Diplomarbeit

# Knowledge-Management with OWL and F-Logic: A Combination of Description Logic Reasoning with F-Logic Rules

6-Monats-Arbeit im Rahmen der Prüfung
für Diplom-Wirtschaftsinformatiker
an der Universität Göttingen

vorgelegt am 30. Mai 2007

von Heiko Kattenstroth
aus Gütersloh

Betreut durch Professor Dr. Wolfgang May
Institut für Informatik
Arbeitsgruppe Datenbanken
Georg-August-Universität Göttingen

## Abstract

This thesis proposes an approach to combine the abilities of Description Logic reasoning with the expressiveness of F-Logic rules.

*Description Logics (DL)* play a key role in ontology modeling for the Semantic Web, most notably as the underlying logic of OWL DL. However, their expressiveness is limited, in particular in conjunction with efficient reasoning. Furthermore, the logic programming paradigms such as F-Logic have been researched extensively in the past and powerful tools and systems are available. The approach developed in this thesis utilizes a DL reasoner (Pellet) and an F-Logic system (Florid) for an alternating reasoning process with an OWL DL ontology backed by the Jena Semantic Web Framework. The DL reasoner and a set of F-Logic rules are applied iteratively to the ontology in order to combine the advantages of both worlds. For this process, the ontology has to be exchanged between both systems. Hence, the translation of a subset of an OWL DL ontology into F-Logic and reverse is provided. Furthermore, the system is prototypically implemented as part of this thesis. Moreover, the benefits of the combination are shown by some examples.

## Acknowledgements

First of all, I would like to thank Professor Dr. Wolfgang May for offering me this interesting topic as a diploma thesis and for the excellent personal and scientific support and supervision. Likewise, I appreciate Franz Schenk for his helpful technological and scientific support.

Furthermore, I am grateful to Benjamin Roll and Friedemann Lindenthal for their constructive comments, critical revision, and motivation during the writing of this thesis. Special thanks go to my parents and particularly to my girlfriend Cornelia Krüger for their support and their patience at any time.

# Contents

# List of Figures

# List of Tables

# List of Examples

# List of Abbreviations

| | |
|---|---|
| **API** | Application Programming Interface |
| **DAML** | DARPA Agent Markup Language |
| **DIG** | DL Implementation Group |
| **DL** | Description Logics |
| **DLP** | Description Logic Programs |
| **DTD** | Document Type Definition |
| **FOL** | First-Order Logic |
| **Florid** | F-Logic Reasoning in Databases |
| **F-Logic** | Frame Logic |
| **HTML** | Hypertext Markup Language |
| **JSP** | Java Server Pages |
| **LP** | Logic Programs |
| **MVC** | Model-View-Controlloer |
| **OID** | object identifier |
| **OWL** | Web Ontology Language |
| **RDF** | Resource Description Framework |
| **RDFS** | RDF Vocabulary Description Language |
| **RuleML** | Rule Markup Language |
| **SGML** | Standard Generalized Markup Language |
| **SPARQL** | SPARQL Protocol and RDF Query Language |
| **SWRL** | Semantic Web Rules Language |
| **URI** | Uniform Resource Identifier |
| **W3C** | World Wide Web Consortium |
| **WWW** | World Wide Web |
| **XML** | Extensible Markup Language |
| **XMLNS** | XML Namespace |
| **XSD** | XML Schema Definition |
| **XQuery** | XML Query Language |

# 1 Introduction

The Semantic Web is envisioned as an extension of the *World Wide Web (WWW)* in which knowledge is described in such a way that computer can process and understand it. This way, computer would be able to draw conclusions from the given knowledge using processes similar to human reasoning and inference. At present, some of the enabling technologies of the Semantic Web have reached a certain degree of maturity with W3C recommendations such as RDF and OWL. In particular, the *Web Ontology Language (OWL)* adds considerable expressive power by providing means to create ontologies which define the vocabulary needed for the semantic mark-up. Besides this benefits, OWL has expressive limitations. Among others, it does not support composition constructors which would allow, for example, to define an *uncle* relationship via the composition of the *brother* and the *parent* relationship. Furthermore, only a subset of OWL is used in order to retain decidability of key inference problems. In fact, the need for an extension of OWL by rules has been known since the beginning. Hence, the current research focuses on rules an its integration with the ontology languages.

Therefore, an approach to extend the abilities of OWL DL, in terms of Description Logic reasoning, with the expressiveness of F-Logic rules is proposed in this thesis. For such a combination, an translation of the facts between both systems is needed. This thesis provides such a translation. Hence, the translation of a subset of an OWL DL ontology into F-Logic and reverse is provided. Moreover, the approach is implemented as part of this thesis.

This thesis is structured as follows. In the next chapter, the basic concepts on which this thesis is based are presented. Based on these basic principles, a general outline of the Jena Semantic Web framework and a review of the provided functionalities are given in Chapter 3. Afterwards, Chapter 4 introduces the F-Logic implementation Florid. How this F-Logic system can be used for a combination of F-Logic rules and Description Logic reasoning is explained in Chapter 5. The prototypical implementation of this approach is described in Chapter 6 and an evaluation is given in Chapter 7. Subsequently, Chapter 8 presents some related approaches and denotes the main topics for further extensions. Finally, the thesis is concluded in Chapter 9.

# 2 Basics

For the understanding of this thesis a common conception of the basic principles is necessary. As a consequence, this chapter gives a short overview of the essential concepts and used technologies. For more details the study of the given references is recommended.

## 2.1 Semantic Web

At present, the World Wide Web contains an enormous and growing amount of information distributed over nodes which are forming a heterogeneous network. Most of the content of this nodes is marked up in the *Hypertext Markup Language (HTML)* and is almost solely designed for humans to read and to understand. Therefore, computers have no reliable way to manipulate the content meaningfully and to process the semantics (see [BLHL01]). More precisely, humans are capable of using the World Wide Web to carry out tasks, such as finding and making a reservation for the latest book of Tim Berners-Lee in the nearest library. By contrast, computers can only complete small and segregated parts like searching for the latest book or for the nearest library (cf. [BLH01]).

This shortcoming in combination with the exponential growth of the World Wide Web leads to the need for a *Semantic Web*, which was described by Tim Berners-Lee 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." [BLF99]

More precisely Tim Berners-Lee defines the term Semantic Web as follows:

> "The Semantic Web is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation." [BLHL01]

The Semantic Web is envisioned to extend principles of the Web from documents to data. This extension should be usable as an universal medium for data, information and knowledge exchange and be processable by both automatic tools and humans (see [BLHL01]).

According to Tim Berners-Lee the Semantic Web depends on a layered, functional architecture, also known as the "Layer Cake" (cf. Figure 2.1).

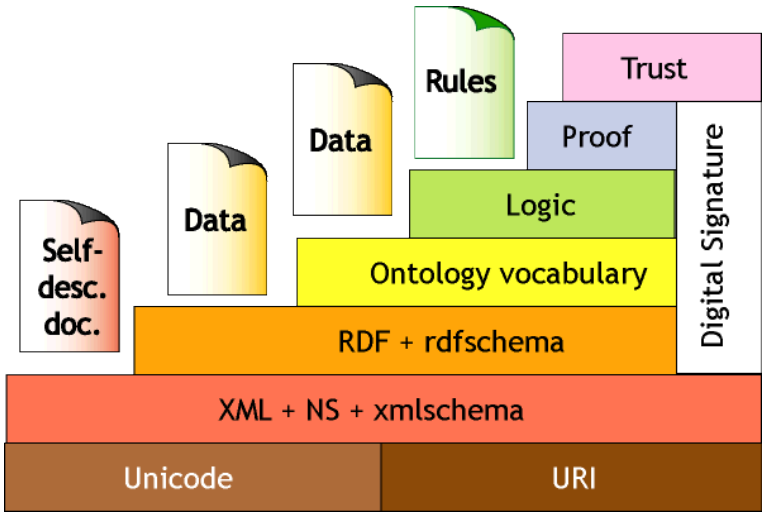


**Figure 2.1:** Semantic Web Architecture ("Layer Cake") [BL00]

The architecture consists of a range of either established or quite recent technologies. The core layers that are of significant importance for this thesis are subject of the subsequent sections.

For more details see [BL98b, SH01]. The *World Wide Web Consortium (W3C)* [W3Ca] standardizes and specifies new technologies in the field of the World Wide Web and related areas. For a general survey of current activities of the W3C around the Semantic Web see [W3C01b].

## 2.2 Ontology

The term ontology originally originates from philosophy and deals with theories of being and existence. It was introduced into computer science in the context of Artificial Intelligence research and gained popularity within the field of the Semantic Web (cf. [GG95] and [Gua98]). A widely used definition was given by Thomas R. Gruber:

> "An ontology is a formal specification of a shared conceptualization of a domain of interest." [Gru93]

In this context, a conceptualization is an abstract, simplified model of a part of the real world (domain) that shall be represented (cf. [GG95]). Furthermore the ontology should be machine-processable ("formal specification") and should reflect consensual knowledge ("shared").

Ontologies are used as a form of knowledge representation and generally describe:

- *Classes (concepts)*: abstract groups, sets, or collections of objects.
- *Individuals (instances)*: the basic or "ground level" objects.
- *Attributes*: objects in the ontology can be described by assigning attributes, which have at least a name and a value, to them.
- *Relations*: describe the relationships between objects in the ontology and are typically stated as a property whose value is another object.

In order to make an ontology computer-readable and the meaning computer-understandable, more technologies are needed which will be introduced briefly in the next sections.

## 2.3 URI

The *Uniform Resource Identifiers (URIs)* are an addressing standard of the W3C that is used for identifying a resource unambiguously. Therefore, a character string is used that has the following structure: `<scheme name> : <scheme specific part>` (cf. [BLFM98]).

The scope of what might be a resource is not limited and resources are not necessarily accessible via the Internet: e.g. books in a library. Even abstract concepts, such as the types of a relationship, can be resources (see [BLFM98]).

The well-known *Uniform Resource Locators (URLs)* are a subclass of URIs, which can be used to *locate* a resource, e.g. a website. For more details see [W3C93] and [W3C07b].

## 2.4 XML and Related Concepts

**XML.** The *Extensible Markup Language (XML)* is an open, flexible and simple text-based standard for the representation of (semistructured) data. It is a simplified subset of the *Standard Generalized Markup Language (SGML)*, and is designed to be human-readable and understandable (see [W3C06a]).

Basically, XML defines a set of syntactic rules which enables a tree-based representation of the document. XML-documents that obey these rules are called *well-formed.* The created

trees consist of nested sets of named elements which may have several attribute-value pairs. Each element starts and ends with a tag, which must have the same name.

The main purpose is to facilitate the sharing and exchange of data between different information systems, particularly systems connected via the Internet. Since XML can define the syntax of arbitrary markups, which means to create a markup language, XML is called a *meta language*.

**XML Namespaces.** The same name for elements and attributes can be declared in different contexts which may lead to ambiguities. The *XML Namespace (XMLNS)* specification has been developed to avoid conflicts between names from different applications (see [BHL99]). For that, each name can be prefixed with a namespace, which must be declared beforehand in one of the superelements, e.g. the root element. In a namespace declaration a URI reference is assigned to a shortcut (the namespace prefix). Example 2.3 makes use of namespaces: for instance, the prefix "`rdf`" is assigned to the namespace URI "`http://www.w3.org/1999/02/22-rdf-syntax-ns`". This shortcut is expanded with the namespace URI when the XML document is parsed.

**XML Schema Languages.** As explained above, XML can represent documents as trees. But many applications need to restrict the structure in order to make use of them. For that, the structure of an XML document has to be specified, e.g. the set of allowed tags and the way they can be nested. This can be done by a *Document Type Definition (DTD)* or in a more powerful way by a *XML Schema Definition (XSD)*.[1]

A brief example of a well-formed XML document is given in Example 2.1, while the corresponding tree structure is depicted in Figure 2.2:

```
<library>
  <book isbn="012345" signature="FMAG 2003 XYZ">
    <title>
      Weaving the Web
    </title>
    <author id="tbl">
      <name>Tim Berners-Lee</name>
    </author>
    <author id="mf">
      <name>Mark Fischetti</name>
    </author>
```

[1] XML Schema is more powerful because it supports the derivation of element types, allows nested definitions, and provides atomic data types. However, XML Schema is also more complex than DTDs (see for example [LC00]).

```
    <year>1999</year>
    ...
</library>
```

**Example 2.1:** XML Example

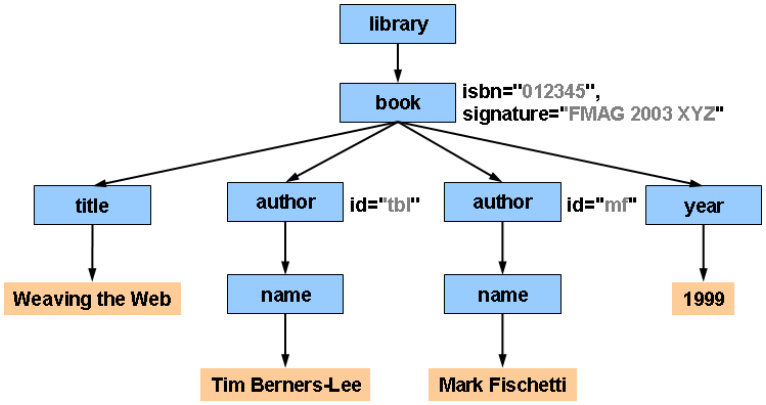


**Figure 2.2:** XML Tree

**XQuery.** The *XML Query Language (XQuery)* is a query language designed to query across all kinds of XML data, whether physically stored in XML or viewed as XML via middleware. XPath expressions are used to address parts of an XML document. Furthermore, XQuery uses SQL-like "*FLWOR*" expressions which are composed from five clauses: `FOR`, `LET`, `WHERE`, `ORDER BY`, `RETURN`. More details can be found in [W3C01c].

## 2.5 RDF and RDF Schema

**RDF.** The *Resource Description Framework (RDF)* has been developed by the W3C and is a language for representing information about resources in the *World Wide Web (WWW)*, in particular for modeling metadata about Web resources. The main purpose is for situations in which information needs to be processed and exchanged between applications without loss of meaning [W3C00].

RDF bases on the idea that *resources* can be described by making *statements* about them. Resources are any type of data which can be described by RDF expressions and identified by a URI. A statement is also called a *triple* and has the following form: *subject-predicate-object*. The three parts are:

- **subject**: The subject denotes the resource that the statement specifies.

- **predicate**: The predicate identifies the property or characteristic of the subject and expresses the relationship between the subject and the object.
- **object**: The object identifies the value of the property. This can either be a literal, e.g. a string, or another resource.

In general, RDF uses URIs (see Section 2.3) to identify subjects, predicates, and objects in statements.[2] An unnamed resource, i.e. a resource that is not directly identifiable (for example by a URI) nor a literal, is called an *anonymous resource*[3]. Since nodes with identical URIs are considered as being identical, it is feasible to merge data from different RDF sources.

An RDF model is a collection of statements which encode a labeled directed graph[4]. A statement thereby denotes an edge with the subject as the initial node, the object as the end node and the predicate as the label (cf. [KC04]). An exemplary graph is illustrated in Figure 2.3.

Furthermore, the RDF model is serialization-independent and can be represented in many different ways, though the proposed standard serialization is an XML markup called *RDF/XML* [W3C04d]. An introductory example for RDF and in particular for the different representation forms is given below.[5]

Statements in common English:

*Tim Berners-Lee is the author of the book "Weaving the Web".*
*It was published in 1999.*

As N-Triples[6] with arbitrary URIs:

```
<http://foo.org/w-t-w> <http://foo.org/author> "Tim Berners-Lee" .
<http://foo.org/w-t-w> <http://foo.org/title>  "Weaving the Web" .
<http://foo.org/w-t-w> <http://foo.org/date>   "1999" .
```

**Example 2.2:** N-Triples

[2] To be more precisely, RDF uses *URI references*. A URI reference is a URI with an optional suffixing fragment identifier which is separated by a **#** (see [KC04]).
[3] The terms *blank node* and *bNode* are used synonymously.
[4] In RDF, the terms *model* and *graph* are used synonymously.
[5] Note that the co-author Mark Fischetti is omitted purposely.
[6] *N-Triples* is a line-based, plain text format for RDF, where one line contains one statement. Among others, URIs are enclosed in angle brackets and literals in quotation marks. See [W3C01a] for details.

In RDF/XML:

