of using namespaces has been mentioned. This RDF block remains unchanged within OWL documents. Additionally, it is possible to provide entity references by using *Document Type Definitions* (DTD). Instead of writing the full URIs in the namespace declaration, we can use an abbreviated syntax as shown in Listing 5.1. DTD is part of the XML specification and usually used to define the structure of documents.

Listing 5.1: Namespaces

```
<!DOCTYPE owl [
    <!ENTITY xsd  "http://www.w3.org/2001/XMLSchema#" >
    <!ENTITY dev  "http://www.mydomain.at/devices#" >
    <!ENTITY par "http://www.mydomain.at/parameters#" > ]>
<rdf:RDF
    xmlns      ="&dev;"
    xmlns:dev  ="&dev;"
    xmlns:base="&dev;"
    xmlns:par="&par;"
    xmlns:owl  ="http://www.w3.org/2002/07/owl#"
    xmlns:rdf  ="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">

  <!-- OWL block -->
</rdf:RDF>
```

After the namespaces have been declared, the OWL ontology block starts, this block is also called the *ontology header*. Delimited by the tags `owl:Ontology`, this block typically contains information about the ontology itself, the recent version (`owl:versionInfo`), comments (`rdfs:comment`) and URIs of imported ontologies (`owl:imports`). A simple header is shown in Listing 5.2.

Listing 5.2: Ontology Header

```
<owl:Ontology rdf:about="">
  <owl:versionInfo> v 1.0 2007/11/15 19:10:30 mrainer </owl:versionInfo>
  <rdfs:comment>A simple header</rdfs:comment>
  <owl:imports rdf:resource="http://www.mydomain.at/devices"/>
</owl:Ontology>
```

As we can see, the `rdf:about` attribute is empty, which identifies the given base URI as an instance of owl:ontology. This is the standard case, as the document containing the header normally also contains the ontology itself.

Similar to software, ontologies may get modified over time. Therefore, we need a possibility to give information about the actual ontology version. This is usually done by the `owl:versionInfo` tag. Additionally, OWL provides `owl:priorVersion` in order to make statements about former versions.

The tag `rdfs:comment` has already been mentioned in Chapter 4 and needs no further explanation.

Like the preprocessor directive *'#include'* - known from various programming languages - OWL provides a tag called `owl:imports`, which can be used to integrate external data in a document. `owl:imports` takes a single argument, which is identified by `rdf:resources`. All assertions of the imported ontology are accessible in the document afterwards. Usually the imported ontology is also included in the namespace definitions, in order to make the use of external properties and classes easier. The difference between namespaces and `owl:imports` is, namespaces are used for disambiguation, whereas imported ontologies provide assertions that can be used. Furthermore, `owl:imports` is transitive, which means if an ontology O_3 is imported by O_2 and O_1 imports O_2 than O_3 is also imported by O_1.

5.4 OWL Basics

Similar to RDF, the most basic elements of OWL are classes, properties and instances. The used built-in vocabulary is located at *http://www.w3.org/2002/07/owl#*, usually associated with the namespace *owl*. The following section will discuss this vocabulary in detail, but will primarily focus on OWL Lite and OWL DL.

5.4.1 Classes

Classes are used to establish groups with similar characteristics. These classes are usually populated with instances afterwards. In OWL Lite and OWL DL, instances can not act as classes at the same time, whereas OWL Full allows such constructs. The class members are also called *extension* of the class in OWL terminology.

Furthermore, OWL uses two predefined classes `owl:Thing` and `owl:Nothing`. Thus all individuals of an ontology are part of the `owl:Thing` extension, whereas the extension of `owl:Nothing` is represented by the empty set. This implies, that every class is a subclass of `owl:Thing` and a superclass of `owl:Nothing`.

Classes are simply defined using `owl:Class`. Hierarchical structures are still established with rdfs:subClass, as already mentioned in the previous chapter. A simple class *Output-Device* (a subclass of the class *Device*) can be defined as shown in Listing 5.3.

Listing 5.3: OWL Class

```
<owl:Class rdf:ID="OutputDevice">
  <rdfs:subClassOf rdf:resource="#Device" />
</owl:Class>
```

`owl:disjointWith` Element

It is also possible to state whether two classes are disjoint by using the predefined element `owl:disjointWith`. For example, we can state that the class *OutputDevice* is disjoint with *InputDevice*. Of course, the ontology designer has to prove if this distinction is useful, because there might be devices that provide both input- and output-characteristics. The disjoint-statements can be included in the class definition, but they can also be added afterwards by using `rdf:about`.

Listing 5.4: owl:disjointWith

```
<owl:Class rdf:about="#OutputDevice">
  <owl:disjointWith rdf:resource="#InputDevice"/>
</owl:Class>
```

`owl:equivalentClass` Element

In order to point out the equivalence of classes, the attribute `owl:equivalentClass` can be used. However, this does not imply class equality, since equivalent classes only represent the same concept. Real class equality can be expressed by using `owl:sameAs`, but this requires to treat classes as individuals and can only be used within OWL Full.

Nevertheless, owl:equivalentClass is a very important feature, because it can be used to tie classes in different ontologies together.

Listing 5.5: owl:equivalentClass

```
<owl:Class rdf:about="#Device">
  <owl:equivalentClass rdf:resource="#Module"/>
</owl:Class>
```

5.4.2 Individuals

Members of classes are called individuals. In Chapter 3, the property `rdf:type` has been introduced. This property can be used to state, which class an individual belongs to. There is also an abbreviation technique to omit the `rdf:type`, as we have already seen in the last chapter. The following two examples are identical in their meaning.

Listing 5.6: Defining Individuals - abbreviated

```
<OutputDevice rdf:ID="AO2" />
```

Listing 5.7: Defining Individuals

```
<owl:Thing rdf:ID="AO2">
  <rdf:type rdf:resource="#OutputDevice">
</owl:Thing>
```

As these examples illustrate, designing ontologies can be quite challenging. Especially drawing the line between classes and instances may be difficult. In relation to this, the OWL Language Guide [W3Ca] states:

> Thus classes should correspond to naturally occurring sets of things in a domain of discourse, and individuals should correspond to actual entities that can be grouped into these classes.

5.4.3 Properties

OWL provides two different types of properties:

Object Properties

These properties are used to relate instances to other instances, e.g. X *hasParameter* Y, with X being instance of class *Device* and Y of class *Parameter*.

Datatype Properties

Datatype properties relate instances to datatype values, i.e. RDF literals or XML Schema datatypes. A typical device property could be *numberOfParameters*, which holds a non-negative integer as its value.

As seen in Chapter 4, properties can be restricted in their domain and range. Even hierarchical structures are possible by using `rdfs:subPropertyOf`. These restriction techniques also apply to OWL properties, as shown in Listing 5.8.

Listing 5.8: Simple OWL Object Property

```
<owl:ObjectProperty rdf:ID="hasParameter">
  <rdfs:range rdf:resource="#Parameter"/>
  <rdfs:domain rdf:resource="#Device"/>
  <owl:inverseOf rdf:resource="#isParameterOf">
</owl:ObjectProperty>
```

Furthermore, OWL also allows to define inverse properties by using `owl:inverseOf`, e.g. *hasParameter* and its inverse *isParameterOf*. Domain and range of the inverse property can be inherited, but they have to be interchanged.

Similar to `owl:equivalentClass`, OWL also provides a possibility to define the equivalence of properties by using the element `owl:equivalentProperty`.

5.4.4 Property Restrictions

Cardinality

In addition to these simple properties, OWL also allows to express more complex restrictions, like cardinality. For example, an *OutputDevice* must have at least one *OutputChannel*. This statement can be defined as following.

Listing 5.9: Cardinality Restriction

```
<owl:Class rdf:ID="OutputDevice">
  <rdfs:subClassOf rdf:resource="#Device" />
  <rdfs:subClassOf>
    <owl:Restriction>
        <owl:onProperty rdf:resource="#numberOfOutputChannels" />
        <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">
          1
        </owl:minCardinality>
```

```
      </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>
```

As shown in Listing 5.9, restrictions are implemented by using the subclass-relation. The reason is `owl:Restriction` defines an unnamed class - the class of things which have at least one output channel. Therefore, each member of the class OutputDevice is also a member of this unnamed class and must adhere to this restriction.

In addition to this, the cardinality value has been specified as *nonNegativeInteger*, which uses the XML Schema namespace declaration made in the document header. OWL supports a lot of XML Schema datatypes. A complete list of the recommended datatypes can be found in Appendix C.1

`owl:allValuesFrom`

Properties may also have restrictions only in combination with specific classes. For example, suppose a class *ControllerDevice*, which has parameters of class *SystemParameter* - a subclass of *Parameter*. If we want to state that all parameters of a *ControllerDevice* must be of type *SystemParameter*, we can use `owl:allValuesFrom` in order to restrict the property *hasParameter*. The fact that all parameters must be of this type, makes this restriction act like a *universal quantifier*.

`owl:someValuesFrom`

Additionally, OWL also provides a restriction comparable to the existential quantifier called `owl:someValuesFrom`. In correspondence to the previous example, a *ControllerDevice* must have a parameter of type *SystemParameter*, but it may also have other types of parameters. Listing 5.10 shows how to use this property restriction.