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:dc="http://purl.org/dc/elements/1.1/">
  <rdf:Description rdf:about="http://foo.org/w-t-w">
    <dc:creator>Tim Berners-Lee</dc:creator>
    <dc:title>Weaving the Web</dc:title>
    <dc:year>1999</dc:year>
  </rdf:Description>
</rdf:RDF>
```

**Example 2.3:** RDF/XML

The corresponding graph[7]:

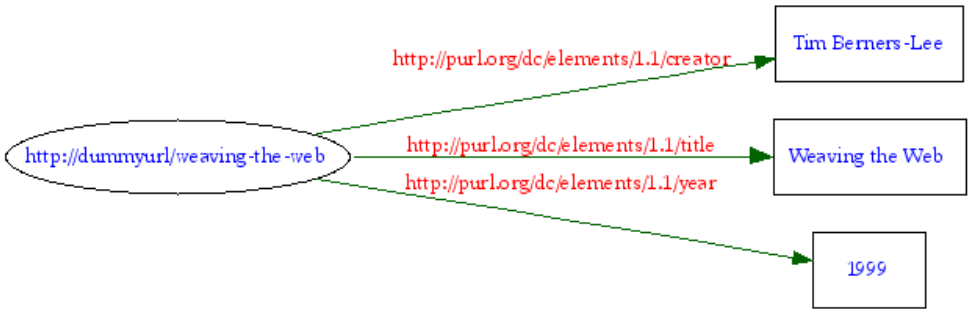


**Figure 2.3:** RDF Graph

For further features of RDF, like *reification* (making statements about statements), please refer to [Hay03, W3C04c] or [W3C00].

**RDF Schema.** As mentioned above, RDF provides the means to express statements about resources. However, RDF defines the syntax but lacks the ability to describe the used vocabularies[8]. Though a description is essential for the interpretation and common understanding of such statements. For this reason, the *RDF Vocabulary Description Language (RDFS)*, also known as *RDF Schema*, was developed. RDFS is a semantic extension of RDF and is published as a W3C recommendation [W3C04c]. It is written in RDF and describes the structure of an RDF instance, i.e. the structure of the underlying RDF graph and not the structure of the RDF/XML file (or any other representation).

[7] This graphical visualization was created with the *W3C RDF Validation Service* [W3Cc].
[8] A common example is the Dublin Core Metadata Initiative, which is widely used to describe digital material and which can be found as a RDFS Schema here: `http://purl.org/dc/elements/1.1/`.

The basic idea of RDFS is to combine individual RDF resources into classes, set these in relation with other classes, and finally to describe individual resources as manifestations (instances) of their original classes. Here, terms from RDFS are marked with the prefix `rdfs:` and terms from RDF with the prefix `rdf:`.[9] Classes (`rdfs:Class`) and properties (`rdf:Property`) can be organized in a hierarchical way by using the `rdfs:subClassOf` property or the `rdfs:subPropertyOf` property, respectively.

An other important part of RDFS is the declaration of global domain (`rdfs:domain`) and range (`rdfs:range`) assertions for properties. The range restriction indicates that the values of the property are instances of a designated class while the domain restriction specifies that the property applies to a designated class.

RDFS can be used for lightweight ontologies, although the W3C prefers the term `vocabularies`. Moreover, RDFS allows to reason about the given data, for example, type inheritance through `rdfs:subClassOf` or type inference through `rdfs:range` and `rdfs:domain` assertions. More details can be found in [W3C04c].

As stated in this section, RDFS provides some basic means for describing vocabularies. But the possibilities are still restricted. For example, neither property characteristics (inverse, transitive, etc.) nor the disjointness of classes can be expressed. Hence a more powerful language is needed, which is described in the following section (cf. [W3C03]).

## 2.6 OWL and Description Logic

This section gives first an introduction into the ontology language OWL, followed by the basics of the underlying Description Logic and finally an insight into Description Logic reasoning.

### 2.6.1 OWL

The Web Ontology Language was introduced 2003 in [DCv+02] as a layer on top of RDFS (see Figure 2.1). In 2004 it was published as a W3C Recommendation [BvH+]. The basic fact-stating ability of RDF and the class- and property-structuring capabilities of RDFS (Section 2.5) are extended in several ways. OWL adds more vocabulary for describing properties and classes: relations between classes (e.g. disjointness), cardinality (e.g. “exactly one”), equality, characteristics of properties (e.g. symmetry), and enumerated classes (e.g. “one of”) (see [W3C04b]). The design of OWL was influenced by several pre-existing languages, in particular by DAML+OIL [CvH+].

[9] For a description of namespaces see Section 2.4.

The W3C chose *Description Logics (DL)* as a logical base for the Web Ontology Language which is introduced in Section 2.6.2. Hence, the different variants of OWL can be regarded as particular Description Logics (cf. [HPS03]).

As full ontology-based applications are hard to implement and to run, three OWL layers were defined to reflect compromises between expressiveness and complexity. Thereby, OWL comes in the following three layers with decreasing level of complexity and expressiveness:

- **OWL Full**: All OWL and RDFS elements are allowed in arbitrary combinations, which leads to maximum expressiveness, but no computational guarantees. In particular, the fact that a class can also be an individual is problematic.
- **OWL DL**: This is the maximal subset of OWL Full for which the reasoning is complete (all entailments are guaranteed to be computed) and decidable (all computations will finish in finite time). OWL DL makes several restrictions on OWL Full, e.g., the sets of identifiers of classes, properties, and individuals must be disjoint.
- **OWL Lite**: This layer is designed to provide a minimal useful subset of OWL DL, that can be easily implemented. Primarily intended for a classification hierarchy and simple constraint features. OWL Lite is created by imposing some restrictions on OWL DL, e.g. the value of a cardinality restriction is limited to 0 or 1.

The relation between these three layers is depicted in Figure 2.4 and can be written as:

$$OWL\ Lite \subset OWL\ DL \subset OWL\ Full$$

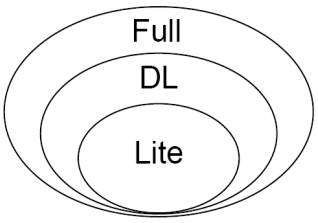


**Figure 2.4:** OWL Hierarchy

In general, OWL makes an *open world assumption*: A statement cannot be assumed false on the basis of a failure to prove it. Furthermore, OWL does not follow the *unique name assumption*: For two resources with different names (URIs), it may still be derived by inference that they must be the same. More details can be found in [HPSvH03], and [W3C04a]. An extension of OWL DL, called *OWL 1.1*, is currently under development [W3C07a]. Please refer to Example A.1 in the appendix for an exemplary OWL DL ontology. This example is explained in more detail in Chapter 5 and it is visualized in Figure 5.1.

Since OWL DL and OWL Lite are based on Description Logic, a short introduction into this form of logic is given next.

### 2.6.2 Description Logic

Description Logics, previously called *terminological logics*, are logical formalisms used in knowledge-based systems to represent and reason about terminological knowledge of a problem domain. They were developed as an extension to object-centric formalisms like semantic networks and frame systems. More precisely, Description Logics add well-defined formal and declarative semantics (cf. [BCM+03]).

Extensive research in this field of knowledge representation resulted in a wide range of Description Logics which can be distinguished from each other by the available constructors and axioms. In particular, the research is concerned with the expressiveness and computational properties of the various DLs. In general, Description Logics are constructed in such a way that inferencing is decidable. Hence, Description Logics can be considered as a decidable subset of *First-Order Logic (FOL)* (see [Bor96]).[10] However, the expressiveness of DL is limited seriously, for example by the absence of variables. However, this limited expressiveness ensures decidability, improves tractability, and allows efficient reasoning.

DLs are based on the description of classes[11] and properties[12]. Classes are described implicitly by the properties that objects must satisfy in order to belong to the class, while a property relates pairs of objects with each other. More precisely, classes correspond to unary predicates of FOL, e.g. $C(a)$ states that $a$ is an instance of $C$. Furthermore, properties correspond to binary predicates of FOL (cf. [Bor96]). For example, $P(a,b)$ denotes the property $P$. Hence, ontology languages like OWL DL may be called *predicate-based* (cf. [BH06]). A more detailed description of the correspondence between DL and FOL can be found in [BH06] and [Bor96]. The set of available operators to build complex class and property constructs characterizes a particular Description Logic (see Table 2.1). Normally, a DL is based on the logic $\mathcal{AL}$ [SSS91] or on $\mathcal{ALC}$. $\mathcal{AL}$ is a basic language and allows atomic negation, concept intersection, universal restrictions, and limited existential quantification whereas $\mathcal{ALC}$ allows additionally more complex concept negation (cf. [SSS91]).

OWL DL is based on the expressive Description Logic $\mathcal{SHOIN}(\mathcal{D})$. As mentioned above, OWL Lite is restricted in several ways, which result in the less expressive Description Logic

[10] Since there is no algorithm that for any formula in FOL returns its validity, First-Order Logic is not decidable. Most DLs are less complex than the decidable Logic $\mathcal{L}^2$, i.e FOL with only two (reusable) variable symbols (cf. [Bor96]).

[11] Here, the term *class* is used instead of the term *concept* as it is commonly used in the Semantic Web context.

[12] Like before, the term *property* is used instead of *role* as it is commonly used in this context.

| Name | Symbol |
|---|---|
| $\mathcal{AL}$ | Attribute Language |
| $\mathcal{ALC}$ | Attribute Language with Complement |
| $\mathcal{S}$ | ALC with transitive properties |
| $\mathcal{H}$ | Property hierarchy (subproperties) |
| $\mathcal{O}$ | Nominals |
| $\mathcal{I}$ | Inverse properties |
| $\mathcal{N}$ | Cardinality restrictions ("number restrictions") |
| $\mathcal{Q}$ | Qualified property restrictions |
| $(\mathcal{D})$ | Data types, e.g. integers |
| $\mathcal{F}$ | Functional properties |

**Table 2.1:** Description Logic Expressiveness

$\mathcal{SHIF}(\mathcal{D})$. The expressiveness of OWL DL leads to a NExpTime complexity for key inference problems whereas key inference in OWL Lite can be computed in exponential time (ExpTime) (cf. [HPS03]).

In general, a DL knowledge base consists of the following two components:

1. A *terminological box* or *TBox*, which contains definitions and assertions about classes and properties. More precisely, it formalizes *subsumption* and *equivalence* relations. Subsumption is typically written as $C \sqsubseteq D$ which means that $D$ subsumes $C$, i.e. the class (property) $D$ is considered as being more general than the class (property) C. Whereas equivalence is denoted as $C \equiv D$ and is often used to define left-hand side classes. For example, $Woman \equiv Person \sqcap Female$ defines a woman as a female person (see [BCM+03]).
2. An *assertional box* or *ABox*, which contains the facts about the instances (individuals) in terms of basic classes, properties and intensional classes. For example, $Person(ADAM)$ states that the instance $ADAM$ is a person.

### 2.6.3 Reasoning

An inference engine, also known as a reasoner, allows to derive implicit knowledge from what has been explicitly stated. DL reasoning can be split in TBox reasoning and ABox reasoning. An inference service which considers only the TBox is the calculation of the subsumption hierarchy. Whereas instance checking, i.e. determining, whether an individual is an instance of a certain class, is part of the ABox reasoning.[13]

[13] Since instance checking depends on the results from TBox reasoning, ABox reasoning is usually activated after the TBox reasoning.

**Pellet.** Pellet is an open-source OWL DL reasoner written in Java which was introduced in [PS04]. The full expressiveness of OWL DL and all the features proposed in OWL 1.1, with the exception of n-ary datatypes, are supported by Pellet (cf. [Mar]). Thus, the expressiveness of Description Logics that are supported by Pellet is $\mathcal{SROIQ}(\mathcal{D})$ (cf. [Mar]). This DL extends the well-known DL $\mathcal{SHOIN}(\mathcal{D})$ which underlies OWL DL. For more details see [HKS06].

Up to now, the basic concepts in the context of the Semantic Web that are relevant in this thesis have been introduced. The next chapter presents a framework which makes use of these concepts.

# 3 The Jena Semantic Web Framework

Jena [jen] is a leading open-source framework for developing Semantic Web applications with Java based on W3C recommendations for RDF and OWL (see Sections 2.5 and 2.6). An essential part of Jena was developed in the Hewlett Packard Labs Semantic Web Programme [hps]. The Jena framework offers a convenient way for working with ontologies and in particular for integrating ontologies into applications.

The current version of Jena is 2.5.2, which is used in this thesis[14]. A brief introduction of the Jena components that are relevant for this thesis is given in the following sections.

## 3.1 Architecture Overview

The Jena architecture consists of the following layers: the *Graph* layer, the *EnhGraph* layer and the *Model* layer(see Figure 3.1). Though, only the Model layer is used by the application programmers, while the Graph layer and the EnhGraph layer are solely used within the Jena framework and hidden from the user. These three layers are explained consecutively in the following paragraphs.

**Graph Layer.** The core of the Jena architecture is the RDF graph (see Section 2.5 or [KC04]), which is represented via the Graph layer. This layer defines an interface to create and manipulate these graphs. Generally, triples are used as the universal data structure. The Graph layer offers several ways to store these triples, either in memory or in persistent storage backed by a database (see Section 3.4).

**Model Layer.** The Model Layer is the primary abstraction layer of the RDF graph used by the application programmer. It offers two *Application Programming Interfaces (APIs)* for operating on the graph itself and on the nodes within the graph. The first one is the *Model* API, which allows access to all parts of the underlying RDF graph. The second one is the *Ontology* API, which can be used to deal with ontology languages, like RDFS or OWL. Both APIs are explained in more detail in Section 3.2.

[14] In some publications [CDD+04] the term *Jena2* is used to distinguish from older versions (1.*). However, here the term *Jena* is used continuously as an synonym for the prevailing version.

Additionally, the Model layer offers means for the input and output of RDF-models, respectively of RDF-graphs. For instance, Jena supports RDF/XML, N-Triple, and N3[15]. Furthermore, an RDF/XML-Parser[16] is used to validate the syntax of models in RDF/XML format.

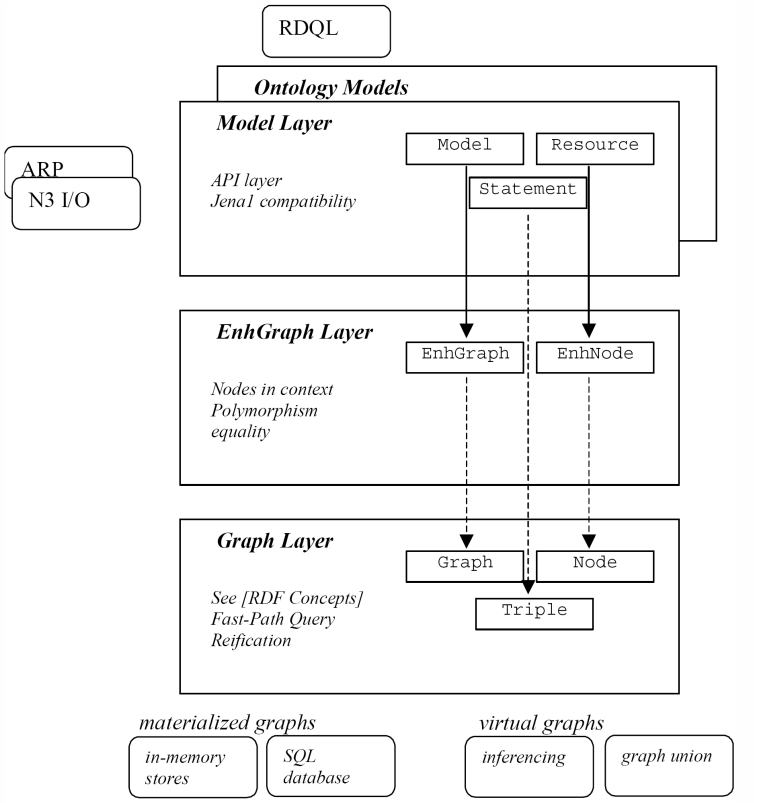


**Figure 3.1:** Jena2 Archicture [CDD+04]

**EnhGraph Layer.** The EnhGraph[17] layer is an intermediate layer between the Model layer and the Graph layer. According to the different used languages, e.g. OWL or RDFS, this layer provides multiple views of graphs and of nodes within a graph which can be used simultaneously. For example, the functionality offered by the EnhGraph layer is used to implement the Model API and the Ontology API (see Section 3.2).

[15] N3 is a compact and readable alternative to RDF/XML syntax which is often used for introductions into key principles of the Semantic Web [W3Cb].
[16] Jena uses the open-source Xerces Java parser [Prob].
[17] Enhanced Graph.

## 3.2 APIs

Jena provides several APIs, which are introduced briefly in the following sections. The heart of Jena is the *Model API*, which supports the creation, manipulation, and querying of RDF graphs (see [McB02]). Among others, it is accomplished by the *Ontology API*, the *Inference API* and the *Query API*.

### 3.2.1 Model API

As pointed out in Section 2.5, RDF uses statements consisting of a subject, a predicate and an object, to describe arbitrary resources. Altogether, these statements form the RDF graph with subjects and objects as nodes and predicates as arcs. In Jena, a graph is called a *model* and is represented by the `Model` interface. This Model interface allows to create, to access and to manipulates the elements of a model which are represented by the corresponding interfaces, e.g., `Statement`, `Property` or `Resource`. The precise use of the Model API is explained by the means of a short example, which is based on Example 2.3.[18]

```
String bookTitle = "Weaving the Web";
String bookAuthor = "Tim Berners-Lee";
String bookUri = "http://someurl.org/tbl";
// create an empty Model
Model model = ModelFactory.createDefaultModel();
// create a resource
Resource wtwBook = model.createResource(bookUri);
// add some properties
wtwBook.addProperty(DC.title, bookTitle);
wtwBook.addProperty(DC.creator, bookAuthor);
...
//print some properties of weaving the web
System.out.println(wtwBook.getProperty(DC.title).getString());
System.out.println(wtwBook.getProperty(DC.creator).getString());
```

**Example 3.1:** Example

Each call of `addProperty` in the example above added another statement to the Model. A statement is represented by the `Statement` interface, which provides access to the subject, predicate and object. Jena offers some convenient `list`-methods to obtain subjects, predicates, objects, and statements matching certain conditions. However, these methods support only simple queries and the more powerful query facilities of *SPARQL Protocol and RDF Query Language (SPARQL)* are described in section 3.3.

[18] Corresponding to Example 2.3, the Dublin Core metadata is used. It is represented in Jena by the `DC` class.

Furthermore, the Model API offers methods for reading and writing RDF as RDF/XML (or any other supported format). These methods can be used to save an RDF model to a file and later read it back in again. Additionally, the common set operations of *union*, *intersection* and *difference*, known from mathematical set theory, can be used to manipulate models as a whole.

### 3.2.2 Ontology API

Since Jena is based on RDF, only ontology languages built on top of RDF are supported. More precisely, RDFS, the three types of OWL, and DAML+OIL can be used with Jena.

Due to the fact that the names of the classes and interfaces are independent from the underlying ontology language, the Jena Ontology API is called *language-neutral*. Each ontology language has a *profile* (`OntModelSpec`), which contains all permitted constructs and the URIs of corresponding classes and properties.

The profile is bound to an ontology model (`OntModel`), which extends the Model from the Model API by adding support for the ontology-specific classes (in a class hierarchy), properties (in a property hierarchy) and individuals. The most important additional interfaces are:

- `OntResource`: extends the `Resource` interface and is a common super-class in the Ontology API. OntResource defines some methods, for example, to find out how many values a resource has for a given property (`getCardinality(Property prop)`), or to remove the resource from the ontology model (`remove()`). Furthermore, OntResource provides methods for listing, getting and setting the RDF types of a resource.
- `OntClass`: represents a class in an ontology and offers methods, for instance, to list subclasses, super-classes, equivalent classes, disjoint classes and all instances of this class.
- `OntProperty`: is an extension of the `Property` interface and acts as a super-class for the ontology properties. OntPropery offers some methods, for example, to access the domain (`getDomain()`) or the range (`getRange()`) of a property.
- `Individual`: represents an individual. Typically used to reflect individuals that are instances of user-defined classes.[19]

These are the basic interfaces, a more complete list can be found in [jen].

[19] In OWL Full and DAML+OIL any value can be an indivdual.

### 3.2.3 Inference API / Reasoner API

The Inference API was developed to be able to use a range of inference engines or reasoners (see Section 3.5 for details on these engines and reasoning in general). Each available reasoner is represented by a factory object which is an instance of the `ReasonerFactory`. These factories can be used to create the corresponding reasoner instances. Such an instance can then be bound to an RDF graph, which results in an `InfGraph`[20]. An InfGraph is a specialization of the standard `Graph` interface that contains all triples from the base graph and additionally, the inferred "virtual" triples. Thus, the Model and the Ontology APIs can use this InfGraph like a normal graph. This is depicted in Figure 3.2.

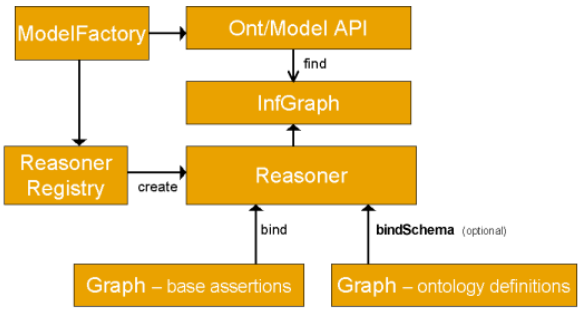


**Figure 3.2:** Inference Subsystem

## 3.3 Query language SPARQL

SPARQL is a recursive acronym standing for SPARQL Protocol and RDF Query Language. As the name implies, SPARQL is a protocol [W3C06b] and a query language [W3C04e] for RDF data, but mostly the acronym SPARQL is used for the query language. It builds on previous RDF query languages such as rdfDB [Guh] and RDQL [Sea04].

As mentioned before, RDF is built on triples consisting of a subject, a predicate, and an object. The SPARQL query language is based on matching graph patterns, which contain *triple patterns*. A triple pattern also consists of a subject, predicate and object, but any or all of the elements may be replaced by variables. Variables are prefixed by a `?` or as an alternative by a `$`. A query returns each variable assignment that complies with the constraints specified by the graph patterns.

[20] Inference Graph

The SPARQL syntax is similar to the `SELECT...FROM...WHERE` style of the SQL syntax.[21] Thus, a simple query has the following structure:

```
PREFIX declaration of prefixes (optional)
SELECT requested variables
FROM specification of data sources (optional)
WHERE
{
  list of conditions, each formulated as triple and/or FILTER
     expression .
}
```

**Example 3.2:** Generic SPARQL Query

A `PREFIX` is basically the SPARQL equivalent of an XML namespace: It associates a short label with a specific URI and keeps the query slightly terser. The result of a query can either be accessed directly by the Jena API or be serialized into XML [W3C04f] or an RDF graph.

The Jena framework supports SPARQL through the ARQ query engine, which can process additionally RDQL queries and queries in its own language ARQ. Since the query engine is provided as an API, the functionalities can be embedded into Java code. For more details see [W3C04e] and [jen].

## 3.4 Persistence

Jena possesses a database subsystem which supports persistent storage of RDF data in relational databases, like MySQL, PostgreSQL, Oracle or Microsoft SQL Server. Therefore, the same interfaces such as Model, Resource and Query, are used to access and manipulate the stored RDF data. The database is not directly accessed by the application: All database-related tasks, like creating the necessary tables, are done transparently by Jena. By default, each model is stored in separate tables, whereas URIs are used to identify the models.[22] Jena trades off space for time and uses a denormalized schema in which literals and resource URIs are stored directly in statement (triple) tables. More details can be found in [jen].

[21] Note that there is no equivalent of the SQL INSERT, UPDATE, or DELETE statements.
[22] Currently, any string can be used, but the use of URIs is encouraged.

## 3.5 Reasoning and Inference

Ontology languages like RDFS or OWL allow additional facts to be inferred from some base data together with ontology information and the associated axioms and rules. Thus, inference engines (reasoners) are needed to make these entailments. As described in Section 3.2.3, Jena provides the Inference API for this purpose. The reasoners vary in their domain and their functional range and they can be either a part of the Jena framework or be external. First, the built-in reasoners are described, followed by the external reasoners.

### 3.5.1 Built-In Reasoners

Currently, Jena contains the following built-in reasoners:

- **Transitive reasoner**: This reasoner provides support for storing and traversing class and property hierarchies. Solely the transitive and symmetric properties of `rdfs:subPropertyOf` and `rdfs:subClassOf` are implemented.
- **Generic rule reasoner**: A rule-based reasoner which supports user defined rules. Furthermore, this reasoner is used to implement both the RDFS and OWL reasoners by instantiating the generic rule-based reasoner with a predefined set of rules. Forward chaining, tabled backward chaining and hybrid execution strategies are supported (see [jen] for details ).
- **RDFS reasoner**: Implements a (configurable) subset of the RDFS entailments. Thereby, almost all of the RDFS entailments described in [Hay03] are supported.
- **OWL reasoner**: Provides a set of useful but incomplete implementations of the OWL Lite subset of the OWL Full language (see Section 2.6). A default OWL reasoner and two small/faster configurations are included. Each of the configurations is intended to be a sound implementation of a subset of OWL Full semantics but none of them is complete (in the technical sense).
- **DAML micro reasoner**: Enables legacy support for the *DARPA Agent Markup Language (DAML)*. Essentially, the RDFS reasoner is augmented by axioms declaring the equivalence between the DAML constructs and their RDFS aliases.

### 3.5.2 External Reasoners

For complete OWL DL reasoning an external reasoner is needed. Due to the open architecture of Jena and in particular of the inference subsystem, external inference engines can easily be integrated.

**DIG-Interface.** The DIG-Interface is specified by the *DL Implementation Group (DIG)* and provides uniform access to DL reasoners. Therefore, the interface defines a simple HTTP-based protocol and offers a minimal set of operations (cf. [dig]). Jena uses this interface and therewith provides a transparent gateway between the ontology models and the external reasoners. At this time, available DIG reasoners are: Racer [Rac], FaCT [Hor] and Pellet.

**Pellet.** Pellet has been already introduced in Section 2.6.3. There are two different ways to use Pellet in Jena. Firstly, Pellet supports the DIG interface and can be accessed this way. Secondly, the Pellet interface can be integrated directly into Jena. The latter way is highly recommended because it is much more efficient and provides more inferences.[23] Thus, the Pellet interface is used in this thesis.

## 3.6 Summary

The essential concepts of the Semantic Web have been briefly introduced in Chapter 2. Based on these concepts, the Jena Semantic Web Framework has been presented in this chapter. Accordingly, Jena provides a wide range of functions to deal with ontologies. Additionally, the Pellet reasoner can be in combination with Jena to infer new facts from OWL DL ontologies.

Due to the fact that OWL DL is based on a (decidable) Description Logic, it shares its benefits and limitations. Hence, OWL DL is well suited for structuring knowledge by classes and properties. Furthermore, it provides means to reason about the explicit information. However, OWL does have a limited expressiveness, particularly in conjunction with properties (see [HPS04]). For example, it cannot express property-chaining rules like "an uncle is precisely a parent's brother" (see [HPSvH03]) and furthermore, it does not support default values, respectively non-monotonic inheritance.[24] For more information on further limitations please refer to [HPSvH03] and the given references.

Several research initiatives focus on these limitations and approaches for an integration of rules within the Web Ontology Languages have been proposed. An overview of some related approaches will be presented in Section 8. Since this thesis deals with a combination of DL reasoning and F-Logic rules the focus of the next chapter will be F-Logic and an implementation called Florid.

[23] The DIG interface lacks datatype support and the general expressiveness of DIG is not sufficient for OWL DL ontologies (cf. [Mar]).

[24] Non-monotonic inheritance is explained in more detail in Section 4.2.2. Here it is enough to know that non-monotonic inheritance allows to override inherited properties.

# 4 Florid - F-Logic Reasoning in Databases

*Frame Logic (F-Logic)* is a deductive, object-oriented language which can be used to represent ontologies (see Section 2.2) and other forms of Semantic Web reasoning (see [Kif05]). The main reference is [KLW95] while the underlying concepts where already presented in [KL89].

F-Logic integrates the paradigms of logic programming with the declarative semantics and expressiveness of deductive database languages. Furthermore it accounts for rich data modeling capabilities of object-oriented concepts and structural aspects of frame-based languages. The salient features include complex objects, non-monotonic multiple inheritance, polymorphism, class-hierarchies, signatures, uniform handling of data and metadata, query methods, and encapsulation (see [KLW95, FHK$^+$97]).

Several implementations of F-Logic are available, for example, FLORA-2 [Sto], Ontobroker [Ont], and Florid. The latter implementation was initially presented in [FHK$^+$97] and is used within this thesis.

**Florid.** *F-Logic Reasoning in Databases (Florid)* is a C++-implementation of F-Logic that implements all essential features. Additionally, it provides some extensions like *path expressions* which facilitate object navigation (see Section 4.4). Florid was developed by the Databases and Information Systems group at Freiburg University and was released in version 1.0 in 1996. Currently, version 4.0 is available which can be found on the Florid homepage [Dat]. For further information the study of the user manual [May00a] and the tutorial [May00b] is recommended.

The fundamental concepts and features of Florid and F-Logic are introduced by examples in the following sections.

## 4.1 Basic Syntax

An excerpt from the family tree of Abraham, depicted in Figure 4.1, is used in the subsequent sections to visualize and to exemplify the given explanations.

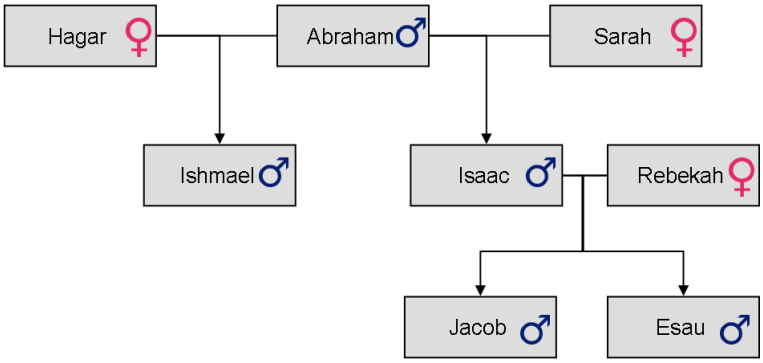


**Figure 4.1:** Biblical Family Tree

### 4.1.1 Objects

As previously mentioned, F-Logic integrates object-oriented paradigms and accordingly, objects build the basic concepts of F-Logic. They can be accessed by their *object names* while they are internally represented by *object identifiers (OIDs)*. Object names always begin with a lowercase letter, whereas variables always begin with an uppercase letter or an underscore followed by an uppercase letter. Object names and variable names are also called *id-terms* and are the basic syntactical elements of F-Logic. Examples for object names are `abraham`, `woman` or `son`, for variables are `Xy` or `_xy`. Two special types of object names exist: integers and strings. Strings are enclosed by quotation marks and may be used as object names, as well as positive or negative integers.

### 4.1.2 Methods

Following the object-oriented paradigm, methods are used to represent relationships between objects. They are expressed by *data-F-atoms* consisting of a *host* object, a *method* object, and a *result* object. Id-terms are used to denote the objects and all objects are allowed to occur in any place of a data-F-atom. A method which results in at most one object is defined as a *functional* method and is represented by a single-headed arrow "`->`". Methods that may result in more than one object are called *multi-valued* methods and are indicated by a double-headed arrow "`->>`". This leads to the following syntactical representation: "`host[method ->> result].`"

An exemplary functional method atom is: "`isaac[father->abraham].`" which expresses that `abraham` (result) is the `father` (method) of `isaac` (host). A similar example for a multi-valued method atom would be: "`abraham[son->>isaac].`" which states that `isaac` is a son of `abraham` and people in general may have additional sons.

**Methods with Parameters.** On various occasions the result of a method does not depend solely on the host object, but on some additional objects. These objects are embraced in F-Logic as *parameters*. Such parameters are always included in parentheses and separated by the symbol "`@`" from the method object. For example the atom "`abraham[son@(sarah)->>isaac].`" states that `abraham` has a son `isaac` who was born by `sarah`.

### 4.1.3 Class Membership and Subclass Relationship

Since F-Logic makes use of object-oriented concepts, objects can be categorized into *classes*. The class membership of an object is expressed by an *Isa-F-atom* which is denoted by a single colon "`:`". Furthermore, classes can be arranged in a class hierarchy by using subclass relationships. A subclass relation is expressed by *subclass-F-atoms* which are syntactically defined by a double colon "`::`". F-Logic permits that an object is an instance of several classes that are incomparable by the subclass relationship. Since classes are objects, Isa-F-atoms as well as subclass-F-atoms use id-terms to denote the objects and classes. Hence, it is possible to define methods on a class which is an instance of another class.

The atom "`man::person.`" defines that the class `man` is a subclass of the class `person`. Whereas the atom "`abraham:man.`" denotes that `abraham` is an instance of the class `man`. Because of the subclass-relationship between `man` and `person` it is possible to infer that `abraham` is also an instance of the class `person`.

### 4.1.4 Signatures

F-Logic provides *signature-F-atoms* to define which methods are applicable on the instances of certain classes. For this purpose, a signature-F-atom declares a method on a class and gives *type restrictions* for parameters and results. Similar to data-F-atoms, functional and multi-valued methods are denoted by "`=>`" respectively "`=>>`". Furthermore, it is possible to use a list of result classes (enclosed by parentheses) so that the result objects have to be in all listed classes.

For the given example (see Figure 4.1) it is reasonable to restrict the method `father` in such a way that the result object must belong to the class `man` and that the method is applicable for every `person`. The corresponding signature-F-atom is "`person[father=>man].`".

### 4.1.5 Miscellaneous

**F-molecules.** Moreover, it is possible and particularly convenient to collect several F-atoms into a single, more complex *F-molecule*. Example 4.1 contains all necessary facts about the family tree of Figure 4.1 in a fairly dense representation.

```
% signatures & sublcass relations
man::person.
woman::person.
person[father=>man].
person[mother=>woman].
% facts
isaac:man[father->abraham:man; mother->sarah:woman].
ishmael:man[father->abraham; mother->hagar:woman].
jacob:man[father->isaac; mother->rebekah:woman].
esau:man[father->isaac; mother->rebekah:woman].
```

**Example 4.1:** Family Tree as F-Molecules

**Negation-As-Failure.** F-Logic supports the *closed world assumption*: everything which is not explicitly known, is assumed to be false. This is also related to *negation-as-failure* which can be expressed as: "¬ P can be inferred if every possible proof of P fails" [Cla78]. Thus, drawn conclusions may become invalid if new information are added. This kind of reasoning is called *non-monotonic* (see Section 4.2.2).

## 4.2 Inheritance

F-Logic supports *structural* and *behavioral* inheritance. The former deals with inheritance of type restrictions for methods from superclasses to their subclasses and the latter refers to propagation of results of a method application from a superclass to its instances and subclasses.

### 4.2.1 Structural Inheritance

As mentioned above structural inheritance refers to the propagation of a type restriction from a superclass to its subclasses. In general, inheritability of a type restriction is depicted in F-Logic with a star attached to the appropriate arrow. This results in "`*=>`" for functional and "`*=>>`" for multi-valued methods. Florid omits the star and does not distinguish between inheritable and non-inheritable methods. Consider Example 4.2 which shows that type

restrictions accumulate: the person, a staff member reports to, must be a staff member and a manager.

```
staff::employee.
manager::employee.
employee[reports_to => staff].
staff[reports_to => manager].

% derived by inference:
employee[reports_to => employee].
employee[reports_to => manager].
% same as a combined F-molecule:
employee[reports_to => (employee,manager)].
```

**Example 4.2:** Structural Inheritance

### 4.2.2 Behavioral Inheritance

Inheritable methods are used to express behavioral inheritance. They are indicated by a prefixed star which leads to "`*->`" for inheritable functional methods and "`*->>`" for inheritable multi-valued methods. If an inheritable method is applied to a certain class, this method application and the corresponding result are inherited by all instances and subclasses of this class unless it is *overridden*. An inheritable method is overridden when it is specified by a more specific class. Example 4.3 depicts how a method can be inherited and be overriden. In general, birds are animals and can fly. However, `tweety` is a `penguin` and penguins cannot fly. Therefore, the method "`can_fly`" is overriden for penguins since the class penguin is the more specific class.

```
bird::animal.
penguin::bird.
bird[can_fly*->yes].
penguin[can_fly*->no].
tweety:penguin. % instance

% derived by inference:
tweety : animal.
tweety : bird.
tweety[can_fly->no].
```

**Example 4.3:** Non-monotonic Inheritance

This kind of inheritance is called *non-monotonic inheritance* and is a part of *non-monotonic reasoning*[25]. More precisely, the inheritance defines a *default* which can be overriden or even canceled by a more specific class. Hence, it is also called *default inheritance*. Note that an inheritable method becomes non-inheritable when it is inherited by instances but remains inheritable when it is passed to subclasses.

## 4.3 Predicate Symbols

Predicate symbols can be used in the same way as in predicate logic, e.g. in Datalog[26]. So called *P-atoms* are used to express information using a predicate. A P-atom consists of a predicate symbol followed by one or more id-terms separated by commas and included in parentheses. An example is "`father(isaac,abraham)`". Usually the information of a binary p-atom can can also be expressed by F-atoms, e.g. the information of the last example can be expressed as "`isaac[father->abraham]`".

Florid provides some built-in predicates that are very useful, e.g. the *equality* predicate and several *comparison* predicates.

**Equality.** An equality predicate is used to indicate that two object names are equal, i.e. they refer to the same object.[27] It is denoted with the "`=`" symbol and it is used in infix notation, e.g. "`abram = abraham.`". The equality predicate may be used in rule heads, rule bodies, facts, queries, and furthermore in combination with variables.

**Comparison.** Florid provides comparison predicates which are defined on objects denoting integer numbers. More precisely the predicates "`<`", "`<=`", "`>`", and "`>=`" may be used within a query or a rule body to compare integer numbers. Like the equality predicate, the comparison predicates are used in infix notation.

## 4.4 Path Expressions

So far, objects are accessed directly by their object names (see Section 4.1.1). In addition, it is possible to navigate to them along method invocations on other objects using so-called

[25] By contrast, monotonic reasoning does not allow that a new piece of knowledge reduces the set of what is known. Here, the fact "tweety is a penguin" invalidates the presumption that tweety can fly.

[26] Datalog is a query and rule language for deductive databases which is mainly used for research purposes. For further reading see [CGT90].

[27] Note that an object name refers to exactly one object, whereas an object may have more than one object name.

*path expressions*. Hence, a path expression describes how to navigate to the specific object. Path expressions generalise the dot-notation of object-oriented programming languages like Java.

In F-Logic a functional path expression is indicated with one dot between the object name and the method name, for example, the path expression "`isaac.father`" can be used to access the `father` of `abraham`. Consequently multi-valued path expressions are denoted with two dots, e.g. "`abraham..son`" accesses the sons of `abraham`. Moreover path expressions can be chained up by successively applying methods. For instance the antecedent expressions can be chained up to "`isaac.father..son`" to access the sons of the father of `isaac`. In addition, methods with parameters can be used in path expressions, e.g. "`abraham..son@(leah)`" can be used to access the sons of abraham who were born by leah.

Path expressions can be used at any position instead of an id-term. Additionally, it is possible to nest path expressions within F-molecules which leads to more concise molecules. Even an object for which no id-term is known can be denoted with a path expressions. This results in the creation of a new object, which can be especially useful in rule heads (see Section 4.5.1).

## 4.5 Rules and Queries

Besides the use of atoms and molecules, rules and queries can be utilized. Both are explained below.

### 4.5.1 Rules

One key feature of F-Logic is the usage of *rules* to derive new information from a given object base, i.e. to extend the object base *intensionally*. A rule consists of a *rule head* and a *rule body*. Both parts are separated by the symbol "`:-`" and a rule ends with a dot followed by a whitespace. When the preconditions of a rule (rule body) are satisfied then the conclusion (rule head) applies. The rule body consists of a conjunction of possibly negated molecules (so called *subgoals*), whereas the rule head is formed by a conjunction of molecules which are not negated.

As stated initially, variables begin with an uppercase letter or with an underscore followed by an uppercase letter. The latter kind of variables have a special meaning which is explained in more detail in the subsequent paragraph. Variables are used to pass information between subgoals and to the rule head.[28]

[28] Note, that the same variable in different subgoals of a rule defines a join condition.

Consider the following rules which extend Example 4.1.

```
X[descendant->>{Y}] :- Y[father->X].
X[descendant->>{Y}] :- Y[mother->X].
X[descendant->>{Z}] :- X[descendant->>{_Y}], _Y[descendant->>{Z}].
```

**Example 4.4:** F-Logic Rules

These rules define the transitive closure for the `descendant` method. The first rule, for instance, can be interpreted as "*if a person Y has a father X, then is Y a descendant of X*". Accordingly, the third rule defines the transitivity: "`if a person X has a descendant Y and Y has a descendant Z, then X has the descendant Z as well`". The following new information can be derived with these rules:

```
abraham [descendant ->> {isaac,ishmael,jacob}].
sarah [descendant ->> {isaac,jacob,esau}].
isaac[descendant  ->> {jacob,esau}].
hagar [descendant ->> {ishmael}].
rebekah [descendant ->> {jacob,esau}].
```

**Example 4.5:** Derived Facts from Rules

Furthermore it is possible to use *negation* within the rule body. A negated subgoal is denoted with a "`not`" prefix, for instance see Example 4.7.[29]

### 4.5.2 Queries

In addition to defining facts and applying rules it is essential to be able to explore the object base. For this purpose, F-Logic supports *queries*. They provide simple and natural means for exploring the object base both on *schema-level* and on *instance-level*.

A query is prefixed with the symbol "`?-`" and can be considered as a rule with an empty rule head. Variables are put in the syntactic positions which are of interest. Hence, the result of a query is returned as variable bindings such that the rule body is satisfied. Occasionally variables are used for intermediate results which should not appear in the answer set. For this purpose, *don't care variables* can be used which are variables starting with an underscore followed by an uppercase letter. They cannot be used in the rule head because such a variable is not propagated to the head.

For instance, the siblings of `jacob` could be queried with the following query:.

[29] Note, that every variable in a negated subgoal has to be limited by other positive subgoals. For more details on negation in Florid, e.g. negation of complex molecules, please refer to [May00b] or [KP88].

```
?- X[father->_Y;mother->_Z],jacob[father->_Y;mother->_Z].
% result:
X/jacob
X/esau
```

**Example 4.6:** Simple Query

The results are listed below the query. They contain `jacob` because he satisfies the conditions. In order to exclude him it is necessary to use negation:

```
?- X[father->_Y;mother->_Z],jacob[father->_Y;mother->_Z],not X=jacob.
% result:
X/esau
```

**Example 4.7:** Simple query with Negation

## 4.6 Summary

In this chapter, the basic concepts of F-Logic and the use of Florid as an implementation have been introduced. As aforementioned, F-Logic is very well suited to represent and to describe ontologies. Furthermore F-Logic provides an efficient inference engine which can be used to reason about instances and also about classes and their relations (cf. [AL04]). Nevertheless, Description Logics provide in some aspects more flexibility. For example, F-Logic does not support disjunctive information directly but only with significant weaker statements (cf. [Kif05]).

By now, the possibilities and limitations of Jena/OWL DL as well as of Florid/F-Logic have been briefly presented. To be able to overcome some of these limitations and to make use both of Description Logics and of F-Logic a combination is needed. An approach to combine Description Logic reasoning with F-Logic rules will therefore be the focus in the next chapter.

# 5 Combining Description Logic Reasoning with F-Logic Rules

This chapter focuses on the central subject of this thesis: the combination of DL reasoning with F-Logic rules.

As stated in Section 3.6, Description Logics are appropriate for structuring knowledge in terms of classes and relationships, but the expressiveness is limited in many respects, most notably with regard to reasoning about properties. Whereas, F-Logic offers powerful rules and some advanced features, like non-monotonic inheritance (see Section 4.6). Both worlds have their own benefits and drawbacks and allow for different usage scenarios. Large groups of researchers from various backgrounds are working on different approaches to bring both worlds together in order to take advantage of the best characteristics from both paradigms. Moreover, an integration of rule languages as a *rule layer* on top of OWL has already been envisioned by Tim Berners-Lee in his Semantic Web layer cake (cf. Figure 2.1).[30] The most prominent approaches that are related to this thesis will briefly be introduced in Chapter 8. The approach presented in this thesis is a combination of Description Logic reasoning with F-Logic rules. This approach is called *DL-Florid* and will be described in the subsequent sections.

**Family Genealogy.** For a better understanding, an exemplary ontology about a family genealogy is used throughout this chapter. This ontology describes the classes a family member (individuals) may belong to and the relationships (properties) within the family. Among others, the classes `Person`, `Man`, `Woman`, `Parent`, `Child`, `Sibling`, `Brother`, `Sister`, `Father`, `Mother`, `Aunt`, `Uncle`, `Ancestor`, and `Descendant` are defined. The corresponding relationships between the family members are `hasChild`, `hasParent`, `hasFather`, `hasMother`, `hasSibling`, `hasBrother`, `hasSister`, `hasAunt`, `hasUncle`, `hasAncestor`, and `hasDescendant`. The relationships are interdependent: The `hasChild` relation is the inverse of `hasParent` and an uncle (`hasUncle`) of a person is the brother (`hasBrother`) of a parent (`hasParent`). Furthermore, the `hasAncestor` and the `hasDescendant` relations are transitive, i.e. if `A` has an ancestor `B` and `B` has an ancestor `C`, then `A` has the ancestor `C` as well.

[30] This layering *on top* of OWL is subject of discussions and a weaker version of the layer cake, with rules and ontologies sitting side by side, has been proposed (cf. [HPPSH05, MHRS06].

The family consists of four generations; a graphical visualization of the family members and the genealogy is depicted in Figure 5.1. The complete ontology can be found in Example A.1 in the appendix. More details of this example will be explained in the subsequent sections, especially in Section 5.5.

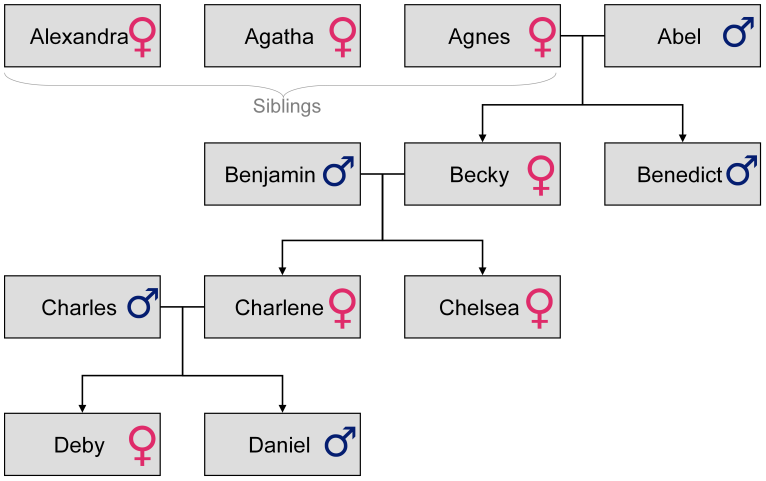


**Figure 5.1:** Family Genealogy

This chapter is structured as follows: At first, the DL-Florid system is introduced in general, followed by a description of the architecture. After that, the underlying relationships between Description Logics and F-Logic will be analyzed and the translation of facts and rules will be explained. Finally, the application of the presented approach on the family example will be described.

## 5.1 The DL-Florid Approach

DL-Florid is mainly based on the ontology language OWL DL in combination with a powerful DL reasoner. This "*DL world*" is augmented with the "*F-Logic world*" consisting of F-Logic rules. Special about this combination is the master-slave relation between both worlds: the OWL ontology acts as a *master*, which uses the F-Logic *slave* as a kind of a supporting tool. The general process can be described as follows. First of all, the DL reasoner is used to derive new information from knowledge already present. Afterwards, the F-Logic rules are applied to an exported subset of the ontology. If this inference leads to new information then this new information is added to the original ontology and the process restarts, with the now extended ontology. The details of the evolution strategy will be explained in Section 5.2.

At this point, it is important to stress that the ontology is the leading part in the whole process. More precisely, the ontology evolves step-by-step by adding newly derived informa-

tion from both the DL reasoner and the F-Logic rules. This means that the F-Logic system uses an one-way knowledge-base, i.e. the system receives the exported facts along with the defined rules in each iteration. Hence, it is necessary to define as much knowledge as possible within ontology and to exchange only the needed facts. The F-Logic rules should only be used for reasoning tasks that could not be done with OWL DL directly. A prototypical implementation of DL-Florid has been developed as part of this thesis and will be presented in Chapter 6.

## 5.2 Architecture

As aforementioned, the general evolution strategy of this approach is an iterative process which uses alternately a Description Logic reasoner and F-Logic rules in order to infer new facts. The general architecture therefore consists of two parts: An OWL DL part and an F-Logic part. Moreover, the newly derived facts are added to the existing facts. These new facts are used in combination with the "old" facts in the next iteration. These again may lead to additional new facts. This architecture is depicted together with the evolution strategy in Figure 5.2 which will be explained in more detail in the subsequent sections. The DL-Florid Wrapper holds the rules and coordinates the entire process. At first, the input components of DL-Florid will be introduced followed by the reasoning process itself.

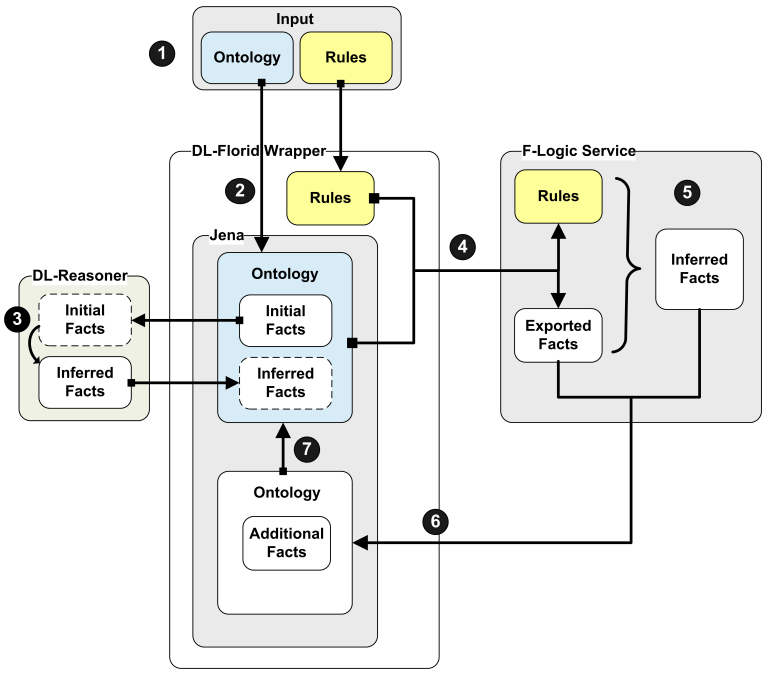


**Figure 5.2:** Architecture and Evolution Strategy

### 5.2.1 Input Components

In accordance with the described architecture, the input of DL-Florid consists of the following two components:

1. **Ontology**: An OWL DL ontology which can be encoded in any format that is supported by Jena. Such an ontology should already contain the definitions of all classes and properties that may be inferred by the F-Logic rules, even if these classes and properties are not used in the initial ontology. This modeling guideline reflects the master-slave relationship between Description Logic reasoning and F-Logic reasoning since the focus is on the ontology. Consequently, as much as possible should be defined directly in the ontology. Applied to the family example, this means that all possible properties (e.g. `hasUncle`) and classes (e.g. `Aunt`) are defined initially.
2. **Rules**: A set of F-Logic rules is specified in order to derive new facts. The underlying details of F-Logic rules have been explained in Section 4.5.1. Moreover, these rules are encoded in an XML markup. They are translated into F-Logic by the DL-Florid system. More details on the markup and the translation will be given in Section 5.4.

These input components are now used in the iterative reasoning process which is explained next.

### 5.2.2 Evaluation Strategy

A graphical representation of the iterative reasoning process is depicted in Figure 5.2. In this regard, the alternating process is divided into the steps listed below. These steps are indicated in the figure by their numbers and they are explained in more detail below.

1. As mentioned before, the input is composed of two parts: A set of F-Logic rules and an OWL ontology. Both components are loaded into the DL-Florid Wrapper.
2. Next, the Jena framework uses the given ontology to construct a model which is bound to an OWL DL reasoner (Pellet). Furthermore, the rules are converted from XML into normal F-Logic syntax.
3. The DL reasoner is used to derive new information from the constructed model. Afterwards, these new facts become part of the ontology and can be used again as "initial" facts (see Section 3.5). If no further information can be derived by the reasoner, i.e. a fixpoint is reached, the process continues with the next step.
4. A subset of the available facts (initial and inferred) is exported and converted into F-Logic (see Section 5.3.2). These translated facts are submitted to the F-Logic system along with the F-Logic rules subsequently.

5. The F-Logic system applies the defined F-Logic rules to the knowledge base in order to infer new information.
6. Afterwards, the F-Logic knowledge base is submitted to the DL-Florid Wrapper and translated back into an OWL compatible syntax (see Section 5.3.3). This converted knowledge base is used to build a new (temporary) ontology.
7. Now, the Jena framework is used to examine whether all information of the new ontology was already defined in the *old* ontology or whether some information is new. In the latter case, both ontologies are merged together, i.e. the new information is added to the old ontology and the process restarts with the extended ontology at Step 3. The process terminates if all information derived by Florid is already part of the ontology and no further information can be found.

The described process yields an evolving ontology. Each iteration adds new information, which can result in the inference of even more information in the next iteration. In the end, neither the DL reasoner nor the F-Logic system can discover new information. Both constructed models, the ontology as well as the exported subset, are complete and stable. However, the definition of new rules or the addition of new facts may necessitate additional iterations until a new stable state is reached.

## 5.3 Translation of Facts

It is essential for the proposed combination of DL reasoning and F-Logic rules that the facts can be translated and exchanged between both systems. This translation is presented as follows: First of all, the general relation between F-Logic and Description Logics is described. This is followed by the translation from OWL DL to F-Logic and vice versa. Moreover, the family example will be used to clarify the translation process.

### 5.3.1 The Relation between F-Logic and Description Logics

As stated in Section 2.6.2, some Description Logics form a decidable subset of First-Order Logic. Moreover, F-Logic is an extension of FOL with explicit support for object-oriented modeling (see Section 4 and [BH06] for general information). Therefore, it is obvious that F-Logic can be used to define a Description Logic subset which has been done in [Bal95]. In other words, F-Logic *subsumes* Description Logics (see [Kif05]). Apparently, this result can be used to translate (a subset of) an OWL DL ontology into F-Logic by using FOL as an intermediate step. Such a translation has been initially presented in [Bal95, Bor96]. In fact, a less expressive DL has been translated in these publications. However, the same principles

apply here as only a subset of OWL DL is considered. A translation of the basic constructs is denoted in the first two columns of Table 5.1. Note that $C$ and $D$ refer to classes whereas $P$ and $Q$ denote properties. Furthermore, the symbol $\top$ specifies the *universal class* which is often called *Thing*.

| | **DL** | **FOL** | **F-Logic** |
|---|---|---|---|
| (1) | $a : C$ | $C(a)$ | $a : C.$ |
| (2) | $\langle a, b \rangle : P$ | $P(a, b)$ | $a[P \twoheadrightarrow b].$ |
| (3) | $C \sqsubseteq D$ | $\forall x.C(x) \rightarrow D(x)$ | $C :: D.$ |
| (4) | $C \equiv D$ | $\forall x.C(x) \Leftrightarrow D(x)$ | $C = D.$ |
| (5) | $\top \sqsubseteq \leq 1P$ | $\forall x, y, z.(P(x, y)) \wedge P(x, z)) \rightarrow y = z$ | (see text) |
| (6) | $\top \sqsubseteq \forall P.C$ | $\forall y.(P(x, y)) \rightarrow C(y)$ | $x[P \Rightarrow C].$ |

**Table 5.1:** Partial Translation from DL to FOL to F-Logic (cf. [GHVD03, VDO03, BH06])

These intermediate results can now be used for the translation from FOL to F-Logic which is given in the last column of Table 5.1. It must be pointed out that the translation of a property (2) into F-Logic depends on its, i.e. a single-valued property, also known as a functional property, is denoted with a "$\rightarrow$", whereas multi-valued properties are depicted with a "$\twoheadrightarrow$". This is directly related to (5) which states that a property is functional. Therefore, no sole translation of statement (5) from OWL DL into F-Logic is needed.

Basically, only a small subset of the available DL constructs and axioms is translated into F-Logic. Assuming that more than these constructs are used for ontology modeling, only a subset of the ontology is exchanged between the DL and the F-Logic system. However, this is a fundamental principle of the approach presented in this thesis: the knowledge is mainly defined in the ontology (master) and the F-Logic rules (slave) are solely used for problems that are not solvable in DL. More precisely, F-Logic rules are defined in such a way that they make use of the exported subset and can infer new information.

### 5.3.2 From OWL DL to F-Logic

The described relation between Description Logics and F-Logic can now be applied to the precise constructs of OWL DL. As mentioned in Section 2.5, RDF and so RDFS and OWL base on statements (triples) which comprise a subject, a predicate, and an object. As a consequence, an OWL ontology consists of a list of statements.

The following list contains some central statements along with either the corresponding F-Logic translation or the explanation why it is not necessary to translate this statement.[31]

[31] Note that the used syntax for statements is similar to N3-Triple with prefixes (see [BL98a]).

Similar to the notation in Table 5.1, `C` and `D` refer to classes, `I` and `J` represent instances, and `P` denotes a property.

**Classes:**

1. `C rdf:type owl:Class`: Classes are defined via the categorization of objects. That is why sole definitions of classes without instances is not possible. However, a possibility to avoid this is to use the universal concept, i.e. the superclass `owl:Thing`. As in reality, everything is a *thing* and thus every individual is a member of the class `owl:Thing` and each class is therefore a subclass of `owl:Thing`. Here, `"owl:Thing"`[32] is used as the as the F-Logic equivalent of the universal concept. This leads to the following F-Logic atom:

   ```
   C :: "owl:Thing".
   ```

   According to the stated principle that F-Logic rules are used solely as a supporting tool, it is questionable whether empty classes are needed within the F-Logic system.

2. `I rdf:type C`: Under the assumption that C is a class, this statement denotes a class-instance. An *Isa-F-atom* is used for the corresponding translation in F-Logic:

   ```
   I : C.
   ```

3. `C rdfs:subClassOf D`: Such a subclass definition can be translated directly into the following *subclass-F-atom*:

   ```
   C :: D.
   ```

Some exemplary translations are stated below in Example 5.1. This example depicts some basic constructs of the family genealogy (Figure 5.1).[33]

```
% Class hierarchy                                  %       Person
family:Man     rdf:type         owl:Class .       %       /     \
family:Man     rdfs:subClassOf  family:Person .   %    Man      Woman
% translation
% "http://foo.org/dummy#Man" :: "http://foo.org/dummy#Person".

family:Woman   rdf:type         owl:Class .
family:Woman   rdfs:subClassOf  family:Person .
% translation
% "http://foo.org/dummy#Woman" :: "http://foo.org/dummy#Person".
```

**Example 5.1:** Translation of Class Hierarchy

[32] Florid supports strings and provides the class `url` as a special subclass of the class `string`. Section 6.5 will explain why and how the class `url` is used as a substitute for URIs. Here it is enough to know that strings as well as urls are enclosed in quotation marks.

[33] Note that the F-Logic translation uses full URIs due to the lack of direct support for namespaces in Florid. Furthermore, the prefix definitions for N3-Triple are omitted.

**Properties:**

1. `P rdf:type rdf:Property`: In F-Logic, properties are expressed by applying methods on objects. More precisely, *data-F-atoms* are used which consist of a host, a method, and a result object. Hence, the definition of properties which are not used in statements is not directly supported. Methods can be divided into functional and multi-valued methods. Consequently, the translation into F-Logic depends on the existence of the following statement.[34]
2. `P rdf:type rdf:FunctionalProperty`: As aforementioned, a single-head arrow denotes a functional property and a double-head arrow a multi-valued property. A property is considered to be multi-valued if the functional property statement is absent. However, the translation into F-Logic depends on the resource (individual) on which the property is applied. Such an application is given in the following statement (3).
3. `I P J`: Depending on statements (1) and (2), this statement can either be translated into

   ```
   I [ P -> J ].
   ```

   for a functional property or into

   ```
   I [ P ->> {J} ].
   ```

   for a multi-valued property.
4. `P rdfs:range C`: Such a statement asserts that the range of property P is of class C. This is translated into F-Logic using a *signature-F-atom* and an equivalence of the universal concept. Similar to the previous statement, the translation depends on cardinality of the property:

   ```
   "owl:Thing"[ P => C ].
   ```

   translates the range assertions for a functional property and

   ```
   "owl:Thing"[ P =>> (C) ].
   ```

   for a multi-valued property.

Again, some examples for the translation of properties and instances of the family genealogy are shown in Example 5.2.

```
% Range assertions
family:hasFather rdf:type owl:FunctionalProperty .
family:hasMother rdfs:range family:Woman .
% translation "owl:Thing"["http://foo.org/dummy#Mother"
%                         =>"http://foo.org/dummy#Woman"].
```

[34] Note that `rdf:FunctionalProperty` as well as `owl:ObjectProperty` and `owl:DatatypeProperty` are subclasses of `rdf:Property` (see Section 2.6).

```
% (functional)

family:hasSibling rdfs:range family:Person .
% translation "owl:Thing"["http://foo.org/dummy#hasSibling"
%                          =>>"http://foo.org/dummy#Person"].

% Instance
family:Becky rdf:type family:Person
% translation "http://foo.org/dummy#Becky"
%                          : "http://foo.org/dummy#Person".

% Properties
family:Becky family:hasSex family:FemaleSex
% translation "http://foo.org/dummy#Becky"
%                          ["http://foo.org/dummy#hasSex"
%                          -> "http://foo.org/dummy#FemaleSex"].
% (hasSex is functional)

family:Becky family:hasParent family:Abel
% translation "http://foo.org/dummy#Becky"
%                          ["http://foo.org/dummy#hasParent"
%                           ->>"http://foo.org/dummy#Abel"].
```

**Example 5.2:** Translation of Properties and Instances

The given translations use the basic atoms and molecules of F-Logic. More translations of OWL DL constructs are possible with the help of F-Logic rules. For example, a symmetric property (`P rdf:type owl:symmetricProperty`) may be translated into the rule:[35]

```
X [ P ->> {Y}] :- Y [ P ->> {X}].
```

However, such translations are not needed since this knowledge can be defined in the ontology and furthermore, the DL reasoner may use this information for inferring new information. Moreover, the F-Logic rules should make use of the translated classes, instances, properties and the class hierarchy.

### 5.3.3 From F-Logic to OWL DL

Based on the results of the previous section, the translation in the opposite direction, i.e. from F-Logic into OWL DL, can be done in a more straightforward way. The translations are de-

[35] Here, `X` and `Y` are variables whereas `P` stands for the property that is defined to be symmetric.

picted in Table 5.2. In accordance with the translation and the explanations of Section 5.3.2, one F-Logic atom/molecule may result in more than one OWL DL statement.

| F-Logic | Statement |
|---|---|
| `I : C` | `C rdf:type owl:Class` |
| `C :: D` | `D rdf:type owl:Class` |
| `I [ P -> J ]` | `P rdf:type owl:FunctionalProperty`<br>`I P J` |
| `I [ P ->> {J}]` | `I P J` |

**Table 5.2:** From F-Logic to OWL DL

This translation from F-Logic to OWL DL is kept minimalistic purposely, because a more comprehensive translation would result in redundant statements. For example, the first F-Logic atom in Table 5.2 can be translated into the following statements:

```
I rdf:type C
C rdf:type owl:Class
```

Although the first statement is not needed, since the DL reasoner infers automatically that `C` is an `owl:Class` on the basis of the second statement. Thus, the first statement would be redundant and is omitted.

Besides the described translation of facts, a translation of F-Logic rules from XML syntax into F-Logic syntax is needed. This translation is presented in the following section.

## 5.4 Translation of Rules

This section describes the XML markup which is used in this thesis to define F-Logic rules. Furthermore, the translation of such rules into "normal" F-Logic rules is explained.

### 5.4.1 XML Rule Markup

A simple and fairly dense XML markup is used to denote F-Logic rules. The basic XML markup is depicted in Example 5.3. Note that rules are defined for a specific ontology and that is why an independent translation is not supported.

```
<?xml version="1.0"?>
<rules>
  <rule>
  <!-- Definition of variables -->
    <var id="X" />
    <var id="Y" />
    ...
    <!-- Rule head  -->
    <head>
      <statement>
        <subject resource="#X" />
        <property resource="http://foo.org/dummy#property" />
        <object resource="#Y" />
      </statement>
      <predicate>
        <symbol resource="c" />
        <argument resource="#X" />
        <argument resource="#Y" />
      </predicate>
      ...
    </head>
    <!-- Rule body !-->
    <body>
      <statement negated="true" default="false" signature="false">
        <subject resource="#X" />
        <property resource="http://foo.org/dummy#property" />
        <object resource="#Y" />
      </statement>
      ...
    </body>
  </rule>
  ...
</rules>
```

**Example 5.3:** Generic F-Logic Rule in XML

Basically, the root element of the rule definition is `<rules>`. This node has several child nodes (`<rule>`), where each of these nodes represents an F-Logic rule. As described in Section 4.5.1, a rule consists of a head and a body and both parts may contain variables. For this reason, a rule definition embodies some variables (<var>), that define the existence of a variable, a rule head (`<head>`) and a rule body (`<body>`). Moreover, the basic constructs in the head and in the body are statements (`<statement>`) and predicates (`<predicate>`).

**Statement.** On the basis of an RDF statement, a statement consists of a subject, a property, and an object. Each of these elements denotes either a variable or a resource of the ontology identified by its URI. Moreover, a statement can be negated (`negated="true"`), can denote a signature, (`signature="true"`) or an inheritable property (`default="true"`). Note that negated statements may only be used in the rule body.

| Florid predicate | XML identifier |
|---|---|
| = | florid:equalOp |
| < | florid:lessOp |
| <= | florid:lessEqOp |
| > | florid:greaterOp |
| >= | florid:greaterEqOp |

**Table 5.3:** Built-in Comparison Predicates

**Predicate.** A predicate consists of a predicate symbol (`<symbol>`) and some arguments (`<argument>`). The predicate symbol can either be a self-defined predicate or a built-in comparison predicate (cf. 4.3).[36] The built-in predicates of Florid are tagged in the XML markup with specific identifiers which are depicted in Table 5.3. Furthermore, a predicate may be negated in the rule body by using the `negated="true"` attribute.

The described means can now be used for an XML representation of the following "hasAunt" rule:

$$hasAunt(x,y) \leftarrow hasParent(x,z) \wedge hasSister(z,y)$$

Example 5.4 denotes the corresponding rule in XML:

```
<rule>
  <var id="X" />
  <var id="Y" />
  <var id="Z" />
  <head>
    <statement>
      <subject resource="#X" />
      <property resource="http://foo.org/dummy#hasAunt" />
      <object resource="#Y" />
    </statement>
  </head>
  <body>
    <statement>
      <subject resource="#X" />
      <property resource="http://foo.org/dummy#hasParent" />
      <object resource="#Z" />
    </statement>
    <statement>
      <subject resource="#Z" />
      <property resource="http://foo.org/dummy#hasSister" />
```

[36] Note that the built-in predicates require two arguments.

```
      <object resource="#Y" />
    </statement>
  </body>
</rule>
```

**Example 5.4:** Exemplary F-Logic rule in XML

### 5.4.2 Translation

The translation of rules from XML into F-Logic is closely related to the translation of facts, which has been explained in Section 5.3.2. In particular, the translation of a rule statement is comparable to the translation of an OWL DL statement into F-Logic. Furthermore, the translation process depends on the ontology on which the rules should be applied. Firstly, subject, property, and object of a rule statement can refer to resources in the ontology by using URIs, or denote variables, or even create new resources. Secondly, some properties are defined to be functional and this definition is part of the ontology. As a consequence, this information is needed to translate a rule statement with a functional property into F-Logic. Predicates are translated in straightforward way into F-Logic by using either the F-Logic equivalent for built-in predicates from Table 5.3 or the given predicate.

For example, the statement in the head of Example 5.4 is translated into:

```
X["http://foo.org/dummy#hasAunt" ->> Y]
```

**Example 5.5:** Translated Statement

All statements and predicates are translated and afterwards these translations are used to form the rule. In accordance with this translation process, the rule from Example 5.4 would be translated into the following F-Logic rule:

```
X["http://foo.org/dummy#hasAunt" ->>Y]
        :- X["http://foo.org/dummy#hasParent"->>Z],
        Z["http://foo.org/dummy#hasSister"->>Y] .
```

**Example 5.6:** Translated Rule

## 5.5 Genealogy Example

The evaluation strategy, which was explained in Section 5.2.2, is now applied to the family genealogy, that was introduced in the beginning of this chapter (see Figure 5.1). First of all, the input components will be described, followed by the results of the iterative and alternating reasoning process.

### 5.5.1 Ontology

Initially, only the properties `hasParent`, `hasSibling`, and `hasSex` are used in combination with the class `Person`. Part a) of Figure 5.3 denotes the most important properties which describe the relationships between family members. All initially used properties are depicted with black points. Moreover, all properties and all classes are defined in the ontology even if they are not used at this point, e.g. the class `Aunt` is defined, despite the fact that no instance exists. An excerpt of the complete ontology can be found in Example 5.7 .

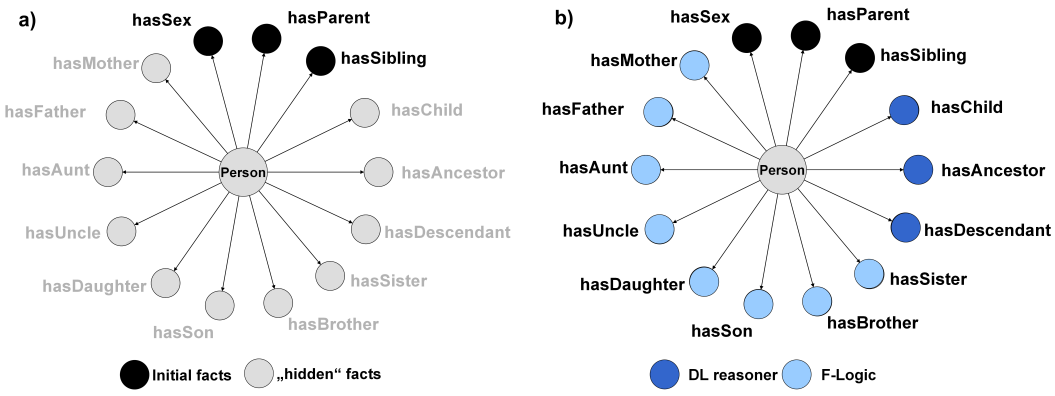


**Figure 5.3:** Genealogy Example

```
@prefix rdf:      <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix :         <http://foo.org/dummy#> .

:Becky    :hasSibling :Benedict .
:Becky    :hasParent :Agnes .
:Becky    :hasParent :Abel .
:Becky    :hasSex   :FemaleSex .
:Becky    rdf:type   :Person .
```

**Example 5.7:** Excerpt of the Family Ontology

This small excerpt describes a person named “Becky”, her sex, her sibling, and her parents. The complete ontology is available in Section A.1. Note that the names of the persons reflect the generation: names beginning with an “A” denote the first generation, “B” the second, et cetera.

### 5.5.2 F-Logic Rules

A set of F-Logic rules is used to infer relationships, which could not be inferred by the DL reasoner, for example, the `hasAunt` and `hasUncle` relations. Due to the extensive overhead

of XML, only the F-Logic equivalents are depicted in Example 5.8.

```
% siblings
X["family#hasSister" ->>Y] :- Y:"family#Woman"["family#hasSibling"->>
    X].
X["family#hasBrother" ->>Y] :- Y:"family#Man"["family#hasSibling"->>X
    ].
% father & mother
X["family#hasFather" ->>Y] :- Y:"family#Man"["family#hasChild"->>X].
X["family#hasMother" ->>Y] :- Y:"family#Woman"["family#hasChild"->>X
    ].
% uncle & aunt
X["family#hasAunt" ->>Y] :- X["family#hasParent"->>Z] , Z["family#
    hasSister"->>Y].
X["family#hasUncle" ->>Y] :- X["family#hasParent"->>Z] , Z["family#
    hasBrother"->>Y].
% niece & nephew
X["family#hasNiece" ->>Y] :- Y:"family#Woman"["family#hasAunt"->>X].
X["family#hasNiece" ->>Y] :- Y:"family#Woman"["family#hasUncle"->>X].
X["family#hasNephew" ->>Y] :- Y:"family#Man"["family#hasAunt"->>X].
X["family#hasNephew" ->>Y] :- Y:"family#Man"["family#hasUncle"->>X].
```

**Example 5.8:** F-Logic Rules for the Family Genealogy

The ontology and the rules are now used as the starting point for the reasoning process which is described next.

### 5.5.3 Reasoning

The reasoning process consists of the alternating application of the DL reasoner and the F-Logic rules. Each application and the resulting new information is described below.

1. **DL Reasoner.** First of all, the DL reasoner is applied to the ontology. The instances are classified in accordance with the class definitions, e.g. a male person is a `Man` or a person with a child is a `Parent`. Furthermore, the DL reasoner uses the known facts about the properties. As a result, the `hasChild` relations are inferred as the inverse of `hasParent` and the transitivity of `hasAncestor` and `hasDescendant` is derived. The new situation is depicted in part b) of Figure 5.3. The newly inferred properties are marked as *DL reasoner*. After that, a subset of the model is exported to F-Logic and submitted to the F-Logic system.
2. **F-Logic Rules.** The F-Logic rules are applied to the exported subset by the F-Logic system. These rules lead to a wide range of derived information. The new relationships

within the family are marked as *F-Logic* in part b) of Figure 5.3. After that, the knowledge base is converted and exported back into OWL DL.

3. **Merging.** Now, the exported knowledge base is used to create a new ontology, which is used to check whether F-Logic could derive new information. Since new information was inferred, the newly created ontology can be merged with the old ontology. This results in an addition of the new facts to the original ontology.
4. **DL Reasoner (2nd time).** As before, the DL reasoner is applied to the ontology. The new information from the F-Logic system result in a range of new classifications, e.g. a `Man` who has a nephew (`hasNephew`) is an `Uncle`. After that, the ontology is exported into F-Logic.
5. **F-Logic (2nd time).** Again, the rules are applied to the exported knowledge. But at this point no new information can be derived. However, the knowledge base is exported back into OWL DL.
6. **End.** The exported facts from the F-Logic system are already part of the ontology. Therefore, the F-Logic rules did not infer any new information and the process terminates.

The newly derived facts about Becky from each step of the described reasoning process are depicted in Example 5.9.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix : <http://foo.org/dummy#> .

% initial facts:
:Becky :hasSex :FemaleSex .
:Becky :hasParent :Abel .
:Becky :hasParent :Agnes .
:Becky :hasSibling :Benedict .
:Becky rdf:type :Person .

% DL reasoner(first time) .
:Becky :hasChild :Chelsea .
:Becky :hasChild :Charlene .
:Becky :hasAncestor :Abel .
:Becky :hasAncestor :Agnes .
:Becky :hasDescendant :Deby
:Becky :hasDescendant :Chelsea.
:Becky :hasDescendant :Charlene .
:Becky :hasDescendant :Daniel .
:Becky rdf:type :OffSpring .
:Becky rdf:type :Sibling .
:Becky rdf:type :Woman .
```

```
:Becky rdf:type :Parent .
:Becky rdf:type :Sister .
:Becky rdf:type :Daughter .
:Becky rdf:type :Mother .
:Becky rdf:type :Ancestor .
:Becky rdf:type :Descendant .

% F-Logic rules
:Becky :hasAunt :Agatha .
:Becky :hasAunt :Alexandra .
:Becky :hasBrother :Benedict .
:Becky :hasMother :Agnes .
:Becky :hasFather :Abel .

% DL reasoner (second time)
:Becky rdf:type :Niece .
```

**Example 5.9:** Reasoning Results for Becky

After the presentation of the DL-Florid approach, the next chapter describes the prototypical implementation.

# 6 Implementation

This chapter describes the prototypical implementation of DL-Florid, which is based on the theoretical concepts from Chapter 5. The chapter is organized as follows. At first, a brief overview of the employed technologies is given, followed by a description of the general architecture. Afterwards, the integration of Florid and of Jena will be explained in detail. Finally, the web interface will be presented.

## 6.1 Employed Technologies

The developed prototype and most of its components are implemented in Java. The Jena Semantic Web Framework plays a prominent role for working with OWL DL ontologies. Jena has been presented in detail in Section 3. Furthermore, the F-Logic rules are evaluated by Florid, which has been described in Section 4. The communication with Florid is realized as a Web Service. An insight to the used technologies is given next.

**Java.** Java is an object-oriented programming language, which has been invented and developed by Sun Microsystems. A Java application is typically compiled to bytecode, which makes the application executable on different operating systems as well as on different hardware. This platform independence is the main reason for the wide-spread use in software development, particularly for server-side applications. For this reason, Java is used for the implementation of DL-Florid. For more information on Java please refer to [Suna].

**Java Servlets and Java Server Pages.** Java servlets [Sunb] are server-side Java programs which provide a simple and consistent mechanism for extending the functionality of a web server and for accessing other systems. For this purpose, servlets receive requests and generate responses based on these requests. Arbitrary Java code can be executed to fulfill the request, e.g. a database can be accessed in order to retrieve data. Hence, the response to the user is dynamic. *Java Server Pages (JSP)* [Sunc] dynamically generate HTML, XML, or other types of documents in response to a web client's request. They are often used in combination with servlets. More details on servlets and JSP can be found in [BSB04].

First, the architecture and implementation of DL-Florid will be described on an abstract level followed by more detailed explanations in the subsequent sections.

## 6.2 General Architecture

The DL-Florid prototype loosely follows the *Model-View-Controlloer (MVC)* design pattern which is depicted in Figure 6.1. Basically, the MVC pattern decouples data access and business logic (*model*) from data representation (*view*) by using an intermediate component: The *controller*. For more information see, for example, [FFB04].

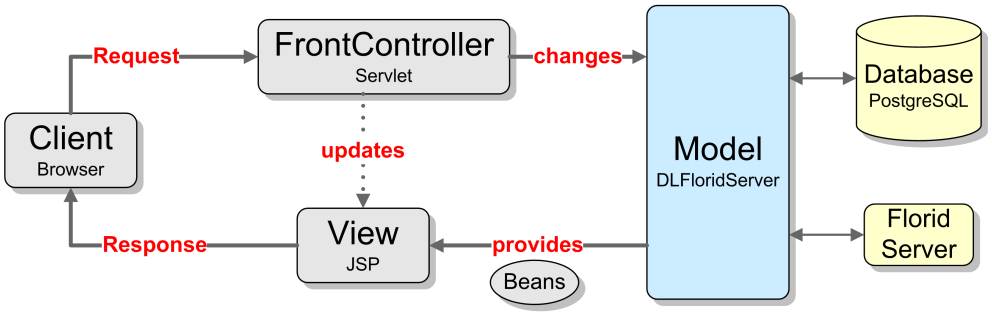


**Figure 6.1:** Model-View-Controller Pattern

- **Model**: The model contains the core functionalities of DL-Florid. It handles the iterative reasoning process and in particular the interaction with the Florid system. This interaction is realized with a Web Service so that the Florid Server itself is separated from the other parts. Moreover, the model is backed by a database as a persistent storage mechanism for ontologies and the corresponding F-Logic rules. The model encapsulates all relevant information into *Java Beans*[37] and makes them available to the view.
- **View**: The view consists of Java Server Pages which utilize the provided beans to access the requested information in order to display this information to the client.
- **Controller**: The controller is a servlet which receives and processes all requests that change the state of an user's interaction with the application. It determines the overall flow of the application. More precisely, the controller updates the model in a way appropriate to the user's action and the submitted data, for instance, an ontology or

[37] JavaBeans are used to encapsulate many objects into a single object (the bean), so that the bean can be passed around rather than the individual objects. For this purpose, the bean provides getter and setter methods.

a set of F-Logic rules. Moreover, the controller updates the view by specifying which particular Java Server Page is to be displayed to the client.

Essentially, the whole reasoning process of DL-Florid is part of the model. Whereas, the view and the controller build the web interface which provides means to work with the system.

In accordance with the MVC pattern and the specific structure of the reasoning process, the DL-Florid prototype is subdivided into the following packages.

- `org.semwebtech.dlflorid`: This is the base package of the DL-Florid prototype. It contains the `DLFloridServer` class which provides the core functionalities prototype and controls the alternating reasoning process. More details will be given in Section 6.3.
- `org.semwebtech.dlflorid.ontology`: All classes used for the work with ontologies and particularly for the translation of an ontology into F-Logic are within this package. A detailed description will be given in Section 6.4.
- `org.semwebtech.dlflorid.florid`: All classes related to Florid, in particular to the Florid web service, can be found in this package. Section 6.5 provides more details.
- `org.semwebtech.dlflorid.util`: Classes which are used by several other classes, e.g., a shared class which provides access to a database, and all utility classes can be found in this package. The precise classes are denoted in Section 6.6.
- `org.semwebtech.dlflorid.bean`: The JavaBeans used for the transfer of information from the model to the view are part of this package.
- `org.semwebtech.dlflorid.controller`: As the name implies, this is the package for the controller servlets which handle the interaction with the client. Together with the beans and the Java Server Pages, these servlets form the web interface which provides access to the DL-Florid system. The provided functions of this client are explained in Section 6.7.

Besides these packages, a set of Java Server Pages is used, which is not part of the package structure. These JSP consist of static HTML with some Java parts, for example, to access dynamic content in terms of JavaBeans. These JSP are part of the view of the DL-Florid prototype. They are utilized in the web client in Section 6.7.

## 6.3 The DLFlorid Server

As mentioned before, the DL-Florid server is the central component which handles and controls the alternating reasoning process between the Florid system and the DL reasoner. The `DLFloridServer` class and all related classes the server depends on are depicted in Figure 6.2. The reasoning process can either iterate autonomously or the user can interact

with the system and check the intermediate results. For this purpose, either the `auto` method is used for the autonomously iterating process or the provided methods, which correspond to the described reasoning strategy in Section 5.2.2, are used subsequently to track the reasoning process step-by-step. Accordingly, the following methods are provided.

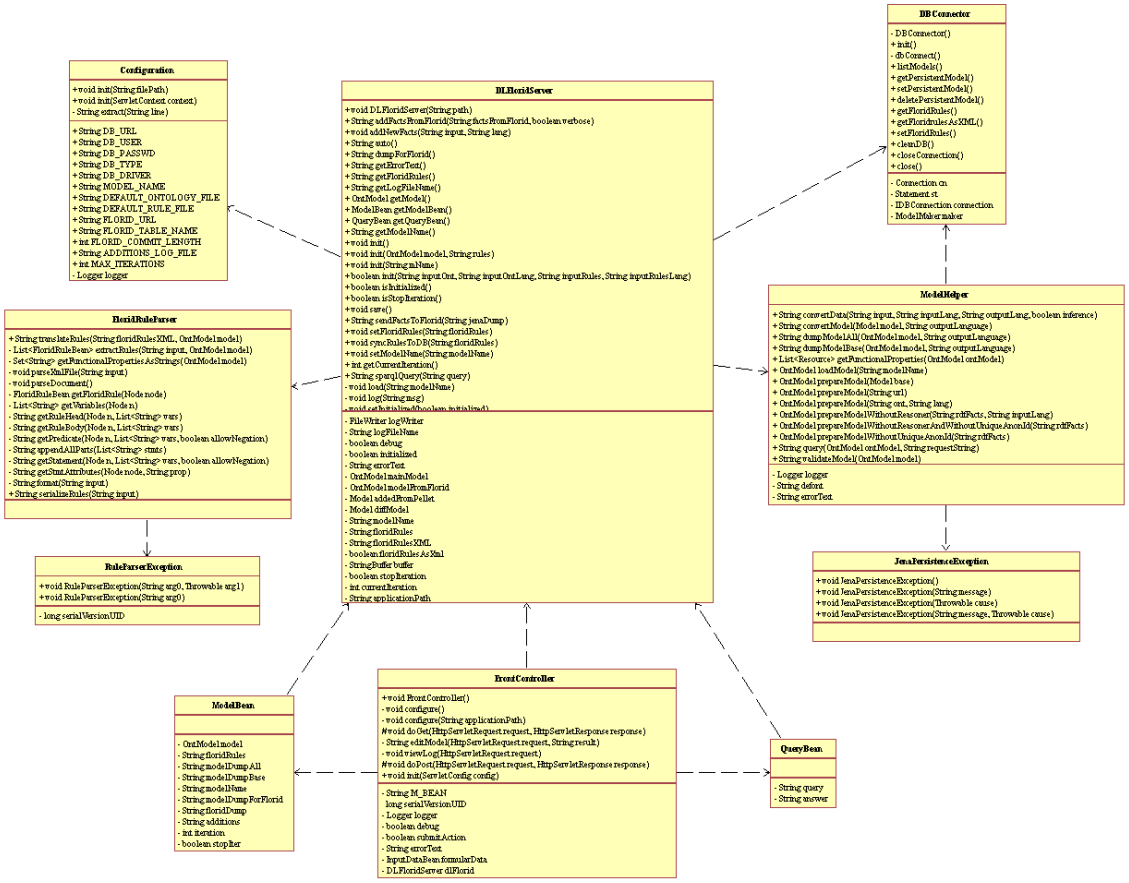


**Figure 6.2:** Class Diagram for the core DLFlorid Classes

- `init`: First of all, the system has to be initialized. Therefore, several methods for different usage scenarios are provided. The default `init` method expects a serialized ontology and a set of F-Logic rules, either in XML or in plain text. The `ModelHelper` class from the `jena` package is used in conjunction with the given facts to create the main ontology (a `OntModel`), which evolves during the reasoning process. In addition, this ontology is bound to the Pellet reasoner, which uses the given facts to infer new information and to complete the ontology.[38] Furthermore, all rules in XML are translated into plain F-Logic rules with the help of the `FloridRuleParser`. Both the rules

[38] Note that the reasoning of the Pellet reasoner is not initiated explicitly. In fact, Pellet processes the initial model as soon as it is bound to the model. Any further modifications to the model usually result in a restart of the reasoning.

and the ontology are kept in the `DLFloridServer` during the process.

- `auto`: This method starts the process which iterates autonomously until no new facts can be inferred. Finally, this method returns all facts which were inferred during the reasoning process.
- `dumpForFlorid`: The DL-Florid server utilizes the `ModelConverter` class to convert the ontology into F-Logic. More details on this class are given below.
- `sendFactsToFlorid`: The translated facts are submitted along with the rules to the Florid Server by using an instance of the `FloridWrapper`. This wrapper is also used to retrieve the facts from Florid and to translate them into *N-Triples*.
- `addFactsFromFlorid`: The retrieved facts are checked whether they are new. This is done by creating a new ontology from these facts with the help of the `ModelHelper` and by checking if this new ontology is a part of the initial ontology. More precisely, it is checked whether the difference between the new and the old ontology is empty. If the difference is not empty, the new ontology is added to the old ontology and the reasoning process restarts. Furthermore, a similar approach is used to identify all new facts which are inferred by Pellet on account of the newly added facts from Florid. In this case, the facts from Florid are added to a copy of the main ontology which is not bound to a reasoner. Besides this, the original ontology is amended with the facts from Florid so that Pellet can reason about these new facts. The differences between the copy and this ontology denote the facts which have been inferred by Pellet.

**Persistence.** Additionally, the used ontology and the corresponding rules can be saved in a database. For this purpose, the `DBConnector` class is utilized to gain access to the database. Moreover, details on the storage of an ontology are handled by Jena. Whereas the rules are stored directly in a separate table. A `modelName` is used to identify both, the ontology and the corresponding rules.

## 6.4 Ontology

In general, the Jena framework is used to work with ontologies. This section describes the most important functionalities in terms of ontologies which are implemented in this thesis.

### 6.4.1 Translation of Ontologies

The theoretical concepts of the translation of an OWL DL ontology into F-Logic have been given in detail in Section 5.3.2. On implementation level, this translation process is imple-

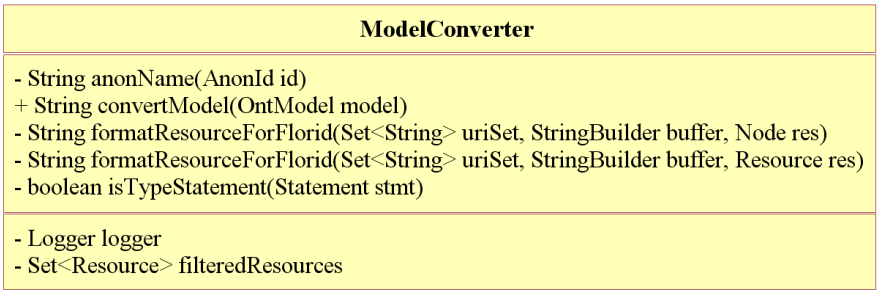


**Figure 6.3:** The ModelConverter Class

mented in the class `ModelConverter`. More precisely, the method `convertModel` translates the relevant subset of a given ontology into Florid-compatible F-Logic atoms and returns the translations. The class is depicted in Figure 6.3.

**Translation of Statements.** As mentioned before, a Florid atom can be the equivalent of more than one RDF statement (see Section 5.3.2). For this reason, the following translation strategy is used:

1. Get all defined classes from the ontology.
2. For each class get all direct subclasses and create the corresponding *subclass-F-atoms*.
3. Get the instances of every class and create the corresponding *Isa-F-atom*.
4. For each instance, the defined properties and the resulting *data-F-atoms* are created, either functional or multi-valued.
5. All (global) defined range assertions are retrieved and the corresponding *signature-F-atoms* are created.

The resources of a statement have to be translated as well, in order to comply with the F-Logic syntax. This translation is triggered as soon as a resource is considered which has not been translated before.

**Translation of Resources.** Florid does not support namespaces or URIs directly. Hence, the provided built-in class `url` is used as a replacement for real URIs. Since Florid encloses urls in quotation marks, every resource has to be enclosed in these as well. Furthermore, every resource has to be an instance of the `url` class. For example, a resource with the URI

`http://foo.org#Person`

is translated into

```
"http://foo.org#Person" : url.
```

Anonymous resources are translated in a similar way using the unique ID which Jena assigns to them. The translation of literals follows the notation of N-Triple. For example,

```
"10.2"^^<http://www.w3.org/2001/XMLSchema#decimal>
```

is translated into

```
"10.2^^http://www.w3.org/2001/XMLSchema#decimal".
```

Moreover, Florid does support integers directly. Thus, they are not translated but treated as normal numbers. Therefore, rules can be defined which utilizes these integers, for example, in conjunction with the duration of train connections (see Example 7.1). The translated ontology can now be submitted to the Florid server.