Listing 5.10: owl:someValuesFrom Restriction

```
<owl:Class rdf:ID="ControllerDevice">
  <rdfs:subClassOf rdf:resource="#Device" />
  <rdfs:subClassOf>
    <owl:Restriction>
        <owl:onProperty rdf:resource="#hasParameter" />
        <owl:someValuesFrom rdf:resource="#SystemParameter" />
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

`owl:hasValue`

Furthermore, the constraint `owl:hasValue`, allows to link a restriction class to either an individual of a data value.

5.4.5 Property Characteristics

In addition to restrictions, we can also describe property characteristics. This additional information is very important for efficient reasoning. The following properties are supported by OWL.

`owl:TransitiveProperty`

We have already mentioned the subclass relation in Chapter 4, which represents a transitive relation. The associated logic of transitive properties is shown below:

$$P(x,y) \ \wedge \ P(y,z) \ \Rightarrow \ P(x,z)$$

For example, the property *isPartOf* may be defined as transitive, because if x *isPartOf* y and y *isPartOf* z, x *isPartOf* z, too.

`owl:SymmetricProperty`

Properties that are characterized as being symmetric, adhere to the following relation:

$$P(x, y) \text{ if, and only if } P(y, x)$$

Typical members are *isSiblingOf* or *isAdjacentDevice*.

`owl:FunctionalProperty`

Functional properties are a bit more sophisticated. The associated logic is shown below:

$$P(x, y) \;\wedge\; P(x, z) \;\Rightarrow\; y = z$$

Suppose a relation *isSerialnumberOf*, which states x isSerialnumberOf y and x isSerialnumberOf z. Since serialnumbers are usually unique, this implies $y = z$.

`owl:InverseFunctionalProperty`

Inverse functional properties are used, when different objects cannot have the same value, e.g. *hasSerialnumber*

$$P(y, x) \;\wedge\; P(z, x) \;\Rightarrow\; y = z$$

5.4.6 Boolean Combinations

OWL provides predefined vocabulary in order to implement basic boolean operations. The resulting complex classes can also be nested, even without creating names for intermediate classes. The following operations are supported by OWL.

`owl:intersectionOf`

OWL allows to create an intersection of classes or properties by using `owl:intersectionOf`. For example, a device class *MultiDevice*, may be defined as intersection of the two classes *OutputDevice* and *InputDevice*. Instances of *MultiDevice* are also instances of the two other classes. The appropriate usage of `owl:intersectionOf`, is shown in the listing below.

Listing 5.11: owl:intersectionOf

```
<owl:Class rdf:ID="MultiDevice">
  <rdfs:subClassOf>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:ID="OutputDevice"/>
        <owl:Class rdf:ID="InputDevice"/>
      </owl:intersectionOf>
    </owl:Class>
  </rdfs:subClassOf>
  <rdfs:subClassOf rdf:resource="#Device"/>
  <rdfs:comment xml:lang="en">Class of devices having input and output
      channels</rdfs:comment>
</owl:Class>
```

`owl:unionOf`

In contrast to `owl:intersectionOf`, OWL also provides a way to define the union of classes. Syntax and usage correspond to Listing 5.11.

`owl:complementOf`

Another construct provided by OWL is `owl:complementOf`. This selects all individuals, that do not belong to a specified class. Usually this involves a lot of individuals and is typically used in combination with other boolean operations.

5.4.7 Enumerations

Classes can also be defined by simply enumerating its members. This can be done using `owl:oneOf`. An enumeration represents a closed set of members, therefore no other individuals can be valid members of this class.

The following listing has been created with Protégé[1] - a very popular ontology editor. This tool will also be used later on in Chapter 7.

Listing 5.12: OWL Enumerations