### 6.4.2 Ontology Handling

The class `ModelHelper` provides methods for the handling of ontologies. For example, to query an ontology, load an ontology from the database or to write an ontology to a string. Furthermore, several methods are implemented to create new ontologies from a given fact base, either with or without a reasoner.

Most of these methods strictly use the functionalities provided by the Jena APIs. In this context, Jena ensures that the ID (`AnondId`) of an anonymous resource remains unique within the whole system. Accordingly, the ID of an anonymous resource is changed, for example, when a new ontology is created from given facts and the ID of such a fact is already present, even if this resource is in another ontology. Applied to the reasoning process and particularly to the addition of facts from Florid to the main ontology, this would lead to false results: If an anonymous resource is submitted to Florid and then retrieved, a new (temporary) ontology is created on the basis of the retrieved facts. This new ontology would contain different IDs than the initial ontology. The identity of the anonymous resource in the new ontology with the resource in the initial ontology would be destroyed. As a consequence, this anonymous resource and all statements about this resource would be wrongly classified as newly inferred from Florid. Hence, the way Jena creates an ontology from given facts has been altered for this specific purpose. Since the `FloridWrapper` returns fact in N-Triple formatt, the `NTripleReader` has been modified. The altered reader (`NTripleReaderCustom`) strictly uses the IDs from the triples for the anonymous resources, even if these IDs already exist.

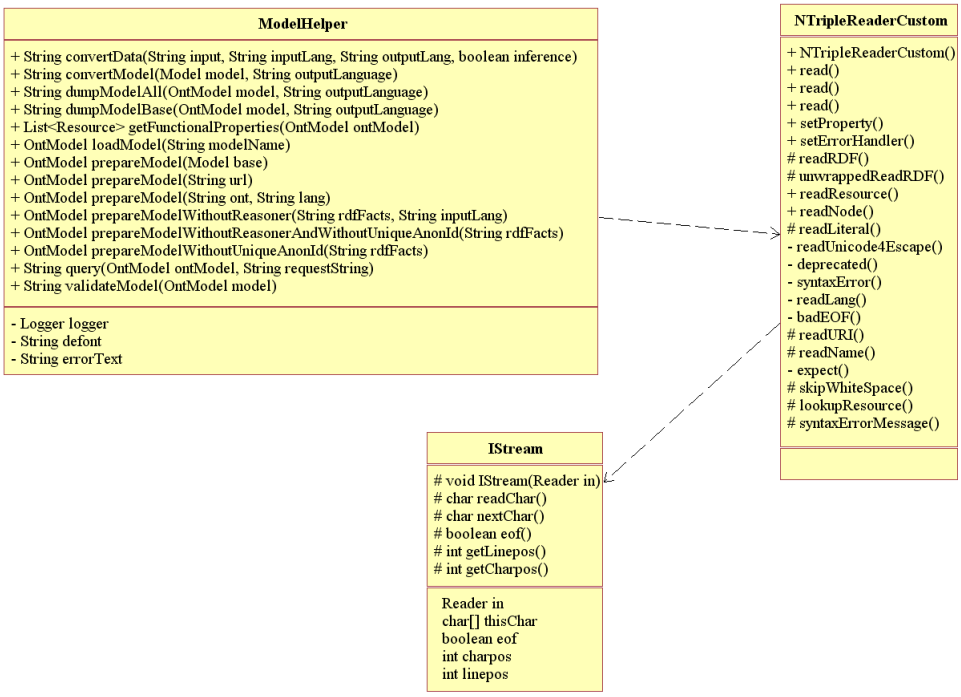


**Figure 6.4:** Class Diagram for the ModelHelper

## 6.5 Florid

The integration of Florid is implemented by means of a `FloridWrapper` around the Florid web service client (`FloridWSClient`).[39] Furthermore, a parser for the F-Logic rules in XML is provided.

**Florid Wrapper.** The Florid Wrapper is responsible for the integration of the Florid Server. Hence, the web service client is used to submit the fact base along with the rules to the server, evaluate the data and query the server in order to retrieve the facts. For example, the query `X : Y.` retrieves all instances and the corresponding classes. In accordance with the described translation process from Section 5.3.3, the Wrapper translates the retrieved facts into the N-Triple format. As mentioned before, all resources are enclosed in quotation marks. The class diagram for the `FloridWrapper` class is given in Figure 6.5.

**Translation of Rules.** The XML markup for rules and the theoretical translation into F-Logic rules have been described in 5.4. For this purpose, the `FloridRuleParser` class

[39] The web service client and the Florid Server have been implemented at the University of Freiburg. They are available on request from [Dat].

provides the `translateRules` method. This method uses the DOM-API to traverse the XML document in order to construct equivalent F-Logic rules. See Figure 6.2 for the corresponding class diagram.

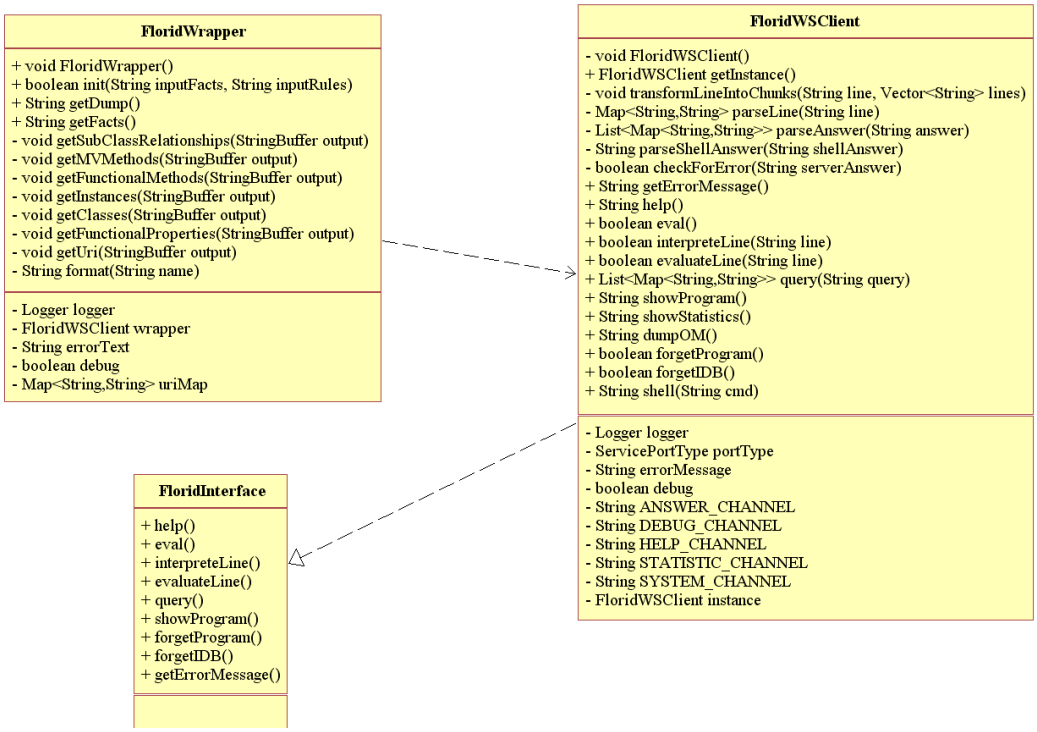


**Figure 6.5:** Class Diagram of the FloridWrapper

## 6.6 Other Classes

All classes that provide utility functionalities which are used by multiple components are kept below the subpackage `util`. Most notably classes for the handling of input forms (`FormHelper`) and for the handling of a database connection (`DBConnector`). The former class extracts the submitted form fields, their value and the content of any submitted file from the request and provides these data in terms of a `InputDataBean`. The latter class is realized as a singleton class, which means that only one instance exists at the same time. This class handles the connection to the underlying database for both the Jena framework and the DL-Florid specific tasks.

Two more JavaBeans are used within the DL-Florid prototype: `ModelBeans` and a `QueryBeans`. Both are kept in the session of the user. The `ModelBean` is instantiated by the DL-Florid server and contains the current ontology along with the name, the rules, the number of

already completed iterations, and a flag which denotes whether further iterations are necessary. This information is displayed to the user. The QueryBean is used to return an answer to a SPARQL query to the user. See for Figure 6.2 for the class diagrams of both beans.

## 6.7 Web Interface

The prototypical implementation of DL-Florid provides a client by means of a web interface. This web interface is mainly backed by the `FrontController` servlet, which processes the requests from the users and triggers the corresponding actions on the DL-Florid server. For example, a user submits an ontology and a set of rules to the server. This data is part of the request which is processed by the `FronController`.[40] However, Java Server Pages are used for the graphical representation. The functions provided by the client can mainly be summarized by the following parts:

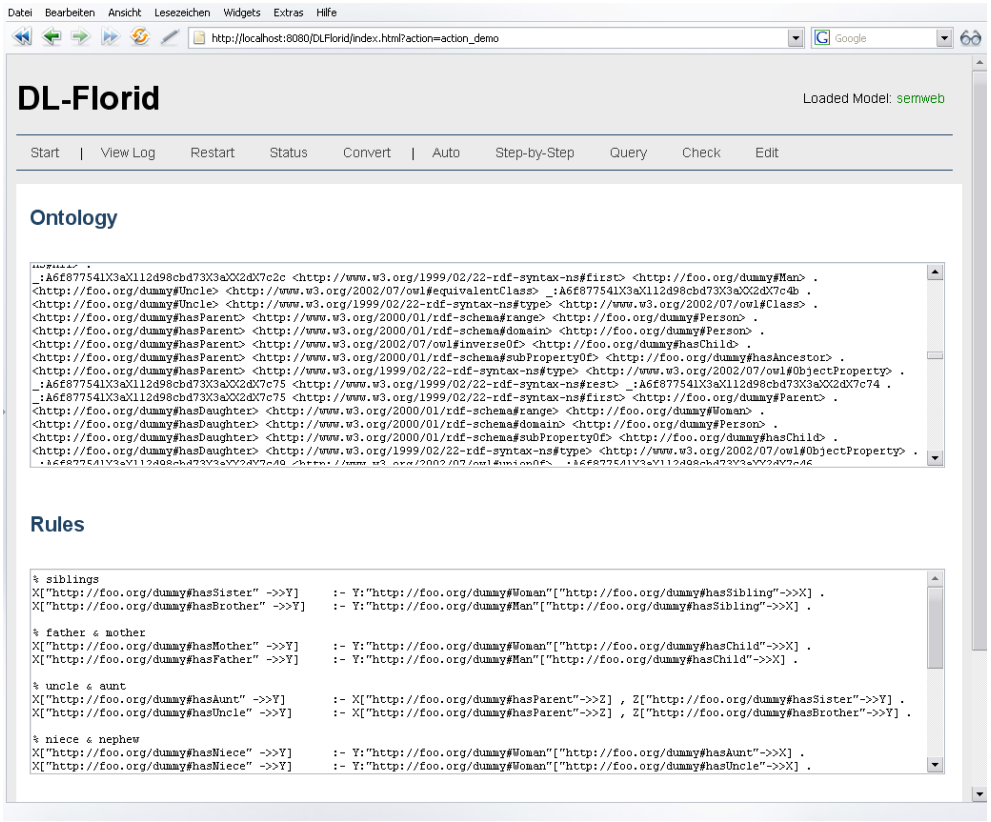


**Figure 6.6:** Using the Web Client for the Initialization of the System

- The client allows for the initialization of the system in one of the following ways:

[40] To be more precisely, the controller utilizes the `FormHelper` to retrieve the data from the request.

  - A new ontology can be created, either from an uploaded file or an input field. Furthermore, a set of F-Logic rules can be defined. Again, by processing an uploaded file or an input field. The ontology can be marked up in RDF/XML, N3, or N-Triple. Whereas, the rules can be either defined in XML or as plain text. Figure 6.6 depicts a newly initialized ontology along with the corresponding F-Logic rules.
  - As an alternative, an ontology and a set of corresponding rules can be loaded from the database.
  - Additionally, some exemplary ontologies along with the corresponding rules are provided which can be used as the starting point.
- Once the system is initialized, the user can modify the ontology by adding new facts, altering defined rules, or by even adding new rules.
- The reasoning process can either be initiated to run autonomously or step-by-step. In the latter case, the intermediate results are displayed to the client and the system waits for the next action whereas in the former case the system only displays the final results. Refer to Figure 6.7 for a visualization of the the final ontology along with the additions from Florid as well as from Pellet.

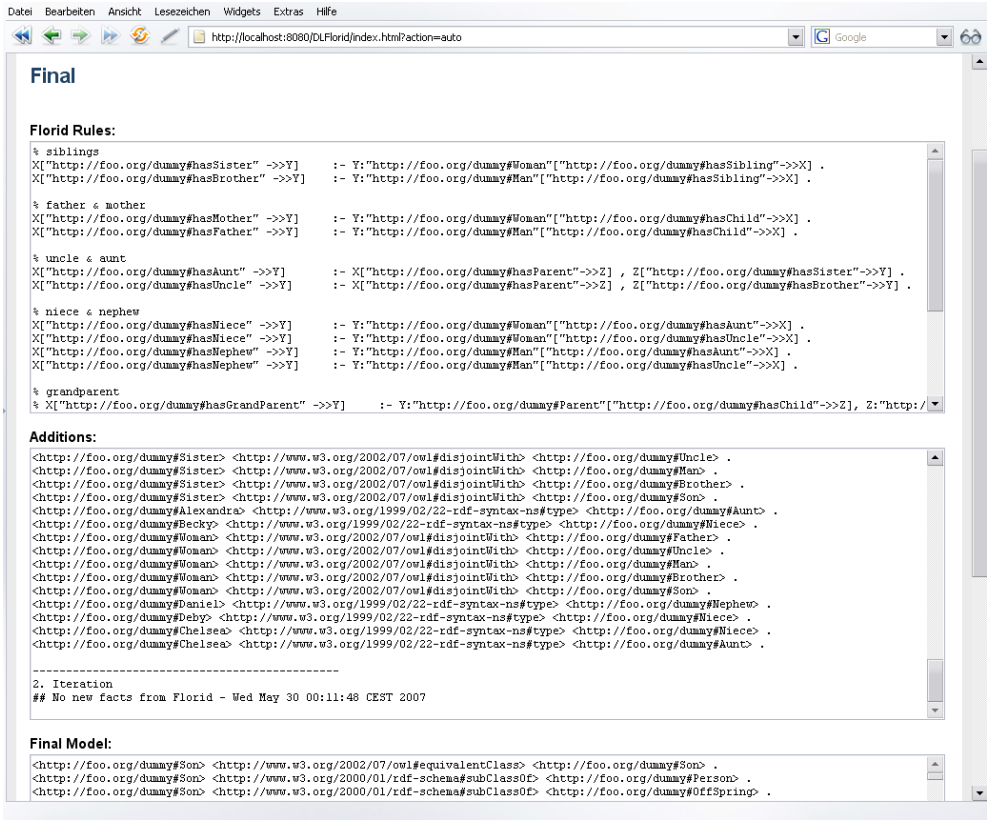


**Figure 6.7:** Viewing the final Ontology and Additions with the Web Client

- The user can query the ontology by means of SPARQL. For this purpose, the namespaces defined in the ontology are retrieved and processed to build a query template. The user can use this template as the scaffolding for the intended query. Figure 6.8 depicts an exemplary query and the returned results.
- As stated initially, all additions from either Florid or from Pellet are logged to a file. These additions can be viewed using the web client. Some exemplary additions are presented in Figure 6.9.

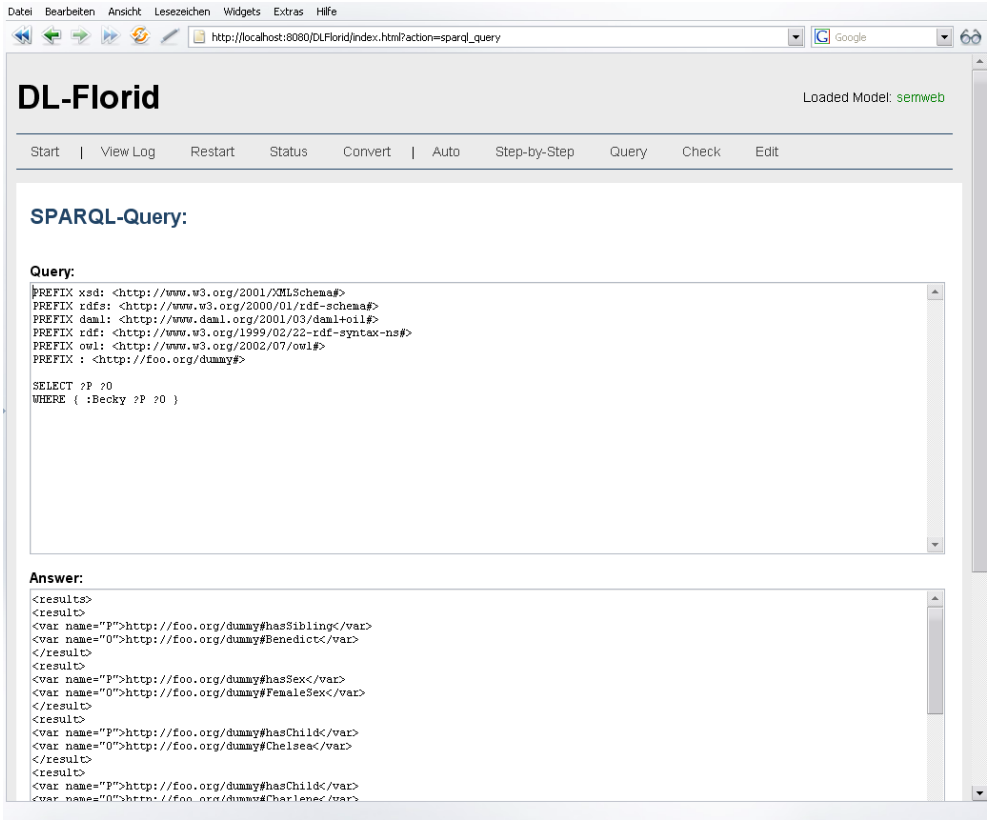


**Figure 6.8:** Sending SPARQL Queries and Viewing the Results with the Web Client

- Furthermore, the user can add new facts to the used ontology and edit the rules. As a consequence, the reasoning process has to restart because the new facts could lead to new results.

Note that all actions related to the ontology and to the rules are only allowed if the system is initialized. This is the case if a valid `ModelBean` can be found in the session of the user. Otherwise the relevant actions are disabled.

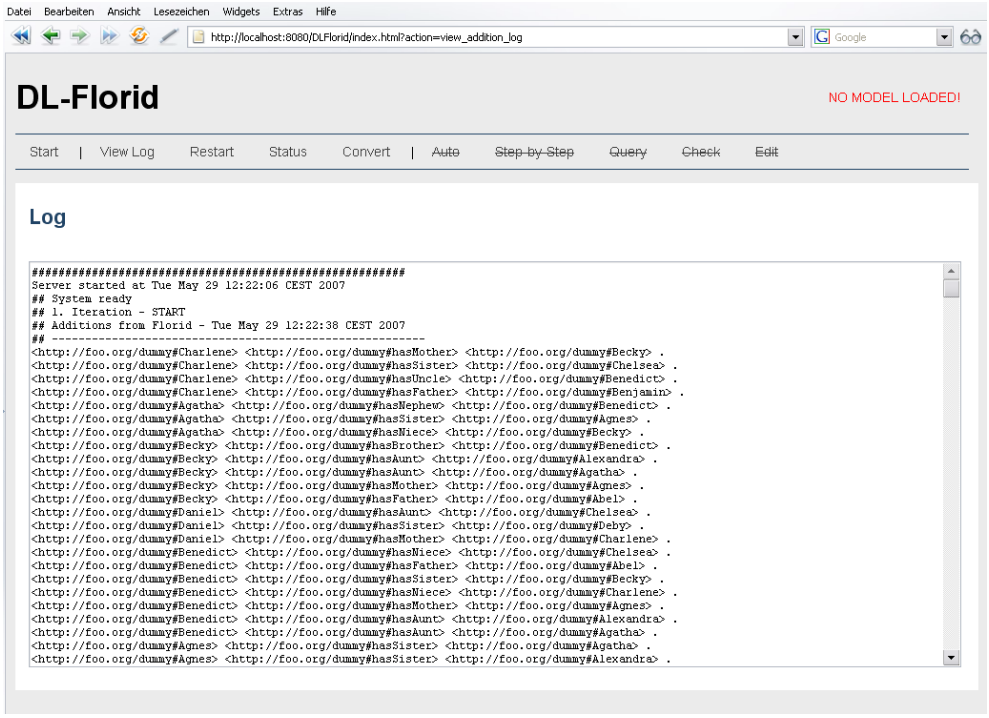


**Figure 6.9:** Viewing the Additions Log with the Web Client

# 7 Evaluation

An approach to augment Description Logic reasoning with F-Logic rules has been presented in the previous chapters. This chapter provides an evaluation of the presented approach. First of all, the benefits are described with the help of some examples in Section 7.1. Finally, the performance of DL-Florid is considered in Section 7.2.

## 7.1 Benefits

One of the main strength of the DL-Florid approach is that it provides support for OWL DL ontologies *combined* with F-Logic rules. The proposed combination uses an alternating reasoning process which results in an evolving ontology, i.e. the newly inferred facts from one system, either DL reasoner or F-Logic system, are added to the ontology. Hence, these facts are available for the other system in the next iteration. Moreover, the DL-Florid approach utilizes existing tools, namely Jena, Pellet and Florid, which are well-established and are being used in various existing applications.

An overview of the precise benefits of DL-Florid in terms of expressiveness is given below.