```
<owl:Class rdf:ID="PressureUnit">
  <rdfs:subClassOf rdf:resource="#Parameter"/>
  <owl:equivalentClass>
    <owl:Class>
      <owl:oneOf rdf:parseType="Collection">
        <PressureUnit rdf:ID="psi"/>
        <PressureUnit rdf:ID="bar"/>
        <PressureUnit rdf:ID="mbar"/>
      </owl:oneOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>
```

[1]http://protege.stanford.edu/

Chapter

6 Ontologies In Factory Automation

The bottom line is that automation lowers the risk of human error and adds some intelligence to the enterprise system. (Stephen Elliott)

6.1 Introduction

Nowadays, production engineering devices are characterized by increasing complexity and functionality. Additionally, life cycle times are getting shorter and products have to be highly flexible to suit different application scenarios. At the physical level, these conditions can be achieved by using modular components, e.g. *Festo CPX* devices. At the software level, this is more sophisticated.

In former times, system integrators were forced to use a lot of different configuration tools. However, since it is very inefficient to write specific software for each device, description languages have been developed, like the *Electronic Device Description Language* (EDDL[1]).

[1] http://www.eddl.org/

The EDDL was presented as layer 8 of fieldbus technology in 1991. The idea was to have a single tool, which is capable of configuring different devices by interpreting their *Electronic Device Description* (EDD) file. This file can be used to described the following characteristics:

- Description of device parameters and data structure
- Description of device functions
- Support for graphical representations, e.g. charts
- Support of communication with control devices

This concept of a vendor independent device description is very old, but still not fully reached yet. A lot of different solutions have been presented the last years, e.g. DTM/FDT[2].

A *Device Type Manager* (DTM) can be seen as some sort of driver, which gives detailed information about a specific device. This DTM is usually installed like a separate program, but can only be used within a frame application. At the application level, a standard called *Field Device Tool* (FDT) has been introduced. Unlike the name suggests, FDT does not refer to a tool. In fact, it provides an interface definition for the proper use of DTMs.

As mentioned before, the intended goal of these standards is to have a single application, that can be used to configure different devices. At first glance, DTM/FDT provides all these features, but every device still needs a specific DTM in order to work properly. Therefore, there is still room for further improvements.

As a matter of fact, device description has become a very important part of software development in industrial automation. Although this process is very time-consuming, trends

[2]http://www.fdtgroup.org/en/home-en.html

indicate that manufacturers are migrating towards knowledge-based concepts. Therefore, a possible future method of describing industrial devices, might also include semantic web techniques.

6.2 The Next Generation Device Description

Future devices are supposed to interact and collaborate autonomously. A precondition is, that devices share a global understanding about the environment and about themselves. Therefore, using the same syntax and semantics is essential in order to establish a common knowledge base. This knowledge can later be used to infer data about other devices. The result will be an *intelligent device*, which provides inference, reasoning and even learning skills.

One significant drawback of a standard-based approach - like the one presented previously using DTM/FDT - is, that standards are established on currently available knowledge. However, this may rule out newer devices that were not considered, when the standard has been created. This problem becomes even more important, regarding three current trends identified by [MLD].

> Firstly, a homogeneous solution to manufacturing technology is unlikely to arise, given the diversity of domains of application, the diversity of benefits associated to different approaches, [...]. Secondly, the diversity of devices for manufacturing is likely to increase as more specialization fields and new processes arise. Thirdly, technological evolution and introduction of new tools will continuously add unknown elements to the existing technology base.

Therefore, a Semantic Web approach may be a possible way to overcome these difficulties. RDF/OWL are content independent and not based on former made assumptions. Furthermore, OWL provides the following advantages.

- It is mature language, which offers a solid specification.
- It is suited for distributed environments, which is a prerequisite for the use in industrial automation devices.

6.3 A Common World

Once a well-defined ontology has been established, it can be used in different ways. This is illustrated in Figure 6.1.

First, an ontology can be used as knowledge base for an online shop. It provides every information that may be relevant for customers, e.g. number of output channels, device parameters, etc.

Secondly, the stored information is also of great importance for system integrators. The configuration tool can interpret the data in order to set up devices properly. This includes parameterization and configuration of devices.

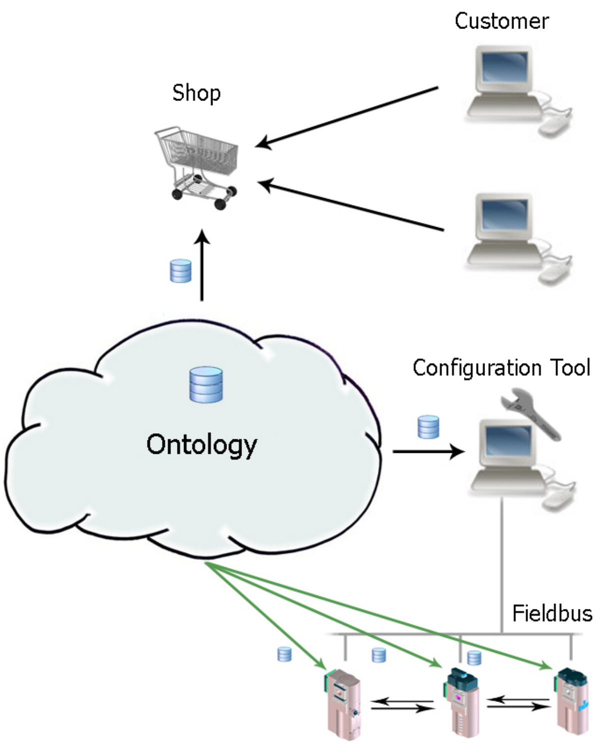


Figure 6.1: A Common Knowledge Base

Finally, there is a trend to use description data also in product stages like operation and maintenance. Future devices shall even be able to publish their features. Other interacting devices can read this information and compute a way to achieve a common goal.

6.4 Service-Oriented Architecture (SOA)

Additionally to a knowledge representation language like OWL, an appropriate mechanism to advertise this information is needed. In Section 6.2 we have already mentioned the suitability for distributed environments.

The structure of a distributed environment is represented by the *Service-Oriented Architecture*[3]. A SOA can be defined as a collection of services, which communicate with each other. This communication can include simple data exchange or the coordination of some activity. The connection between one or more services is a prerequisite for a Service-Oriented Architecture.

These services are usually implemented using Web Services (see Section 6.5). Furthermore, a protocol is needed in order to enable data exchange. One of these protocols is called *Service-Oriented Architecture Protocol* (SOAP[4]).

6.4.1 SOAP

SOAP is used to exchange XML-based messages, usually tunneled via protocols like HTTP or HTTPS (see also Figure 6.2). Critics argue that these protocols are not intended for the use with SOAP. In fact, using HTTP has a lot of advantages. For example, it makes communication via proxies and through firewalls easier, but SOAP also benefits from its wide deployment and acceptance. Nearly every programming language and platform offers

[3]http://en.wikipedia.org/wiki/Service-oriented_architecture
[4]http://en.wikipedia.org/wiki/SOAP

communication routines for HTTP, there is no need to implement any proprietary protocols. It is even possible to use encrypted data transport via HTTPS. Furthermore, this tunneling technique provides great flexibility at the datalink layer.

SOAP supports several message types. The most prominent example is the *Remote Procedure Call* (RPC).

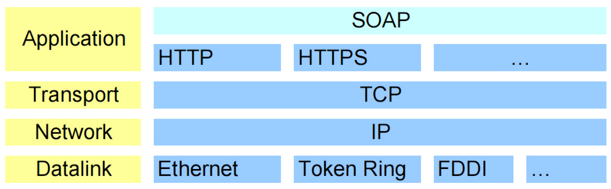


Figure 6.2: SOAP Layer

6.5 Web Services (WS)

The idea is to use *Web Service*[5] techniques, which are especially designed for the use in distributed environments. The W3C Working Group defines Web services as following:

> A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL[6]).[W3Ch]

However, a Web Service encapsulates a process. This can be a software process or even a physical process. Usually a WS provides some sort of Web API that can be accessed remotely. These interface specifications are also called *Advertisements*. An advertisement gives information about the used syntax of exchanged messages.

[5]http://en.wikipedia.org/wiki/Web_service
[6]http://www.w3.org/TR/wsdl

6.6 Web Service Description Language (WSDL)

The language used to formalize such advertisements is called *Web Service Description Language*. WSDL defines *Messages* and *Ports*. A *Port* is linked to a specific network address, whereas *Messages* represent an abstract definition of the data being exchanged by the Web services. Once a WSDL file has been created to describe a specific interface, manual adaptations have to be made in case of service alteration. In order to overcome these problems, Semantic Web techniques - like OWL-S - may offer a possible solution. Figure 6.3 illustrates the concept of WSDL.

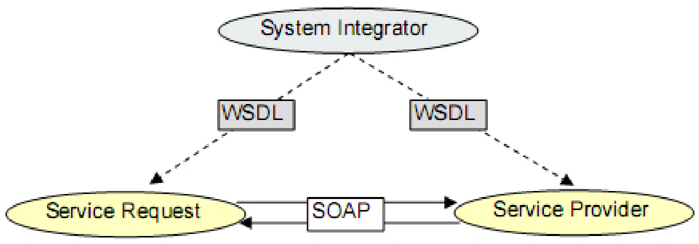


Figure 6.3: WSDL Concept

6.7 OWL-S

Regarding the W3C Submission, "OWL-S is an OWL-based Web service ontology, which supplies a core set of markup language constructs for describing the properties and capabilities of Web services in unambiguous, computer-interpretable form." A service described by OWL-S is called a Semantic Web Service (SWS). [MBH+04] identifies three task types OWL-S is expected to enable.

Automatic Web Service discovery

Automatic Web Service discovery describes the automated detection of a WS that is capable of performing a particular task.

Automatic Web Service invocation

Automatic Web Service Invocation describes the automated service invocation by a computer program or an agent. This is possible, due to a detailed description of how the WS has to be used. Special software should then be able to interpret this description in order to remotely call the service afterwards.

Automatic Web Service composition and interoperation

Automatic Web Service composition and interoperation describes the automated selection, composition and interoperation of WS to perform a complex task, given only a high-level description of an objective.

6.7.1 OWL-S structure

OWL-S provides three different types of knowledge to describe a Semantic Web Service. These three types are covered by the following questions.

- What does the service provide?
- How is it used?
- How does one interact with it?

ServiceProfile

The *ServiceProfile* gives information about what the service provides. This enables a software agent to clarify if a certain service meets its needs by interpreting this information. Furthermore, the *ServiceProfile* includes a description of the capabilities of the service, its limitations and the provided service quality.

ServiceModel

The *ServiceModel* is used by the service requester to gather information about how to use the service. In other words, it provides a description of how a service is invoked.

ServiceGrounding

The *ServiceGrounding* provides information about the communication details. Typically it describes a protocol, message formats, port numbers, etc.

A class *Service* contains three properties, corresponding to these knowledge types. The properties are *presents*, *describedBy* and *supports*. This relation is illustrated in Figure 6.4.

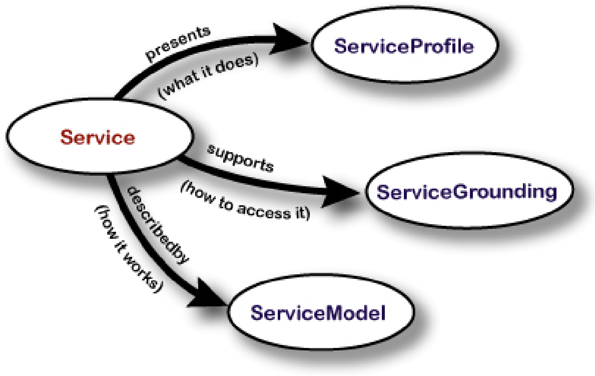


Figure 6.4: OWL Service Ontology

This technique to publish service capabilities could be used in industrial automation to create a self-describing device interface. This evolving class of *intelligent modules* would be able to communicate and interact autonomously. This would also highly improve flexibility and even enable some sort of Plug&Play. New devices will be recognized automatically and time-consuming integration processes will not be necessary anymore.

Chapter

7 Festo Device Ontology Implementation

We live in a time when automation is ushering in a second industrial revolution. (Adlai E. Stevenson)

7.1 Introduction

This chapter will introduce the Festo[1] *CPX* device family. These devices are characterized by the possibility to combine electric, pneumatic and controller devices on a single rack.

In the following sections an ontology will be developed, which enables the *Festo Configuration Tool* (FCT) to configure these devices. At the moment the needed information is stored in ordinary INI-Files.

The FCT allows system integrators to configure and maintain nearly all Festo devices. In former times, several different tools were needed to accomplish this task. The solution is

[1]http://www.festo.com/

a plug-in based framework, where each device family is represented by its corresponding plug-in.

The following sections will give a detailed insight in ontology development. Furthermore, common mistakes and general development rules will be outlined. Section 7.2 gives a brief overview about the intended goal. Afterwards, a possible design approach will be presented.

7.2 Overview

As illustrated in Figure 7.1, the ontology will only be used within the configuration tool at the moment. Nevertheless, it is also feasible to use this knowledge in a shop system later on.

An implementation at the fieldbus level would go beyond the scope of this thesis. However, the fact that the functionality of devices is rapidly increasing supports the need for other knowledge representation techniques at this level.

The INI-File will be used as starting point to create the corresponding device ontology. It defines 26 devices, including parameters and other information.

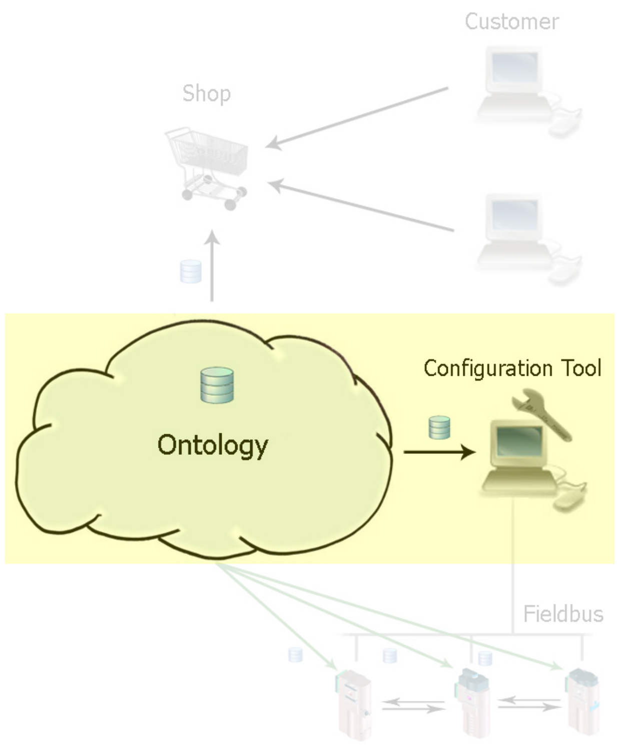


Figure 7.1: Application overview

7.3 The Design Process

First, there is no mandatory way of ontology modeling. Different approaches are feasible, the best solution depends on the application it is intended for. Nevertheless, there are three fundamental rules, which may ease some design decisions.

1. As already mentioned there is no 'correct' way in ontology design - it depends on the application the ontology will be used.
2. Developing an ontology is an iterative process.
3. It is recommended to map real world objects and their relations as directly to ontology concepts as possible.

> We also need to remember that an ontology is a model of reality of the world and the concepts in the ontology must reflect this reality. [NM01]

The second rule states, that ontology development is an iterative process. Starting with a quite rough release, the ontology will be revised and refined afterwards. The document *Ontology Development 101* [NM01] suggests a 7 step approach, which will now be discussed.

7.3.1 Step-by-Step

Step 1: Determine the domain and scope of the ontology

At the beginning it is judicious to sketch a list of the intended goals. The following questions help to track down essential ontology characteristics.

- Which domain should the ontology cover?

- Which application will use the ontology?
- Which answers should the ontology provide?

Step 2: Reusing existing ontologies

Ontologies are designed for distributed usage. Therefore, if appropriate ontologies are already available they should be used. Unfortunately, there are no ontologies that cover devices in factory automation or something similar.

Step 3: List of relevant ontology concepts

The next step is to write down information about important terms. For example, concepts like *Device* or *Parameter*, but this also includes relevant properties like *numberOfChannels*, *hasDeviceBitmap*, etc.

Within the next two steps a hierarchical structure will be established and class properties will be defined. In practice, these steps are performed simultaneously. However, they represent the most important part of ontology development.

Step 4: Defining classes and establishing a class hierarchy

Regarding to [UG96], there are three different ways to develop a class hierarchy.

- **Top-down**
  A top-down approach starts with the definition of the most abstract (top-level) concepts, e.g. *Device*. Specializations are defined later on. This usually results in better control of the detail level. Nevertheless, this approach may also introduce unnecessary high-level categories.

- **Bottom-up**

  In contrast, the bottom-up method starts with the most specific classes. This approach is normally associated with a very high detail level. The drawback is the increasing difficulty to track down commonalities between concepts.

- **Middle-Out**

  According to [UG96], the middle-out approach is a good trade-off concerning the level of detail and the naturally arising generalizations. For example, we can take the class *DigitalModule* and start refining it. Afterwards we try to generalize the concept, e.g. *DigitalModule* is a direct subclass of *ElectricModule*.

Step 5: Defining class properties

Once a class hierarchy is established, properties describing relations between those classes have to be created. In Step 3, a list of the properties has already been defined.

The task is now to attach a property to the most general class possible. For example a property *numberOfChannels* could be attached to *ElectricDevice*. It would be false to assign it to the class *Device*, because there is also a subclass *ControllerDevice*, which may contain devices that do not have any channels.

Class properties are inherited to subclasses. Therefore, a properly defined class hierarchy is an absolutely essential prerequisite.

Step 6: Further property refinement

Properties can be further enriched by adding information about their characteristics. These relations have already been mentioned in Sections 5.4.3, 5.4.4 and 5.4.5. For example, the property *hasDeviceName* may have a minimum cardinality of 1, but since the ontology provides support for different languages it may also have more than a single name.

Furthermore, it is possible to describe the types of values a datatype property may hold. The most common types are *Boolean*, *Int*, *String* and *Float*.

Another important information is the *domain* and *range* of a property (see also Section 4.2). [NM01] refers to properties as *slots* and suggests the following approach concerning the *domain* and *range*:

> When defining a domain or a range for a slot, find the most general classes or class that can be respectively the domain or the range for the slots. On the other hand, do not define a domain and range that is overly general: all the classes in the domain of a slot should be described by the slot and instances of all the classes in the range of a slot should be potential fillers for the slot. Do not choose an overly general class for range (i.e., one would not want to make the range THING) but one would want to choose a class that will cover all fillers.

Step 7: Creating instances

The final step is the creation of instances. Once the class to hold the instance is chosen, the property values have to be defined.

7.3.2 Common Mistakes

Singular vs. plural

A common mistake in ontology development is, that people mix up singular and plural terms. It is also false to define a class *Device* as subclass of *Devices*. This is why it is better, to interpret the subclass-relation as *kind-of*-relation in order to avoid such mistakes. Although the usage of plural terms is possible, the use of singular terms is suggested, because sometimes plural terms may be confusing.

Concepts and names

Classes and their names have to be considered separately. For example, it is possible that different names describe the same class and its concept, respectively. Sometimes, it is also possible to associate synonyms with a concept to solve this problem. For example a *Device* may also be referred to as *Module*.

Class cycles

It is not allowed to define a class A as subclass of B and class B as subclass of A. This would result in a cycle within the hierarchy.

A balanced tree

There is no limitation concerning the number of subclasses a class may have. Nevertheless, a balanced structure is quite important. [NM01] presents two basic rules to overcome this problem.

- If a class has only one direct subclass there may be a modeling problem or the ontology is not complete.
- If there are more than a dozen subclasses for a given class then additional intermediate categories may be necessary.

New class or property value?

Another difficult design decision to make is, whether it is better to introduce a new class or store information in a property value. Suppose we have a class *ElectricDevice* containing devices with a boolean datatype property *isOutputDevice* and all devices having this property set to *true*. It is possible to introduce a subclass *OutputDevice* in order to omit this property.

Instance vs. class

It is also very challenging to decide whether a specific concept should be implemented as class or individual. The question is *what are the most basic items the ontology should represent?* Usually this is predefined by the application that uses the ontology afterwards.

Limited scope

Finally, an ontology should be able to provide all the information that may be needed. However, too much information may result in a hardly understandable ontology. Therefore, the creation of a well-balanced knowledge base is very important.

7.3.3 Naming Conventions

Capitalization

In order to make an ontology more readable, a consistent naming convention is necessary. A very common method is to capitalize class names and use lower case letters for properties.

Delimiters

As already mentioned earlier, Protégé does not allow the use of spaces in concept names. Instead an underscore or dash may be used. It is also very common to omit delimiters and capitalize the first letter of each new word.

7.4 Tools and Software

Due to the inherent complexity of ontologies, appropriate tool support is absolutely necessary. Furthermore, manual ontology development is an error-prone process. Fortunately, there are already tools that make this task a lot easier.

7.4.1 Protégé

The most popular and powerful tool for ontology development is called Protégé. Protégé is a free, open source editor providing a framework that can be extended by third-party plug-ins. It is written in Java and maintained by the Stanford University and the University of Manchester.

Protégé can be used to create domain-models and knowledge collections. It supports the creation, manipulation and visualization of ontologies in different formats, including OWL/RDF.

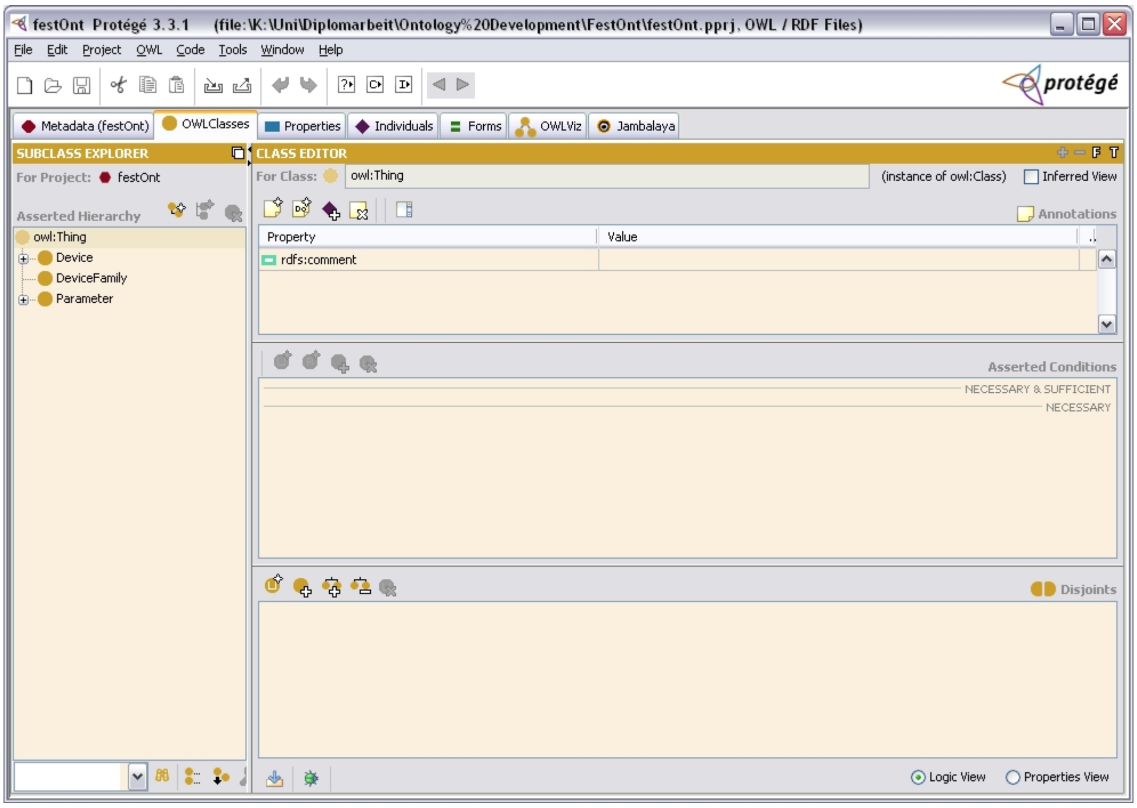


Figure 7.2: The ontology editor Protégé

As illustrated in Figure 7.2, Protégé uses tabs to keep OWL classes, properties and instances separated. The tabs *'OWLViz'* and *'Jambalaya'* represent two of the many available plug-ins for Protégé. These visualization plug-ins provide a good possibility to keep track of growing ontologies.

Protégé automatically generates OWL files. There is no need to have detailed knowledge about complex OWL structures.

7.4.2 OwlDotNetApi

Most of the *Semantic Web* tools are written in Java. The *OwlDotNetApi*[2] is written in C# and provides an API for .NET applications. It is based on *Drive*, a .NET based RDF parser. The API uses the *Drive* data model to create a directed linked graph. The corresponding RDF parser has been modified to parse OWL instead. *OwlDotNetApi* provides the following features:

- C# based RDF parser for the .NET platform
- Compliant with the OWL syntax specification
- Builds a directed linked graph
- Can be used with any .NET language
- Merges graphs from multiple sources
- Simple generator using the visitor pattern

[2]http://users.skynet.be/bpellens/OwlDotNetApi/index.html

The OwlDotNetApi is available as pre-compiled DLL or as sourcecode. The FCT project does not allow the usage of assemblies without a strong name[3]. Therefore, the sources had to be compiled again, because it is not possible to sign (strong name) compiled assemblies afterwards.

Assemblies can be signed using different ways in Visual Studio, either by invoking the project properties or by adding the following lines to the file *AssemblyInfo.cs*.

Listing 7.1: Signing assemblies