### 7.1.1 Property Chaining and Rules

As explained in Section 5.5, DL-Florid allows to chain properties, i.e. to use composition constructors to define, for example, the uncle relationship via the composition of the brother and the parent relationship (see Example 5.6).

Even more general, DL-Florid supports (almost) the full power of F-Logic rules, e.g., for transfer of properties. Furthermore, the built-in predicates of Florid can be used within rules. Consider the following example which derives train connections from a given set of sections.[41] The complete ontology can be found in Example A.2 in the appendix.

*A* `section` *connects a* `startPoint` *with an* `endPoint`*, for example, the section* `#Augsburg-Wuerzburg` *starts in Augsburg and ends in Wuerzburg. Furthermore, a section has a* `duration` *and a section is the smallest part, which means that no further stop between the*

[41] Note that a section from A to B is not the same as a section from B to A.

*startPoint and the endPoint exists. These sections are used to construct connections. A connection interlinks two cities by means of several sections. For example, the connection #Augsburg_Muenchen_Parsing connects Augsburg with Muenchen Parsing via Wuerzburg. Thus, two sections are used to form this connection. Accordingly, the duration of a connection depends on the durations of the underlying sections.*

The following rules can be used to derive the connections and their duration.

```
% (1) A section is also a connection, transfer the property values from
%     the section to the connection.
c(A,B):"http://localhost/test.rdf#Connection"[
        "http://localhost/test.rdf#startPoint" ->> A;
        "http://localhost/test.rdf#endPoint" ->> B;
        "http://localhost/test.rdf#duration" ->> D]
:- _Y:"http://localhost/test.rdf#Section"[
        "http://localhost/test.rdf#startPoint"->> A;
        "http://localhost/test.rdf#endPoint"->> B;
        "http://localhost/test.rdf#duration" ->> D].

% (2) A connection can be a section combined with a connection.
%     The duration of a new connection is the total of the duration of
%     underlying connection and section. The connection between two
%     cities is constructed this way only once.
c(A,C):"http://localhost/test.rdf#Connection"[
        "http://localhost/test.rdf#startPoint" ->> A;
        "http://localhost/test.rdf#endPoint" ->> C;
        "http://localhost/test.rdf#duration" ->> D]
:-c(A,_B):"http://localhost/test.rdf#Connection"[
        "http://localhost/test.rdf#startPoint" ->> A;
        "http://localhost/test.rdf#endPoint" ->> _B;
        "http://localhost/test.rdf#duration" ->> E],
X:"http://localhost/test.rdf#Section"[
        "http://localhost/test.rdf#startPoint"->>_B;
        "http://localhost/test.rdf#endPoint"->>C;
        "http://localhost/test.rdf#duration" ->> F],
        D=E+F,
        not c(A,C):"http://localhost/test.rdf#Connection".

% (3) If more than one connection between two cities is possible, the
%     connection with the shortest duration is prefered.
c(A,C):"http://localhost/test.rdf#Connection"[
        "http://localhost/test.rdf#startPoint" ->> A;
        "http://localhost/test.rdf#endPoint" ->> C;
        "http://localhost/test.rdf#duration" ->> D]
:- c(A,_B):"http://localhost/test.rdf#Connection"[
        "http://localhost/test.rdf#startPoint" ->> A;
        "http://localhost/test.rdf#endPoint" ->> _B;
        "http://localhost/test.rdf#duration" ->> E],
X:"http://localhost/test.rdf#Section"[
        "http://localhost/test.rdf#startPoint"->>_B;
        "http://localhost/test.rdf#endPoint"->>C;
        "http://localhost/test.rdf#duration" ->> F],
c(A,C):"http://localhost/test.rdf#Connection"[
        "http://localhost/test.rdf#duration" ->>T], D=E+F, D<T.
```

**Example 7.1:** Rules to derive Train Connections

These rules are evaluated and the derived facts are added to the ontology. Note that in this case, the F-Logic rules create new resources, namely the connections, using `c(A,B)`. The resulting ontology can now be queried using SPARQL. For example, the following query retrieves all connections starting in Goettingen and the corresponding durations:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX : <http://localhost/test.rdf#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>

SELECT ?Connection ?Duration
WHERE {
        {?Connection rdf:type :Connection}
        {?Connection :duration ?Duration}
        {?Connection :startPoint :Goettingen}
}
```

**Example 7.2:** SPARQL Query

This query results in the following answer:

```
<result>
  <var name="Connection">c(_http://localhost/test.rdf#Goettingen_,
     _http://localhost/test.rdf#Berlin_Ostbahnhof_)</var>
  <var name="Duration">150^^http://www.w3.org/2001/XMLSchema#integer<
     /var>
</result>
<result>
  <var name="Connection">c(_http://localhost/test.rdf#Goettingen_,
     _http://localhost/test.rdf#Munich_Hbf_)</var>
  <var name="Duration">239^^http://www.w3.org/2001/XMLSchema#integer<
     /var>
</result>
<result>
  <var name="Connection">c(_http://localhost/test.rdf#Goettingen_,
     _http://localhost/test.rdf#Hamburg_Hbf_)</var>
  <var name="Duration">118^^http://www.w3.org/2001/XMLSchema#integer<
     /var>
</result>
...
```

**Example 7.3:** Query Result

These rules go far beyond the expressiveness of OWL DL and thus, they provide an insight into the possibilities of DL-Florid.

### 7.1.2 Closed-World Reasoning and Defaults

As mentioned before, the Web Ontology Language follows the open world assumption. In general, this assumption is reasonable on the huge and only partially known World Wide Web. Nevertheless, in some cases the opposite assumption, the close world assumption, is needed. For example, it should be possible to define that in general, a bird can fly. Whereas a penguin is a bird but cannot fly. More precisely, a bird can fly by default, unless it is stated explicitly that the bird is a *"non flier"*, for example, a penguin. Consider the *Tweety* example from Section 4.2.2. The following ontology denotes the basic facts and particularly the class hierarchy along with the instances.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE rdf:RDF [
    <!ENTITY owl   "http://www.w3.org/2002/07/owl#">
    <!ENTITY xsd   "http://www.w3.org/2001/XMLSchema#">
]>
<rdf:RDF
  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#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns="http://tweety#"
  xml:base="http://tweety">

  <owl:Class rdf:ID="Bird" />
  <owl:Class rdf:ID="Flier" />
  <owl:Class rdf:ID="NonFlier">
    <owl:complementOf rdf:resource="#Flier" />
  </owl:Class>

  <owl:Class rdf:ID="Penguin">
    <rdfs:subClassOf rdf:resource="#Bird" />
    <rdfs:subClassOf rdf:resource="#NonFlier" />
  </owl:Class>

  <Penguin rdf:ID="Tweety"/>
  <Bird rdf:ID="Foo"/>
  <!-- This is needed because
       unused properties are not translated into F-Logic !-->
  <owl:FunctionalProperty rdf:ID="flies">
    <rdfs:range rdf:resource="&owl;Thing"/>
  </owl:FunctionalProperty>

  <owl:FunctionalProperty rdf:ID="notFlies">
    <rdfs:range rdf:resource="&owl;Thing"/>
  </owl:FunctionalProperty>

</rdf:RDF>
```

**Example 7.4:** Tweety Ontology

Accordingly, the following F-Logic rules provide the described default values.

```
"http://tweety#Yes" : url.
"http://tweety#No" : url.
% By default, a bird flies, if it is not a non-flier
X["http://tweety#flies" -> "http://tweety#Yes"]
        :- X:"http://tweety#Bird", not X:"http://tweety#NonFlier".

% non flier does not fly
X["http://tweety#flies" -> "http://tweety#No"]
        :- X:"http://tweety#NonFlier".
```

**Example 7.5:** Rules for Birds and Penguins

As a consequence, such F-Logic rules can be used for the closed world assumption (*"a bird can fly until it is known that it is a penguin"*) and thus, DL-Florid provides means for non-monotonic inheritance.

### 7.1.3 Queries

In general, the SPARQL query language is used for querying RDF graphs. However, SPARQL provides only limited means for more advanced queries and moreover, poor performance on large data sets. As a solution, the DL-Florid system can be used for complex queries. For this purpose, the F-Logic rules utilize a new (temporary) class to classify all facts which comply with the query into this class. As the facts are added to the main ontology, a simple SPARQL query retrieves the facts. For example, the *Mondial* database contains a large amount of data about countries, cities, organizations, et cetra. Please refer to [May99] for further details. Due to the large amount of data and the way SPARQL queries are evaluated, the performance is poor. The following F-Logic rule classifies all countries (`country`) which have a city with a population over 100,000 into the class `result`.

```
"http://www.semwebtech.de/mondial/10/meta#result":url.
X:"http://www.semwebtech.de/mondial/10/meta#result"
:- X:"http://www.semwebtech.de/mondial/10/meta#Country",
   Y:"http://www.semwebtech.de/mondial/10/meta#City",
   Y["http://www.semwebtech.de/mondial/10/meta#cityIn" ->> X],
   Y["http://www.semwebtech.de/mondial/10/meta#population" ->> P],
   P > 100000.
```

**Example 7.6:** F-Logic Rule for Mondial

The classified city can now easily be queried with SPARQL. This use of F-Logic rules is similar to defining views in a relational database. Furthermore, the view is stored in the ontology and can be easily reused.

The previous sections gave an insight into the expressiveness of F-Logic rules in conjunction with Description Logic reasoning. The subsequent section deals with the performance of DL-Florid and particularly of the developed prototype.

## 7.2 Performance

In general, the performance of DL-Florid prototype has been entirely satisfactory. However, two major bottlenecks were identified: First, the integration of Florid by using a web service is for sure not the best solution for large amounts of data. Second, the used Description Logic reasoner Pellet is considerably slow for some ontologies, depending on the complexity of the used constructs.

In addition, the rules must be defined in accordance with the ontology. To be more precisely, the considered classes and properties should be defined in the ontology and the rules should refer to these resources using the corresponding URIs. Thereby, the ontology and the rules must fit together: The user has to take care not to define cyclic combinations. Otherwise, the reasoning does not reach a fixpoint and the ontology is extended in every iteration. For example, the train example from Section 7.1.1 can easily be modified to a cyclic reasoning process. For such a cyclic problem, the definition of the `startPoint` property has to be altered. The new definition is depicted in Example 7.7.

```
<owl:ObjectProperty rdf:ID="startPoint">
  <!-- This leads to cycles !-->
  <rdfs:range rdf:resource="#Section" />
  <rdfs:subPropertyOf rdf:resource="#routePoint"/>
</owl:ObjectProperty>
```

**Example 7.7:** Exemplary Rule for Cycles

As stated before, the F-Logic rules utilize the defined section to create new connections. Similar to sections, these connections have a `startpoint` and an `endpoint`. When these new connection are added to the ontology, the Pellet reasoner classifies these connections as sections because of the range assertions of the `startpoint` property. As a consequence, Florid would infer new connections on account of these new sections. These new (and wrong) connection are added to the ontology, classified as sections, and the reasoning restarts. Therefore, the definition of rules and of ontologies must be carefully coordinated. Moreover, the proposed combination is undecidable. This can be proved by showing that it is possible to encode a known undecidable problem with the means provided by DL-Florid (cf. [HPSBT05]). An often used example is the undecidable domino problem.[42] However, the lack of decidabil-

[42]This problem is also known as the *Wang Tiles*. For more details please refer to [CSHD03].

ity is accepted since the DL-Florid system terminates in most of the cases and the provided functionalities overweight this disadvantages.

The following chapter describes some related approaches and denotes further work.

# 8 Further and Related Work

First, this chapter gives an overview over some approaches which are related to this thesis. Afterwards, the next steps in the development of the DL-Florid approach are described.

## 8.1 Related Approaches

A wide range of languages and standards have been emerged around the integration of rules in the field of the Semantic Web. Similar to the presented DL-Florid approach, several proposal for the combination of rule languages with ontology languages have been made. For a general survey see [ADG+05]. These approaches range from *homogeneous* approaches, in which rules and ontologies are combined and integrated seamlessly, to *hybrid* approaches which keep rules and ontologies separate. Examples of the former approach are the *Semantic Web Rules Language (SWRL)* and *Description Logic Programs (DLP)* which are explained below. Whereas, $\mathcal{AL}$-Log and Carin are examples of the latter approach (cf. [ADG+05]). They are presented afterwards.

**SWRL.** The Semantic Web Rules Language has been proposed in [HPSB+04, HPS04] as the basic rule language for the Semantic Web. It is based on a combination of OWL DL and OWL Lite with the Unary/Binary Datalog sublanguage of RuleML [The]. Basically, the *Rule Markup Language (RuleML)* is a markup language for publishing and sharing rules using a standard XML encoding. The considered Datalog sublanguage restricts a relation to be unary or binary whereas a relation in the RuleML Datalog kernel can be n-ary. Please refer to the RuleML Tutorial [BGT05] for more information on RuleML and the used sublanguage. The SWRL proposal extends the set of OWL axioms with Horn-like[43] rules which can be used to reason about OWL individuals and to infer new knowledge about the individuals. Roughly speaking, the SWRL is the *union* of Horn logic and OWL DL. The main strength of SWRL is the tight integration with OWL DL since SWRL is being defined as a syntactic and semantic extension of OWL DL. Furthermore, Horn clauses are fairly simple and understandable. At present, SWRL neither supports disjunction nor provides non-monotonic features like

[43] Horn logic consists of so called Horn clauses. A Horn clause is a clause (a disjunction of literals) with at most one positive literal. Moreover, Horn clauses form a decidable subset of FOL. Note, that in this context the terms “rules” and “clauses” are often used synonymously.

defaults or negation-as-failure. Moreover, SWRL is more expressive than OWL DL and Horn clauses separate but this increased expressiveness leads to undecidability (cf. [HPSBT05]). An implementation of SWRL has been presented in [Gol04] which is based on existing technologies such as Protégé OWL [Proa], RACER [Rac] and Jess [FH]. However, OWL and SWRL were criticized and an alternative ontology language called *OWL-Flight*, based entirely on logic programming (F-Logic), was proposed in [BPLF05]. More precisely, they propose that OWL and rules should exist side-by-side. This new approach was criticized in [HPPSH05] because the separation of OWL and rules would create two Semantic Webs with little or no semantic interoperability.

**DLP.** Description Logic Programs have been initially presented in [GHVD03]. In this proposal, Description Logic Programs are defined as a new intermediate knowledge representation which is contained within the intersection of Description Logics and *Logic Programs (LP)*. More precisely, the Horn fragment of FOL that contains no function symbols is used as the considered logic subset. Moreover, a translation from the DLP fragment of DL to LP and from the DLP fragment of LP to DL is provided. Therefore, the information can be exchanged between both fragments and enables to build rules on top of ontologies. In other words, DLP is the *intersection* of Horn logic and OWL in contrast to SWRL which bases on the union of Horn logic and OWL. However, the DLP approach denotes a rule and ontology language with very restrictive expressiveness, e.g., they are restricted to universal quantification and lack basic negation (cf. [MSS04]).[44] More information can be found in [VMHG03] and in [Vol04]. Furthermore, the description of an early implementation can be found in [VDO03].

The DLP fragment as a decidable but unexpressive proposal on the one side and the expressive yet undecidable SWRL approach on the other side mark two extremes of a wide range of possible approaches. In between, several proposals have been made to extend expressiveness while still retaining decidability (cf. [EIP+06]). One of the first approaches is $\mathcal{AL}$-Log.

**$\mathcal{AL}$-Log.** $\mathcal{AL}$-Log has been proposed in [DLNS98]. This approach extends the description logic $\mathcal{AL}$ by Horn rules which are restricted in such a way that each variable in a rule must appear in at least one non-DL-atom in the rule body. This restriction retains decidability and such rules are called *DL-safe*. However, this restriction makes rules only applicable to explicitly named objects, i.e. to known individuals. Furthermore, $\mathcal{AL}$-Log does not include negation. An extension of this approach to the more expressive DL $\mathcal{SHIQ}$ has been presented in [MSS04] which brings the whole approach closer to OWL.

[44] Note that it was possible to show for a wide range of ontologies that they are inside the used intersection (cf. [GHVD03]).

**CARIN.** CARIN has been presented in [LR96] as a family of languages intended to integrate Datalog with different Description Logics. Contrary to $\mathcal{AL}$-Log and other approaches on the basis of DL-save rules, CARIN does not restrict each variable from the rule head to appear in at least one non-DL-atoms in the rule body. Thus CARIN extends AL-Log with more expressive DL and more general rules which results in undecidability. However, decidable subset of CARIN can be obtained by either restricting rules to be non-recursive or by *role-safety*, according to which at least one variable from a literal with a role predicate must also occur in a non-DL body atom.

**OWL 1.1.** Additionally to the approaches presented before, an extended version of the Web Ontology Language is under development [W3C07a]. This new version is called *OWL 1.1* and provides more Description Logic expressiveness by moving from the $\mathcal{SHOIN}$ Description Logic of OWL DL to the more expressive $\mathcal{SROIQ}$ Description Logic [HKS06]. This DL is still decidable and allows for more expressiveness around properties. Moreover, it provides some syntactic sugar and user-defined datatypes. However, at present the OWL 1.1 specification is a W3C Member Submission and accordingly, some essential parts are not implemented yet.

The DL-Florid developed in this thesis proposes a hybrid approach. The F-Logic rules are kept separately and a subset of an ontology is exchanged between both systems. This provides greater flexibility and expressiveness than the homogeneous approaches. However, the resulting system is not decidable. But currently, no decidable systems have been presented which provide a similar expressiveness and which integrate into the existing technologies seamlessly like the DL-Florid approach.

## 8.2 Further Tasks

After the presentation of some approaches related to the DL-Florid approach, this section denotes the next step for the development of the DL-Florid prototype.

Up to now, the implemented prototype has been used separately from other frameworks. Therefore, the next step is the integration into already existing frameworks like the *MARS Framework*. MARS stands for *Modular Active Rules in the Semantic Web* and initially, this framework has been presented in [AAB+05]. The DL-Florid approach could be used, for example, to reason about rules (see [AAB+07]).

Furthermore, the prototype can be improved and extend in several ways. Possible areas of further development are:

- The change of the rule markup from the specific XML markup to more established markups would allow for interchanging rules with other frameworks.
- As mentioned before, a direct integration of Florid into DL-Florid would increase the performance significantly. At present, a Java extension to Florid is under development.
- At present, the prototype is focused on the web interface. For an integration into other framework, a more sophisticated technology, like web services, is needed.

# 9 Conclusion

In this thesis, a combination of Description Logic reasoning with F-Logic rules has been proposed. First, the basic concepts of the Semantic Web have been presented. To work with these concepts, particularly with OWL DL, the Jena Semantic Web framework has been introduced. This was followed by the presentation of Florid as an F-Logic system.

These concepts and frameworks have been utilized to develop the proposed combination on a theoretical level as well as on an implementation level in terms of a prototype. For this purpose, an exchangeable subset of OWL DL has been identified and the translation from OWL DL into F-Logic and vice versa has been given.

The use of state-of-the-art technologies (Jena, Pellet, Web Services) for the alternating reasoning process ensures that the DL-Florid system can easily be integrated into existing systems in order to utilize the reasoning capabilities of DL-Florid. The implemented web interface of the prototype has been used for some examples which show the benefits of the DL-Florid combination. For example, DL-Florid combines the abilities of OWL DL to define and to reason about class hierarchies with the power of F-Logic rules about properties. The expressiveness and the abilities of this proposal are way beyond of the functionalities provided by OWL DL. Finally some related approaches have been presented and the further steps in the development of DL-Florid have been denoted.

This thesis represents only the first step for the development of DL-Florid. Further steps will include the integration into existing frameworks as a firm foundation for reasoning, e.g. about rules.