```
[assembly: AssemblyKeyFile("..\\..\\<filename>.snk")]
[assembly: AssemblyKeyName("<filename>.snk")]
```

Appendix C.2 and C.3 give further information about the provided classes and interfaces of the *OwlDotNetApi*.

7.5 The Festo Device Ontology

7.5.1 Step 1: Determine the domain and scope of the ontology

The intended goal of the Festo Device Ontology, is to provide detailed information about devices used in industrial automation. Even though the ontology also stores information that could be usefully integrated into a Web shop system, the primary objective is the usage within the *Festo Configuration Tool*. This includes knowledge about the devices, their parameters and other relevant concepts.

[3]http://msdn2.microsoft.com/en-us/library/wd40t7ad.aspx

7.5.2 Step 2: Reusing existing ontologies

Unfortunately, there are no ontologies that can be reused within this area of interest.

7.5.3 Step 3: List of relevant ontology concepts

It is very difficult to identify all concepts that may be needed later on. The relevance of important concepts often emerges in subsequent design stages. The best practice in ontology design is to stick to real world concepts and their inherent relations. In the case of the *Festo Device Ontology*, this means to map the existing INI-File structure to the ontology structure. The following table provides a list of the needed concepts, including those identified in subsequent steps.

Classes	**Properties**
Device	isOfDeviceClass
ElectricDevice	hasDeviceCode
ControllerDevice	hasDeviceName
PneumaticDevice	hasDeviceBitmap
AnalogueDevice	hasDeviceDescription
DigitalDevice	hasParName
HybridDevice	hasParMinValue
Parameter	hasParMaxValue
DeviceParameter	hasParFormat
SystemParameter	hasParData
ParameterText	hasSysParData
ParText	hasDefaultValue
DeviceFamily	hasText

	hasParameter
	hasParText
	isFollowedByParText
	ofDeviceFamily

Table 7.1: List of relevant concepts

7.5.4 Step 4: Defining classes and establishing a class hierarchy

The list of needed classes is not very long. Therefore, a proper hierarchy can easily be introduced. The top-level nodes are *Device*, *Parameter*, *ParameterText* and *DeviceFamily*. The separation of *Parameter* and *ParameterText* is based on the underlying structure of the INI-file.

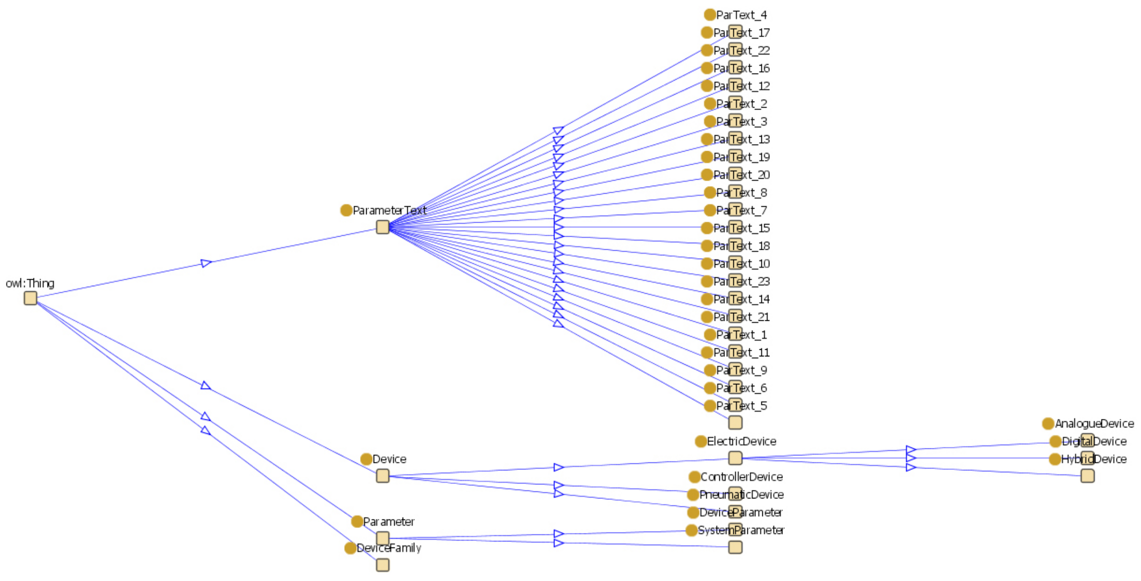
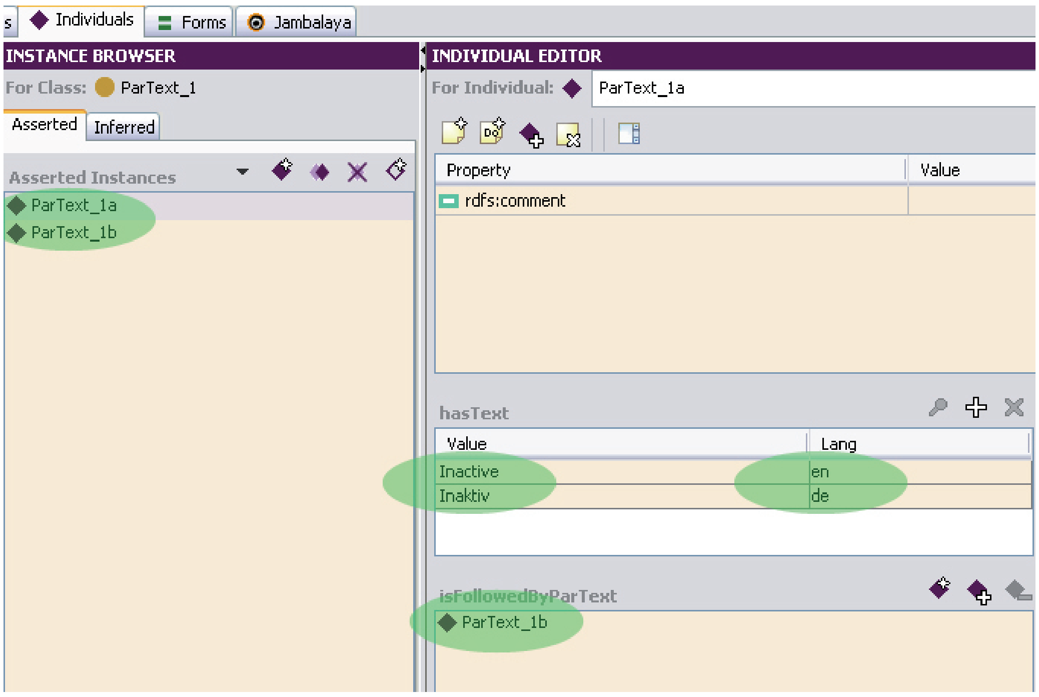


Figure 7.3: Ontology Hierarchy

7.5.5 Step 5: Defining class properties

The definition of properties is a bit more sophisticated. Datatype properties like *hasDeviceCode* are easier to identify than object properties that represent relations between instances, e.g. *hasParameter*. Most of the properties can be extracted directly from the INI-file, but in order to link the *Device* class to *Parameter*, a property *hasParameter* is needed. This also applies to the property *hasParText*, which is used to link text like *inactive*, to the Parameter class.

For example, we have a class *ParText_1*, which contains the two instances *ParText_1a* and *ParText_1b*. *ParText_1a* contains the properties *hasText* and *isFollowedByParText* as illustrated in Figure 7.4.



For Class: ParText_1

ParText_1a
ParText_1b

For Individual: ParText_1a

Property	Value
rdfs:comment	

hasText

Value	Lang
Inactive	en
Inaktiv	de

isFollowedByParText

ParText_1b

Figure 7.4: ParText Instance

The advantage of using a property *hasText* to represent the parameter text instead of directly using the instance name, is that we can easily define multilingual parameter texts. Furthermore, an instance name must not start with a number or symbol, but this does not apply to the values of data properties like *hasText*.

The property *isFollowedByParText* has been introduced, because OWL has no support for ordering. In fact, the RDF techniques that would solve this problem are not allowed within OWL-DL and are therefore not available within Protégé. More information dealing with this problem can be found in [DRS+]. However, we can use *isFollowedByParText* in order to implement a linked list of parameter texts. The list is terminated by an empty parameter instance, as shown in Figure 7.4.

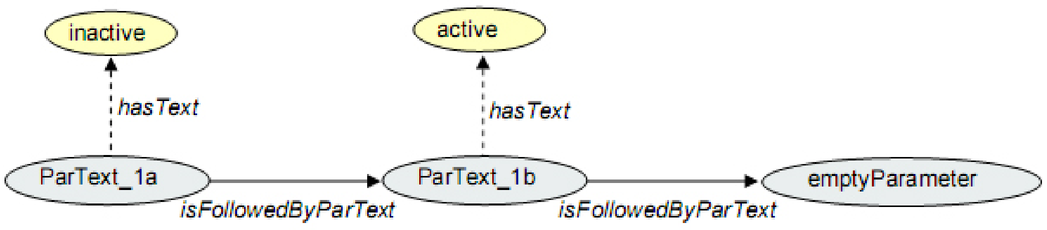
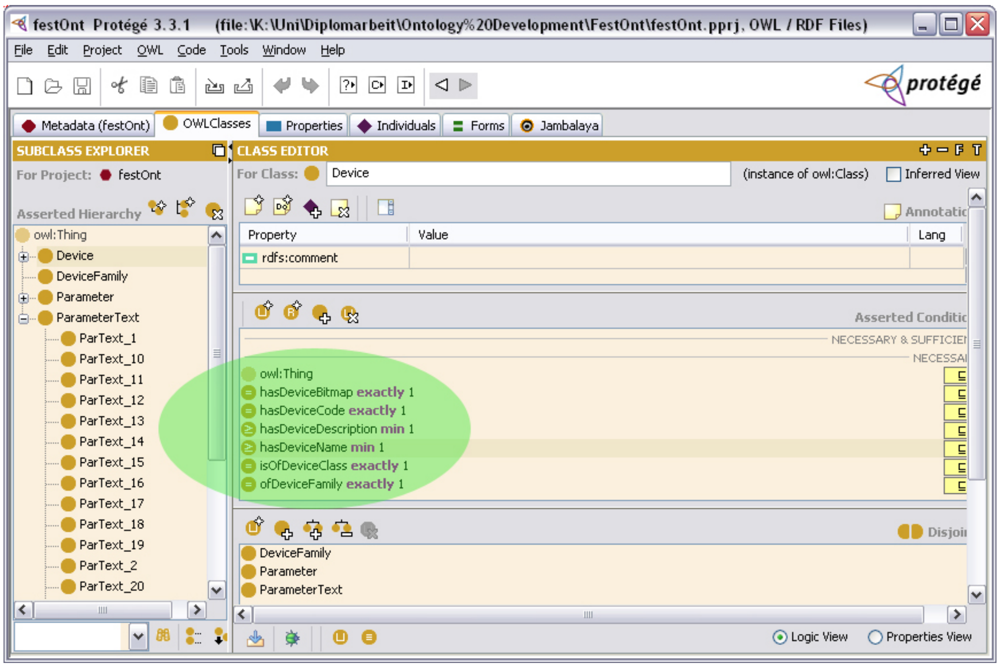


Figure 7.5: Introducing a Sequence Mechanism

7.5.6 Step 6: Further property refinement

This step deals with additional property refinement including cardinality, data type restrictions, domain and range. Providing restrictions is the best way to implement fault prevention. For example, every device belongs to a specific device family. Therefore, it could be useful to implement a property restriction, stating that every *Device* has exactly one property *ofDeviceFamily*. This is illustrated in Figure 7.6.



For Project: festOnt

owl:Thing
- Device
- DeviceFamily
- Parameter
- ParameterText
  - ParText_1
  - ParText_10
  - ParText_11
  - ParText_12
  - ParText_13
  - ParText_14
  - ParText_15
  - ParText_16
  - ParText_17
  - ParText_18
  - ParText_19
  - ParText_2
  - ParText_20

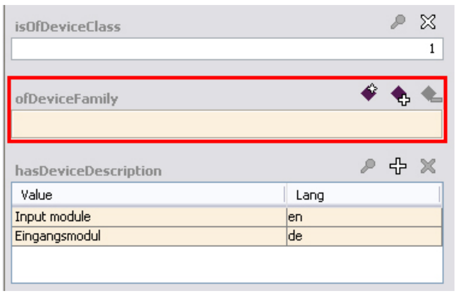
For Class: Device (instance of owl:Class) Inferred View

Property	Value	Lang
rdfs:comment		

owl:Thing
hasDeviceBitmap exactly 1
hasDeviceCode exactly 1
hasDeviceDescription min 1
hasDeviceName min 1
isOfDeviceClass exactly 1
ofDeviceFamily exactly 1

DeviceFamily
Parameter
ParameterText

Figure 7.6: Property Refinement


isOfDeviceClass
1

ofDeviceFamily

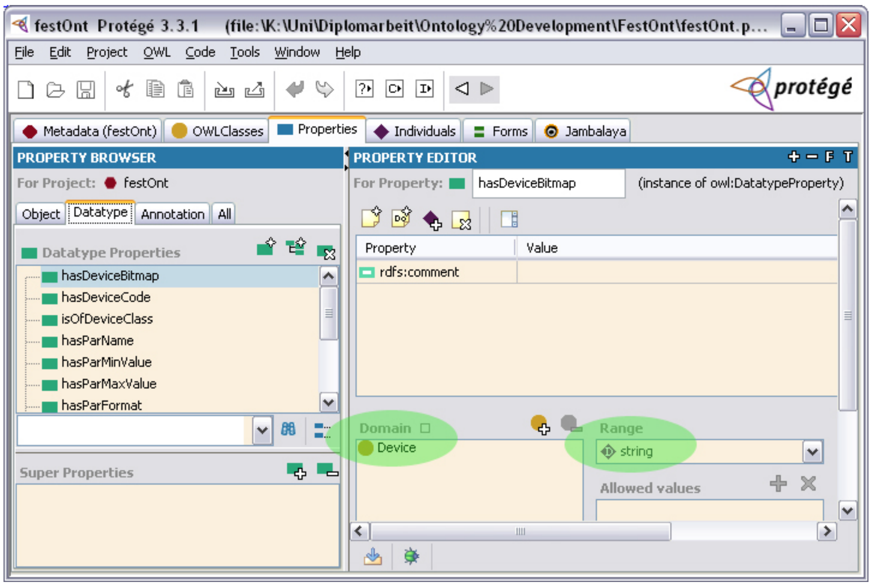
hasDeviceDescription

Value	Lang
Input module	en
Eingangsmodul	de

Figure 7.7: Restriction Violation

If a property restriction is violated, this is indicated by Protégé as shown in Figure 7.7.

Furthermore, it is possible to define range and domain of a property as mentioned in Section 4.2.1 and 4.2.2. Object properties have classes as range, whereas datatype properties have values of string, int, etc. For example, we can define the range of *hasDeviceCode* and *isOfDeviceClass* as int.


festOnt Protégé 3.3.1 (file:\K:\Uni\Diplomarbeit\Ontology%20Development\FestOnt\festOnt.p...
File Edit Project OWL Code Tools Window Help
protégé
Metadata (festOnt) OWLClasses Properties Individuals Forms Jambalaya
PROPERTY BROWSER
PROPERTY EDITOR
Object Datatype Annotation All
Datatype Properties
Super Properties
Property Value
Allowed values

For Project: festOnt

For Property: hasDeviceBitmap (instance of owl:DatatypeProperty)

hasDeviceBitmap
hasDeviceCode
isOfDeviceClass
hasParName
hasParMinValue
hasParMaxValue
hasParFormat

rdfs:comment

Domain: Device

Range: string

Figure 7.8: Domain and Range Restrictions

7.5.7 Step 7: Creating instances

The last step is to populate the ontology with instances, which is very time-consuming. However, it is a lot easier if the property restrictions have been defined precisely.

Finally, the Festo Device Ontology contains 21 devices, 2 device families, 35 parameters and 86 parameter texts. Once all devices are implemented, we can start to integrate the ontology into the *Festo Configuration Tool*.

7.6 Integration Process

At the moment the FCT uses a C# class called *IniFileParser.cs*, which extracts all the needed information from two separate INI-files - an English and a German version.

In order to use the *OwlDotNetApi*, we must first import the pre-compiled DLL into the *Visual Studio* project. The next step is to create an OWL graph from the ontology file. This should be done only once, because otherwise it would significantly decrease the program's performance. Therefore, the needed objects should be declared *static* and *readonly* as shown in Listing 7.2.

Listing 7.2: Creation of the OWL Graph

```
static readonly IOwlParser parser = new OwlXmlParser();
static readonly IOwlGraph graph = parser.ParseOwl("D:\\Uni\\Diplomarbeit\\
    Ontology_Development\\FestOnt\\festOnt.owl");
```

Once an OWL graph has been created, further methods can be implemented. Experience has shown, that it is useful to have a method *getChildEdges()* and *getParentEdges()*, which can be used to retrieve detailed information from the graph.

Listing 7.3: *getChildEdges()* and *getParentEdges()*

```
        private OwlEdgeCollection getChildEdges(string sNode)
        {
            IOwlNode owlNode = graph.Nodes[sNode];
            OwlEdgeCollection edges = (OwlEdgeCollection)owlNode.
                ChildEdges;
            return edges;
        }

        private OwlEdgeCollection getParentEdges(string sNode)
        {
            IOwlNode owlNode = graph.Nodes[sNode];
            OwlEdgeCollection edges = (OwlEdgeCollection)owlNode.
                ParentEdges;
            return edges;
        }
```

The FCT provides also a mechanism to scan for connected devices. Each device can be identified by its device code. It is necessary to create a list of all devices that belong to the specific device family, in order to be able to compare it with the found devices.

As mentioned before, the FCT is a plug-in based framework. Every device family is represented by a separate plug-in. The ontology implements devices of the Festo *CPX* and the *MPX* family. It would also be possible to implement devices of other families.

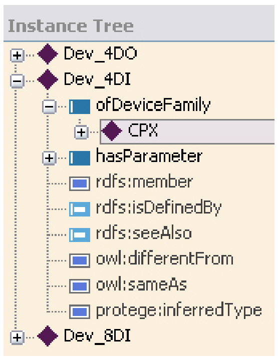


Figure 7.9: Instance Tree

The trick is to use a property *ofDeviceFamily*, which links each device to its corresponding device family instance. By invoking the method *getParentEdges* with the *CPX* family node, we get a list of edges pointing to all *CPX* device nodes. In a simple *foreach* loop we can retrieve all the information needed about the devices. The corresponding source code is shown in Listing 7.4.

Listing 7.4: getModulesByCode()

```
public Dictionary<int, string[]> getModulesByCode()
        {
            Dictionary<int, string[]> modulesByCode = new Dictionary<int,
                string[]>();

        OwlEdgeCollection edges = getParentEdges("http://www.festo.com
            /festOnt#CPX");

        foreach (OwlEdge e in edges)
        {
            string[] moduleInfo = new string[5];
            OwlEdgeCollection parentEdges = (OwlEdgeCollection)e.
                ParentNode.ChildEdges;

            foreach (OwlEdge eP in parentEdges)
            {
                if (eP.ID == "http://www.festo.com/festOnt#
                    hasDeviceBitmap")
                    moduleInfo[0] = eP.ChildNode.ID.Substring(0, eP.
                        ChildNode.ID.IndexOf("^^"));
                if (eP.ID == "http://www.festo.com/festOnt#
                    isOfDeviceClass")
                    moduleInfo[1] = eP.ChildNode.ID.Substring(0, eP.
                        ChildNode.ID.IndexOf("^^"));
                if (eP.ID == "http://www.festo.com/festOnt#
                    hasDeviceCode")
                    moduleInfo[2] = eP.ChildNode.ID.Substring(0, eP.
                        ChildNode.ID.IndexOf("^^"));
                if (eP.ID == "http://www.festo.com/festOnt#
                    hasDeviceDescription")
                    if (eP.LangID == "en")
                        moduleInfo[3] = eP.ChildNode.ID.Substring(0,
                            eP.ChildNode.ID.IndexOf("@"));
                if (eP.ID == "http://www.festo.com/festOnt#
                    hasDeviceName")
                    if (eP.LangID == "en")
                        moduleInfo[4] = eP.ChildNode.ID.Substring(0,
                            eP.ChildNode.ID.IndexOf("@"));
            }
```

```
                    modulesByCode.Add(Convert.ToInt32(moduleInfo[2]),
                        moduleInfo);
                }
                return modulesByCode;
            }
```

7.7 Conclusion

Semantic Web techniques offer a lot of new possibilities. Especially when retrieving specific information, a networked structure - like it is provided by OWL - is very advantageous. This is also illustrated in Listing 7.4, which extracts information about all devices that belong to the *CPX* family.

Due to the use of XML, ontology documents are difficult to read for humans. Furthermore, these documents are usually very big and it is hard to keep track of the stored data. XML is a solid and widespread language, that is well suited for automated processing. Unfortunately, it is dedicated for hierarchical structuring only. Therefore, additional features are needed in order to add dependencies to data. These features are provided by OWL/RDF.

Tool Support

The inherent complexity of OWL documents makes tool support a prerequisite. Protégé represents a good choice regarding the creation and modification of these documents. It is not necessary to have detailed knowledge about OWL anymore, since ontologies can be developed by using a well structured graphical user interface. Additionally, there are also a lot of third party plug-ins, like *Jambalaya*[4] - a data visualization tool.

[4]http://www.thechiselgroup.org/jambalaya

Language Support

As mentioned before, Festo uses two INI-file versions at the moment - a German and an English version. Besides the fact that it is hard to keep those files consistent, this method of storing information is also very inconvenient. However, OWL ontologies provide language support by using the `xml:lang` attribute.

Performance

Most of the available software dealing with OWL/RDF is written in Java. Since Java is an interpreted language, the program execution takes longer compared to compiled software. Therefore, the usage of a .NET/Java bridge will result in significant performance loss. Fortunately, there are already a few tools written in other languages, like the *OwlDotNetApi*. Nevertheless, the creation of a directed linked graph from the ontology document takes some time, which should be kept in mind. The best practice is to do this only once at the beginning in order to keep execution times low.

Embedded Systems

Performance and memory capacities are also an important topic regarding embedded systems. Device manufacturers have to keep the product prices low in order to remain competitive, but due to the high memory and performance requirements, semantically enriched devices will be more expensive than others. Therefore, as long as these devices do not imply any short term efficiency, customers will not have any reason to buy them. However, there are already methods like the Semantic Web Services, which may enable communication without any human interaction.

Semantic Web techniques offer some interesting possibilities in software development. Especially as central knowledge base for multiple applications it grants facilities. However, Semantic Web implementations have been approved already in various application areas, it remains to be seen, if this also applies to the field of industrial automation.

Appendix

A Acronyms

API	Application Programming Interface
CPX	Compact Performance Extension
DAML	DARPA Agent Markup Language
DL	Description Logic
DLL	Dynamic Link Library
DTD	Document Type Definition
DTM	Device Type Manager
EDD	Electronic Device Description
EDDL	Electronic Device Description Language

FCT Festo Configuration Tool

FDT Field Device Tool

FOL First Order Logic

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

IEEE Institute of Electrical and Electronics Engineers

ILI Interlingual Index

ISBN International Standard Book Number

I/O Input/Output

KIF Knowledge Interchange Format

KIM Knowledge and Information Management

KSL Knowledge System Labs

MCF Meta Content Framework

MILO Middle-Level Ontology

N3 Notation 3

OIL Ontology Inference Layer

OWL Web Ontology Language

RDF Resource Description Framework

RDFS Resource Description Framework Schema

RPC Remote Procedure Call

SHOE Simple HTML Ontology Extension

SOA Service Oriented Architecture

SOAP Service Oriented Architecture Protocol

SUO Standard Upper Ontology

SUMO Suggested Upper Merged Ontology

SWS Semantic Web Service

UML Unified Modeling Language

UNSPSC United Nations Standard Products and Services Code

URI Universal Resource Identifier

WS Web Service

WSDL Web Service Definition Language

W3C	World Wide Web Consortium
XML	Extensible Markup Language

Appendix

B RDF

B.1 RDF Class Descriptions

Class Name	**Description**
rdfs:Resource	The class resource, everything
rdfs:Literal	The class of literal values, e.g. strings or integers
rdf:XMLLiteral	The class of XML literal values
rdfs:Class	The class of all classes
rdf:Property	The class of RDF properties
rdfs:Datatype	The class of RDF datatypes
rdf:Statement	The class of RDF statements
rdf:Bag	The class of unordered containers
rdf:Seq	The class of ordered containers
rdf:Alt	The class of alternatives
rdfs:Container	The class of RDF containers
rdf:List	The class of RDF lists

Table B.1: RDF Classes [W3Cf]

B.2 RDF Property Descriptions

Property Name	Description
rdf:type	The subject is an instance of a class
rdfs:subClassOf	The subject is a subclass of a class
rdfs:subPropertyOf	The subject is a subproperty of a property
rdfs:domain	A domain of the subject property
rdfs:range	A range of the subject property
rdfs:label	A human-readable name for the subject
rdfs:comment	A description of the subject resource
rdfs:member	A member of the subject resource
rdf:first	The first item in the subject RDF list
rdf:rest	The rest of the subject RDF list after the first item
rdfs:seeAlso	Further information about the subject resource
rdfs:isDefinedBy	The definition of the subject resource
rdf:value	Idiomatic property used for structured values
rdf:subject	The subject of the subject RDF statement
rdf:predicate	The predicate of the subject RDF statement
rdf:object	The object of the subject RDF statement

Table B.2: RDF Properties [W3Cf]

B.3 RDF Property Domains and Ranges

Property Name	Domain	Range
rdf:type	rdfs:Resource	rdfs:Class
rdfs:subClassOf	rdfs:Class	rdfs:Class
rdfs:subPropertyOf	rdf:Property	rdf:Property
rdfs:domain	rdf:Property	rdfs:Class
rdfs:range	rdf:Property	rdfs:Class

rdfs:label	rdfs:Resource	rdfs:Literal
rdfs:comment	rdfs:Resource	rdfs:Literal
rdfs:member	rdfs:Resource	rdfs:Resource
rdf:first	rdf:List	rdfs:Resource
rdf:rest	rdf:List	rdf:List
rdfs:seeAlso	Frdfs:Resource	rdfs:Resource
rdfs:isDefinedBy	rdfs:Resource	rdfs:Resource
rdf:value	rdfs:Resource	rdfs:Resource
rdf:subject	rdf:Statement	rdfs:Resource
rdf:predicate	rdf:Statement	rdfs:Resource
rdf:object	rdf:Statement	rdfs:Resource

Table B.3: RDF Properties [W3Cf]

Appendix

C OWL

C.1 OWL: Supported XML Schema Datatypes

xsd:string	xsd:normalizedString	xsd:boolean	
xsd:decimal	xsd:float	xsd:double	
xsd:integer	xsd:nonNegativeInteger	xsd:positiveInteger	
xsd:nonPositiveInteger	xsd:negativeInteger		
xsd:long	xsd:int	xsd:short	xsd:byte
xsd:unsignedLong	xsd:unsignedInt	xsd:unsignedShort	xsd:unsignedByte
xsd:hexBinary	xsd:base64Binary		
xsd:dateTime	xsd:time	xsd:date	xsd:gYearMonth
xsd:gYear	xsd:gMonthDay	xsd:gDay	xsd:gMonth
xsd:anyURI	xsd:token	xsd:language	
xsd:NMTOKEN	xsd:Name	xsd:NCName	

Table C.1: OWL: Supported XML Schema Datatypes [W3Ca]

C.2 OwlDotNetApi Classes

Class	**Description**
InvalidOwlException	Represents an exception that is thrown when invalid OWL Syntax is encountered by the parser
OwlAnnotationProperty	Represents an OWL resource of type owl:AnnotationProperty
OwlClass	Represents an OWL resource of type owl:Class
OwlCollection	Represents an OWL resource of type rdf:List
OwlDatatype	Represents an OWL resource of type rdfs:Datatype
OwlDatatypeProperty	Represents an OWL resource of type owl:DatatypeProperty
OwlEdge	Represents an Edge in the OWL Graph
OwlEdgeCollection	Represents a collection of edges. This class maps edge IDs to lists of OwlEdge objects.
OwlEdgeList	Represents a collection of OwlEdge objects
OwlGenerator	Summary description for OwlGenerator
OwlGraph	Represents an OWL Graph
OwlIndividual	Represents an OWL resource of type owl:Individual
OwlLiteral	Represents a Literal in the OWL Graph
OwlNamespaceCollection	Represents a collection of Namespaces
OwlNode	Represents a node in the OWL Graph
OwlNodeCollection	Represents a collection of OWL Nodes
OwlObjectProperty	Represents an OWL resource of type owl:ObjectProperty
OwlOntology	Represents an OWL resource of type owl:Ontology
OwlOntologyProperty	Represents an OWL resource of type owl:OntologyProperty
OwlParser	Summary description for OwlParser
OwlProperty	Represents an OWL resource of type rdf:Property
OwlResource	Represents a resource in the OWL Graph
OwlRestriction	Represents an OWL resource of type owl:Restriction
OwlXmlGenerator	Summary description for OwlXmlGenerator
OwlXmlParser	The primary OWL Parser

Table C.2: OwlDotNetApi Classes

C.3 OwlDotNetApi Interfaces

Interface	**Description**
IOwlAnnotationProperty	Represents a OWL Node of type owl:AnnotationProperty
IOwlClass	Represents a OWL Node of type owl:Class
IOwlCollection	Represents an OWL List
IOwlDatatype	Represents a OWL Node of type owl:Datatype
IOwlDatatypeProperty	Represents a OWL Node of type owl:DatatypeProperty
IOwlEdge	Defines a generalized mechanism for processing edges in the OWL Graph
IOwlEdgeCollection	Represents a collection of objects that implement the IOwlEdge interface. This collection maps edge IDs to objects that implement the IOwlEdgeList interface.
IOwlEdgeList	Represents a collection of IOwlEdge objects
IOwlGenerator	Represents an OWL Generator
IOwlGraph	Represents an OWL Graph comprising Nodes and Literals connected by Edges
IOwlIndividual	Represents a OWL Node of type owl:Individual
IOwlLiteral	Represents a Literal in an OWL Graph
IOwlNamespaceCollection	Represents a collection of Namespaces
IOwlNode	Represents a Node in the OWL Graph
IOwlNodeCollection	Represents a collection of IOwlNode objects
IOwlObjectProperty	Represents a OWL Node of type owl:ObjectProperty
IOwlOntology	Represents a OWL Node of type owl:Ontology
IOwlOntologyProperty	Represents a OWL Node of type owl:DatatypeProperty
IOwlParser	Represents an OWL Parser
IOwlProperty	Represents a OWL Node of type owl:Property
IOwlResource	Represents a OWL Node of type owl:Resource
IOwlRestriction	Represents a OWL Node of type owl:Restriction

IOwlVisitor	This interface defines the type of object that the nodes and edges will accept. The Node hierarchy classes call back a Visiting object's Visit() methods; In so doing they identify their type. Implementors of this interface can create algorithms that operate differently on different type of Nodes.
IOwlXmlGenerator	Represents an OWL Xml Generator
IOwlXmlParser	Represents an OWL Xml Parser

Table C.3: OwlDotNetApi Interfaces

Appendix

D Festo Ontology

D.1 The Festo INI-file (excerpt)

Listing D.1: The Festo INI-file

```
...
Class(1) = 1, 1, "Digital modules"
Class(2) = 2, 2, "Analogue modules"
...
[Module2]
Type = "4DO"
Text = "Output module"
Bitmap = "MODUL-A-CPX"
Ordercode = "A"
Code = 3
Class = 1
Channels(1) = "-", 0, 0, 4, 1
ParamBytes = 6
ParamDefaults = 06 00 00 00 00 00
Param(1) = 2,64,0,1,1
Param(2) = 3,64,0,2,1
Param(3) = 10,64,1,1,1
```

```
...
[Param2]
Text = "Monitor SCO"
Format = 32
Range = 1 0-0
PrmText = 1

[Param3]
Text = "Monitor Vout/Vval"
Format = 32
Range = 1 0-0
PrmText = 1
...
[Param10]
Text = "Behaviour after SCO"
Format = 32
Range = 0 0-0
PrmText = 14
...
[PrmText1]
Text(0) = "Inactive"
Text(1) = "Active"
...
[PrmText14]
Text(0) = "Leave switched off"
Text(1) = "Switch on again"
```

D.2 The Festo Device Ontology (excerpt)

Listing D.2: The Festo Device Ontology

```
<?xml version="1.0"?>
<rdf:RDF
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:owl="http://www.w3.org/2002/07/owl#"
     xmlns="http://www.festo.com/festOnt#"
     xmlns:p1="http://www.owl-ontologies.com/assert.owl#"
  xml:base="http://www.festo.com/festOnt">
  <owl:Ontology rdf:about=""/>

  <DigitalDevice rdf:ID="Dev_4DO">

    <ofDeviceFamily rdf:resource="#CPX"/>

    <hasDeviceName xml:lang="en">4DO</hasDeviceName>
    <hasDeviceName xml:lang="de">4DA</hasDeviceName>

    <hasDeviceDescription xml:lang="en">Output module</
        hasDeviceDescription>
        <hasDeviceDescription xml:lang="de">Ausgangsmodul</
            hasDeviceDescription>

        <hasDeviceBitmap rdf:datatype="http://www.w3.org/2001/XMLSchema#
            string">MODUL-A-CPX</hasDeviceBitmap>

        <hasDeviceCode rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
            >3</hasDeviceCode>

        <isOfDeviceClass rdf:datatype="http://www.w3.org/2001/XMLSchema#
            int">1</isOfDeviceClass>

        <hasParData rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >2,64,0,1,1</hasParData>
        <hasParData rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >3,64,0,2,1</hasParData>
        <hasParData rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >10,64,1,1,1</hasParData>

        <hasParameter>
                <DeviceParameter rdf:ID="Par_2">
                        <hasParName xml:lang="en">Monitor SCO</hasParName>
                        <hasParName xml:lang="de">Überwachung KZA</
                            hasParName>

                        <hasParFormat rdf:datatype="http://www.w3.org
                            /2001/XMLSchema#int">32</hasParFormat>

                        <hasDefaultValue rdf:datatype="http://www.w3.org
                            /2001/XMLSchema#int">1</hasDefaultValue>

                        <hasParText>
                                <ParText_1 rdf:ID="ParText_1a">
                                        <hasText xml:lang="en">Inactive</
                                            hasText>
                                        <hasText xml:lang="de">Inaktiv</
                                            hasText>

                                        <isFollowedByParText>
                                          <ParText_1 rdf:ID="ParText_1b">
                                                <hasText xml:lang="en">
                                                    Active</hasText>
                                                <hasText xml:lang="de">
                                                    Aktiv</hasText>
                                                <isFollowedByParText rdf:resource="#emptyParText"/>
                                          </ParText_1>
                                    </isFollowedByParText>
                              </ParText_1>
                        </hasParText>
                  </DeviceParameter>
      </hasParameter>

            <hasParameter>
                  <DeviceParameter rdf:ID="Par_3">
                        <hasParName xml:lang="en">Monitor Vout/Vval</hasParName>
                        <hasParName xml:lang="de">Überwachung Uaus/Uven</hasParName>

                        <hasParFormat rdf:datatype="http://www.w3.org/2001/XMLSchema#int">32</hasParFormat>

                        <hasDefaultValue rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</hasDefaultValue>

                        <hasParText rdf:resource="#ParText_1a"/>
                  </DeviceParameter>
            </hasParameter>

      <hasParameter>
                  <DeviceParameter rdf:ID="Par_10">
                        <hasParName xml:lang="en">Behaviour after SCO</hasParName>
                        <hasParName xml:lang="de">Verhalten nach KZA</hasParName>

                        <hasParFormat rdf:datatype="http://www.w3.org
                            /2001/XMLSchema#int">32</hasParFormat>

                        <hasDefaultValue rdf:datatype="http://www.w3.org
                            /2001/XMLSchema#int">0</hasDefaultValue>

                        <hasParText>
                                <ParText_14 rdf:ID="ParText_14a">
                                        <hasText xml:lang="en">Leave
                                            switched off</hasText>
                                        <hasText xml:lang="de">
                                            ausgeschaltet lassen</hasText>

                                        <isFollowedByParText>
                                          <ParText_14 rdf:ID="ParText_14b"
                                              >
                                                <hasText xml:lang="en">
                                                    Switch on again</
                                                    hasText>
                                                <hasText xml:lang="de">
                                                    wieder einschalten</
                                                    hasText>
                                                <isFollowedByParText
                                                    rdf:resource="#
                                                    emptyParText"/>
                                          </ParText_14>
                                        </isFollowedByParText>
                                </ParText_14>
                        </hasParText>
                </DeviceParameter>
    </hasParameter>

  </DigitalDevice>
</rdf:RDF>
```

Bibliography

[Con07] Gene Ontology Consortium. Biological process ontology guidelines, 2007. http://www.geneontology.org/GO.process.guidelines.shtml.

[Cor07] Teknowledge Corporation. Suggested upper merged ontology (SUMO), 2007. http://www.ontologyportal.org/.

[Cov] R. Cover. Updated DAML+OIL language specification supports W3C XML schema data types. http://xml.coverpages.org/ni2001-03-28-a.html.

[DRS$^+$] N. Drummond, A.L. Rector, R. Stevens, G. Moulton, M. Horridge, H. Wang, and J. Seidenberg. Putting OWL in order: Patterns for sequences in OWL. http://owl-workshop.man.ac.uk/acceptedLong/submission_12.pdf.

[FBL99] M. Fischetti and T. Berners-Lee. *Weaving the Web*. Harper San Francisco; Auflage: 1st (Oktober 1999), 1999.

[Fen07] D. Fensel. *Enabling Semantic Web Services*. Springer, 2007.

[GvH04] A. Grigoris and F. van Harmelen. *A Semantic Web Primer*. The MIT Press, 2004. http://www4.wiwiss.fu-berlin.de/bookmashup/books/1591405033.

[KSD01] A. Kiryakov, K. Simov, and M. Dimitrov. OntoMap: ontologies for lexical semantics, 2001. http://www.ontotext.com/publications/ranlp01.pdf.

[MBF+05] G. Miller, R. Beckwith, C. Fellbaum, D. Gross, and K. Miller. Introduction to wordnet: An on-line lexical database, 2005. ftp://ftp.cogsci.princeton.edu/pub/wordnet/5papers.ps.

[MBH+04] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. Mcdermott, S. Mcilraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara. OWL-S: Semantic markup for web services, 2004. http://www.w3.org/Submission/OWL-S/.

[MLD] J. L. Martinez Lastra and I. M. Delamer. Semantic web services in factory automation: Fundamental insights and research roadmap. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?tp=&arnumber=1593597&isnumber=33541.

[MNV02] M. Missikoff, R. Navigli, and P. Velardi. The usable ontology: An environment for building and assessing a domain ontology, 2002.

[NM01] N. Noy and D. McGuinness. Ontology development 101: A guide to creating your first ontology. Technical report, Stanford University School of Medicine, 2001. http://protege.stanford.edu/publications/ontology_development/ontology101.pdf.

[NP] I. Niles and A. Pease. Towards a standard upper ontology. http://home.earthlink.net/~adampease/professional/FOIS.pdf.

[oCSUoM] Department of Computer Science University of Maryland. SHOE - simple html ontology extension. http://www.cs.umd.edu/projects/plus/SHOE/.

[Ogb] U. Ogbuji. Use rdf:about and rdf:ID effectively in RDF/XML. http://www.ibm.com/developerworks/xml/library/x-tiprdfai.html.

[Pow03] S. Powers. *Practical RDF*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2003.

[RFS07] C. Ringelstein, T. Franz, and S. Staab. *The Process of Semantic Annotation of Web Services*. Idea Publishing Group, USA, 2007. http://www.uni-koblenz.de/~staab/Research/Publications/2006/.

[UG96] M. Uschold and M. Gruninger. Ontologies: principles, methods, and applications, 1996. http://www.aifb.uni-karlsruhe.de/Lehrangebot/Sommer2001/SemanticWeb/papers/Uschold-96.pdf.

[W3Ca] W3C. OWL web ontology language guide. http://www.w3.org/TR/owl-guide/.

[W3Cb] W3C. OWL web ontology language use cases and requirements. http://www.w3.org/TR/webont-req/#onto-def.

[W3Cc] W3C. Primer: Getting into RDF and semantic web using N3. http://www.w3.org/2000/10/swap/Primer.html.

[W3Cd] W3C. RDF primer. http://www.w3.org/TR/REC-rdf-syntax/.

[W3Ce] W3C. RDF validator. http://www.w3.org/RDF/Validator/.

[W3Cf] W3C. RDF vocabulary description language 1.0: Rdf schema. http://www.w3.org/TR/rdf-schema/.

[W3Cg] W3C. Resource description framework (RDF): Concepts and abstract syntax. http://www.w3.org/TR/rdf-concepts/.

[W3Ch] W3C. Web services glossary. http://www.w3.org/TR/ws-gloss/.

[W3Ci] W3C. XML schema part 2: Datatypes. http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/.