# A Examples

## A.1 Family Genealogy

```
<?xml version="1.0"?>
<rdf:RDF xmlns="http://foo.org/dummy#"
  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#"
  xml:base="http://foo.org/dummy">
  <owl:Ontology rdf:about="">
    <owl:versionInfo
      rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
      An example ontology created by Matthew Horridge, modified by
      Heiko Kattenstroth: derived classes are used and some
      classes and properties were added
    </owl:versionInfo>
  </owl:Ontology>

  <owl:Class rdf:ID="Person" />
  <owl:Class rdf:ID="MaleSex">
    <owl:disjointWith rdf:resource="#FemaleSex" />
  </owl:Class>
  <owl:Class rdf:ID="FemaleSex" />

  <owl:Class rdf:ID="Parent">
    <owl:equivalentClass>
      <owl:Class>
        <owl:intersectionOf rdf:parseType="Collection">
          <owl:Class rdf:about="#Person" />
          <owl:Restriction>
            <owl:someValuesFrom rdf:resource="#Person" />
            <owl:onProperty>
              <owl:ObjectProperty rdf:ID="hasChild" />
            </owl:onProperty>
          </owl:Restriction>
        </owl:intersectionOf>
      </owl:Class>
    </owl:equivalentClass>
  </owl:Class>

  <owl:Class rdf:ID="Father">
    <owl:equivalentClass>
      <owl:Class>
        <owl:intersectionOf rdf:parseType="Collection">
          <owl:Class rdf:about="#Parent" />
          <owl:Class rdf:about="#Man" />
```

```
        </owl:intersectionOf>
      </owl:Class>
    </owl:equivalentClass>
  </owl:Class>

  <owl:Class rdf:ID="Mother">
    <owl:equivalentClass>
      <owl:Class>
        <owl:intersectionOf rdf:parseType="Collection">
          <owl:Class rdf:about="#Parent" />
          <owl:Class rdf:about="#Woman" />
        </owl:intersectionOf>
      </owl:Class>
    </owl:equivalentClass>
  </owl:Class>

  <owl:Class rdf:ID="GrandParent">
    <owl:equivalentClass>
      <owl:Class>
        <owl:intersectionOf rdf:parseType="Collection">
          <owl:Class rdf:about="#Person" />
          <owl:Restriction>
            <owl:onProperty>
              <owl:ObjectProperty rdf:about="#hasChild" />
            </owl:onProperty>
            <owl:someValuesFrom rdf:resource="#Parent" />
          </owl:Restriction>
        </owl:intersectionOf>
      </owl:Class>
    </owl:equivalentClass>
  </owl:Class>
<!--
  <owl:Class rdf:ID="GrandFather">
    <owl:equivalentClass>
      <owl:Class>
        <owl:intersectionOf rdf:parseType="Collection">
          <owl:Class rdf:about="#GrandParent" />
          <owl:Class rdf:about="#Man" />
        </owl:intersectionOf>
      </owl:Class>
    </owl:equivalentClass>
  </owl:Class>

  <owl:Class rdf:ID="GrandMother">
    <owl:equivalentClass>
      <owl:Class>
        <owl:intersectionOf rdf:parseType="Collection">
          <owl:Class rdf:about="#GrandParent" />
          <owl:Class rdf:about="#Woman" />
        </owl:intersectionOf>
      </owl:Class>
    </owl:equivalentClass>
  </owl:Class>
-->
  <owl:Class rdf:ID="Man">
    <owl:equivalentClass>
      <owl:Class>
```

```
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Person" />
        <owl:Restriction>
          <owl:onProperty>
            <owl:FunctionalProperty rdf:about="#hasSex" />
          </owl:onProperty>
          <owl:hasValue rdf:resource="#MaleSex" />
        </owl:Restriction>
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
  <rdfs:subClassOf>
    <owl:Class rdf:about="#Person" />
  </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="Woman">
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Person" />
        <owl:Restriction>
          <owl:onProperty>
            <owl:FunctionalProperty rdf:about="#hasSex" />
          </owl:onProperty>
          <owl:hasValue rdf:resource="#FemaleSex" />
        </owl:Restriction>
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
  <rdfs:subClassOf>
    <owl:Class rdf:about="#Person" />
  </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="OffSpring">
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Person" />
        <owl:Restriction>
          <owl:someValuesFrom rdf:resource="#Person" />
          <owl:onProperty>
            <owl:ObjectProperty rdf:about="#hasParent" />
          </owl:onProperty>
        </owl:Restriction>
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>

<owl:Class rdf:ID="Daughter">
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#OffSpring" />
        <owl:Class rdf:about="#Woman" />
```

```
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>

<owl:Class rdf:ID="Son">
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#OffSpring" />
        <owl:Class rdf:about="#Man" />
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>

<owl:Class rdf:ID="Sex">
  <owl:equivalentClass>
    <owl:Class>
      <owl:oneOf rdf:parseType="Collection">
        <owl:Class rdf:about="#MaleSex" />
        <owl:Class rdf:about="#FemaleSex" />
      </owl:oneOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>

<owl:Class rdf:ID="Sibling">
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Person" />
        <owl:Restriction>
          <owl:someValuesFrom rdf:resource="#Person" />
          <owl:onProperty>
            <owl:SymmetricProperty
              rdf:about="#hasSibling" />
          </owl:onProperty>
        </owl:Restriction>
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>

<owl:Class rdf:ID="Brother">
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Man" />
        <owl:Restriction>
          <owl:someValuesFrom rdf:resource="#Person" />
          <owl:onProperty>
            <owl:SymmetricProperty
              rdf:about="#hasSibling" />
          </owl:onProperty>
        </owl:Restriction>
      </owl:intersectionOf>
```

```
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>

<owl:Class rdf:ID="Sister">
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Woman" />
        <owl:Restriction>
          <owl:someValuesFrom rdf:resource="#Person" />
          <owl:onProperty>
            <owl:SymmetricProperty rdf:ID="hasSibling" />
          </owl:onProperty>
        </owl:Restriction>
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>

<owl:Class rdf:ID="Uncle">
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Man" />
        <owl:Class>
          <owl:unionOf rdf:parseType="Collection">
            <owl:Restriction>
              <owl:someValuesFrom
                rdf:resource="#Person" />
              <owl:onProperty>
                <owl:ObjectProperty
                  rdf:about="#hasNephew" />
              </owl:onProperty>
            </owl:Restriction>
            <owl:Restriction>
              <owl:someValuesFrom
                rdf:resource="#Person" />
              <owl:onProperty>
                <owl:ObjectProperty
                  rdf:about="#hasNiece" />
              </owl:onProperty>
            </owl:Restriction>
          </owl:unionOf>
        </owl:Class>
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>

<owl:Class rdf:ID="Aunt">
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Woman" />
        <owl:Class>
          <owl:unionOf rdf:parseType="Collection">
```

```
            <owl:Restriction>
              <owl:someValuesFrom
                rdf:resource="#Person" />
              <owl:onProperty>
                <owl:ObjectProperty
                  rdf:about="#hasNephew" />
              </owl:onProperty>
            </owl:Restriction>
            <owl:Restriction>
              <owl:someValuesFrom
                rdf:resource="#Person" />
              <owl:onProperty>
                <owl:ObjectProperty
                  rdf:about="#hasNiece" />
              </owl:onProperty>
            </owl:Restriction>
          </owl:unionOf>
        </owl:Class>
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>

<owl:Class rdf:ID="Niece">
  <owl:disjointWith rdf:resource="#Nephew" />
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class>
          <owl:unionOf rdf:parseType="Collection">
            <owl:Restriction>
              <owl:someValuesFrom
                rdf:resource="#Person" />
              <owl:onProperty>
                <owl:ObjectProperty
                  rdf:about="#hasUncle" />
              </owl:onProperty>
            </owl:Restriction>
            <owl:Restriction>
              <owl:onProperty>
                <owl:ObjectProperty
                  rdf:about="#hasAunt" />
              </owl:onProperty>
              <owl:someValuesFrom
                rdf:resource="#Person" />
            </owl:Restriction>
          </owl:unionOf>
        </owl:Class>
        <owl:Class rdf:about="#Woman" />
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>

<owl:Class rdf:ID="Nephew">
  <owl:equivalentClass>
    <owl:Class>
```

```
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class>
          <owl:unionOf rdf:parseType="Collection">
            <owl:Restriction>
              <owl:onProperty>
                <owl:ObjectProperty
                  rdf:ID="hasUncle" />
              </owl:onProperty>
              <owl:someValuesFrom
                rdf:resource="#Person" />
            </owl:Restriction>
            <owl:Restriction>
              <owl:onProperty>
                <owl:ObjectProperty
                  rdf:ID="hasAunt" />
              </owl:onProperty>
              <owl:someValuesFrom
                rdf:resource="#Person" />
            </owl:Restriction>
          </owl:unionOf>
        </owl:Class>
        <owl:Class rdf:about="#Man" />
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
  <owl:disjointWith>
    <owl:Class rdf:about="#Niece" />
  </owl:disjointWith>
</owl:Class>

<owl:Class rdf:ID="Ancestor">
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Person" />
        <owl:Restriction>
          <owl:someValuesFrom rdf:resource="#Person" />
          <owl:onProperty>
            <owl:ObjectProperty rdf:about="#hasDescendant" />
          </owl:onProperty>
        </owl:Restriction>
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>

<owl:Class rdf:ID="Descendant">
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Person" />
        <owl:Restriction>
          <owl:someValuesFrom rdf:resource="#Person" />
          <owl:onProperty>
            <owl:ObjectProperty rdf:about="#hasAncestor" />
          </owl:onProperty>
        </owl:Restriction>
```

```
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>

<!-- Properties -->

<owl:TransitiveProperty rdf:ID="hasAncestor">
  <rdfs:domain rdf:resource="#Person" />
  <rdfs:range rdf:resource="#Person" />
</owl:TransitiveProperty>

<owl:TransitiveProperty rdf:ID="hasDescendant">
  <rdfs:domain rdf:resource="#Person" />
  <rdfs:range rdf:resource="#Person" />
</owl:TransitiveProperty>


<owl:ObjectProperty rdf:about="#hasNephew">
  <rdfs:domain rdf:resource="#Person" />
  <rdfs:range rdf:resource="#Man" />
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasNiece">
  <rdfs:domain rdf:resource="#Person" />
  <rdfs:range rdf:resource="#Woman" />
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasAunt">
  <rdfs:domain rdf:resource="#Person" />
  <rdfs:range rdf:resource="#Woman" />
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasUncle">
  <rdfs:domain rdf:resource="#Person" />
  <rdfs:range rdf:resource="#Man" />
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasParent">
  <rdfs:subPropertyOf rdf:resource="#hasAncestor" />
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="#hasChild" />
  </owl:inverseOf>
  <rdfs:domain rdf:resource="#Person" />
  <rdfs:range rdf:resource="#Person" />
</owl:ObjectProperty>

<owl:FunctionalProperty rdf:about="#hasFather">
  <rdf:type
    rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty" />
  <rdfs:subPropertyOf rdf:resource="#hasParent" />
  <rdfs:domain rdf:resource="#Person" />
  <rdfs:range rdf:resource="#Man" />
</owl:FunctionalProperty>

<owl:FunctionalProperty rdf:about="#hasMother">
  <rdfs:subPropertyOf rdf:resource="#hasParent" />
```

```
  <rdf:type
    rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty" />
  <rdfs:domain rdf:resource="#Person" />
  <rdfs:range rdf:resource="#Woman" />
</owl:FunctionalProperty>

<owl:ObjectProperty rdf:about="#hasSibling">
  <rdf:type
    rdf:resource="http://www.w3.org/2002/07/owl#SymmetricProperty" />
  <rdfs:domain rdf:resource="#Person" />
  <rdfs:range rdf:resource="#Person" />
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasBrother">
  <rdfs:subPropertyOf rdf:resource="#hasSibling" />
  <rdfs:domain rdf:resource="#Person" />
  <rdfs:range rdf:resource="#Man" />
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasSister">
  <rdfs:subPropertyOf rdf:resource="#hasSibling" />
  <rdfs:range rdf:resource="#Woman" />
  <rdfs:domain rdf:resource="#Person" />
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasChild">
  <rdfs:subPropertyOf rdf:resource="#hasDescendant" />
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="#hasParent" />
  </owl:inverseOf>
  <rdfs:domain rdf:resource="#Person" />
  <rdfs:range rdf:resource="#Person" />
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasSon">
  <rdfs:subPropertyOf rdf:resource="#hasChild" />
  <rdfs:range rdf:resource="#Man" />
  <rdfs:domain rdf:resource="#Person" />
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasDaughter">
  <rdfs:subPropertyOf rdf:resource="#hasChild" />
  <rdfs:domain rdf:resource="#Person" />
  <rdfs:range rdf:resource="#Woman" />
</owl:ObjectProperty>

<owl:FunctionalProperty rdf:about="#hasSex">
  <rdfs:range rdf:resource="#Sex" />
  <rdf:type
    rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty" />
</owl:FunctionalProperty>

<!-- Individuals -->

<!-- first generation -->
<Person rdf:ID="Agnes">
  <hasSex rdf:resource="#FemaleSex" />
```

```
  <hasSibling rdf:resource="#Agatha" />
  <hasSibling rdf:resource="#Alexandra" />
</Person>

<Person rdf:ID="Abel">
  <hasSex rdf:resource="#MaleSex" />
</Person>

<Person rdf:ID="Agatha">
  <hasSex rdf:resource="#FemaleSex" />
</Person>
<Person rdf:ID="Alexandra">
  <hasSex rdf:resource="#FemaleSex" />
</Person>

<!-- second generation !-->
<Person rdf:ID="Becky">
  <hasSex rdf:resource="#FemaleSex" />
  <hasParent rdf:resource="#Abel" />
  <hasParent rdf:resource="#Agnes" />
  <hasSibling rdf:resource="#Benedict" />
</Person>

<Person rdf:ID="Benedict">
  <hasSex rdf:resource="#MaleSex" />
  <hasParent rdf:resource="#Abel" />
  <hasParent rdf:resource="#Agnes" />
</Person>

<Person rdf:ID="Benjamin">
  <hasSex rdf:resource="#MaleSex" />
</Person>

<!-- third generation !-->
<Person rdf:ID="Charlene">
  <hasSex rdf:resource="#FemaleSex" />
  <hasParent rdf:resource="#Becky" />
  <hasParent rdf:resource="#Benjamin" />
  <hasSibling rdf:resource="#Chelsea" />
</Person>

<Person rdf:ID="Chelsea">
  <hasSex rdf:resource="#FemaleSex" />
  <hasParent rdf:resource="#Becky" />
  <hasParent rdf:resource="#Benjamin" />
</Person>

<!-- fourth generation -->

<Person rdf:ID="Deby">
  <hasSex rdf:resource="#FemaleSex" />
  <hasParent rdf:resource="#Charles" />
  <hasParent rdf:resource="#Charlene" />
  <hasSibling rdf:resource="#Daniel" />
</Person>

<Person rdf:ID="Daniel">
```

```
    <hasSex rdf:resource="#MaleSex" />
    <hasParent rdf:resource="#Charles" />
    <hasParent rdf:resource="#Charlene" />
  </Person>

</rdf:RDF>
```

**Example A.1:** Genealogy Ontology

```
% siblings
X["http://foo.org/dummy#hasSister" ->>Y]      :- Y:"http://foo.org/dummy#Woman
    "["http://foo.org/dummy#hasSibling"->>X] .
X["http://foo.org/dummy#hasBrother" ->>Y]     :- Y:"http://foo.org/dummy#Man"[
    "http://foo.org/dummy#hasSibling"->>X] .

% father & mother
X["http://foo.org/dummy#hasMother" ->>Y]      :- Y:"http://foo.org/dummy#Woman
    "["http://foo.org/dummy#hasChild"->>X] .
X["http://foo.org/dummy#hasFather" ->>Y]      :- Y:"http://foo.org/dummy#Man"[
    "http://foo.org/dummy#hasChild"->>X] .

% uncle & aunt
X["http://foo.org/dummy#hasAunt" ->>Y]        :- X["http://foo.org/dummy#
    hasParent"->>Z] , Z["http://foo.org/dummy#hasSister"->>Y] .
X["http://foo.org/dummy#hasUncle" ->>Y]       :- X["http://foo.org/dummy#
    hasParent"->>Z] , Z["http://foo.org/dummy#hasBrother"->>Y] .

% niece & nephew
X["http://foo.org/dummy#hasNiece" ->>Y]       :- Y:"http://foo.org/dummy#Woman
    "["http://foo.org/dummy#hasAunt"->>X] .
X["http://foo.org/dummy#hasNiece" ->>Y]       :- Y:"http://foo.org/dummy#Woman
    "["http://foo.org/dummy#hasUncle"->>X] .
X["http://foo.org/dummy#hasNephew" ->>Y]      :- Y:"http://foo.org/dummy#Man"[
    "http://foo.org/dummy#hasAunt"->>X] .
X["http://foo.org/dummy#hasNephew" ->>Y]      :- Y:"http://foo.org/dummy#Man"[
    "http://foo.org/dummy#hasUncle"->>X] .
```

**Example A.2:** Genealogy Rules

## A.2 Train Connections

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!DOCTYPE rdf:RDF [
    <!ENTITY owl   "http://www.w3.org/2002/07/owl#">
    <!ENTITY xsd   "http://www.w3.org/2001/XMLSchema#">
]>
<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:owl= "http://www.w3.org/2002/07/owl#"
  xmlns:bahn ="http://localhost/test.rdf#"
  xml:base   ="http://localhost/test.rdf#"
  xmlns      ="http://localhost/test.rdf#"
>
  <owl:Class rdf:ID="Schedule"/>
```

```
<owl:Class rdf:ID="Connection"/>
<owl:Class rdf:ID="Section" />
<owl:Class rdf:ID="Station"/>
<owl:Class rdf:ID="Train"/>

<owl:ObjectProperty rdf:ID="has_Schedule">
  <rdfs:domain rdf:resource="#Train"/>
  <rdfs:range rdf:resource="#Schedule"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasSection">
  <rdf:type rdf:resource="&owl;TransitiveProperty"/>
  <rdfs:domain rdf:resource="#Schedule"/>
  <rdfs:range rdf:resource="#Section"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="routePoint"/>
<owl:ObjectProperty rdf:ID="startPoint">
  <rdfs:subPropertyOf rdf:resource="#routePoint"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="endPoint">
  <rdfs:subPropertyOf rdf:resource="#routePoint"/>
</owl:ObjectProperty>


<owl:ObjectProperty rdf:ID="Trainnumber">
  <rdfs:domain rdf:resource="#Train"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="departure_place">
  <rdfs:domain rdf:resource="#Train"/>
  <rdfs:range rdf:resource="#Station"/>
</owl:ObjectProperty>

<owl:DatatypeProperty rdf:ID="departure_time">
  <rdfs:domain rdf:resource="#Train"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="distance">
  <rdfs:domain rdf:resource="#Section"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="duration">
  <rdfs:domain rdf:resource="#Section"/>
</owl:DatatypeProperty>


<bahn:Train rdf:ID="ICE680">
  <bahn:has_Schedule rdf:resource="#routemap_M-HH"/>
  <bahn:departure_time>0855</bahn:departure_time>
  <bahn:departure_place rdf:resource="#Munich_Hbf"/>
</bahn:Train>

<bahn:Train rdf:ID="ICE108">
  <bahn:has_Schedule rdf:resource="#routemap_S-B"/>
  <bahn:departure_time>1251</bahn:departure_time>
```

```
  <bahn:departure_place rdf:resource="#Stuttgart"/>
</bahn:Train>

<bahn:Train rdf:ID="ICE1516">
  <bahn:has_Schedule rdf:resource="#routemap_B-HH"/>
  <bahn:departure_time>1930</bahn:departure_time>
  <bahn:departure_place rdf:resource="#Berlin_Ostbahnhof"/>
</bahn:Train>


<rdf:Description rdf:about="#routemap_M-HH">
  <rdf:type rdf:resource="#Schedule"/>
  <bahn:hasSection rdf:resource="#Munich_Hbf-Munich_Pasing"/>
  <bahn:hasSection rdf:resource="#Munich_Pasing-Augsburg"/>
  <bahn:hasSection rdf:resource="#Augsburg-Wuerzburg"/>
  <bahn:hasSection rdf:resource="#Wuerzburg-Fulda"/>
  <bahn:hasSection rdf:resource="#Fulda-Kassel_Wilhelmshoehe"/>
  <bahn:hasSection rdf:resource="#Kassel_Wilhelmshoehe-Goettingen"/>
  <bahn:hasSection rdf:resource="#Goettingen-Hannover"/>
  <bahn:hasSection rdf:resource="#Hannover-Hamburg_Harburg"/>
  <bahn:hasSection rdf:resource="#Hamburg_Harburg-Hamburg_Hbf"/>
  <bahn:hasSection rdf:resource="#Hamburg_Hbf-Hamburg_Dammtor"/>
  <bahn:hasSection rdf:resource="#Hamburg_Dammtor-Hamburg_Altona"/>
</rdf:Description>

<rdf:Description rdf:about="#routemap_S-B">
  <rdf:type rdf:resource="#Schedule"/>
  <bahn:hasSection rdf:resource="#Stuttgart-Mannheim"/>
  <bahn:hasSection rdf:resource="#Mannheim-Frankfurt_Main_Hbf"/>
  <bahn:hasSection rdf:resource="#Frankfurt_Main_Hbf-Hanau"/>
  <bahn:hasSection rdf:resource="#Hanau-Fulda"/>
  <bahn:hasSection rdf:resource="#Fulda-Kassel_Wilhelmshoehe"/>
  <bahn:hasSection rdf:resource="#Kassel_Wilhelmshoehe-Goettingen"/>
  <bahn:hasSection rdf:resource="#Goettingen-Hildesheim"/>
  <bahn:hasSection rdf:resource="#Hildesheim-Braunschweig"/>
  <bahn:hasSection rdf:resource="#Braunschweig-Berlin_Spandau"/>
  <bahn:hasSection rdf:resource="#Berlin_Spandau-Berlin_ZoologischerGarten"
      />
  <bahn:hasSection rdf:resource="#Berlin_ZoologischerGarten-
      Berlin_Ostbahnhof"/>
</rdf:Description>

<rdf:Description rdf:about="#routemap_B-HH">
  <rdf:type rdf:resource="#Schedule"/>
  <bahn:hasSection rdf:resource="#Berlin_Ostbahnhof-
      Berlin_ZoologischerGarten"/>
  <bahn:hasSection rdf:resource="#Berlin_ZoologischerGarten-Hamburg_Hbf"/>
  <bahn:hasSection rdf:resource="#Hamburg_Hbf-Hamburg_Dammtor"/>
  <bahn:hasSection rdf:resource="#Hamburg_Dammtor-Hamburg_Altona"/>
</rdf:Description>

<bahn:Station rdf:ID="Munich_Hbf"/>
<bahn:Station rdf:ID="Munich_Pasing"/>
<bahn:Station rdf:ID="Augsburg"/>
<bahn:Station rdf:ID="Wuerzburg"/>
<bahn:Station rdf:ID="Fulda"/>
<bahn:Station rdf:ID="Kassel_Wilhelmshoehe"/>
```

```
<bahn:Station rdf:ID="Goettingen"/>
<bahn:Station rdf:ID="Hannover"/>
<bahn:Station rdf:ID="Hamburg_Harburg"/>
<bahn:Station rdf:ID="Hamburg_Hbf"/>
<bahn:Station rdf:ID="Hamburg_Dammtor"/>
<bahn:Station rdf:ID="Hamburg_Altona"/>

<bahn:Station rdf:ID="Stuttgart"/>
<bahn:Station rdf:ID="Mannheim"/>
<bahn:Station rdf:ID="Frankfurt_Main_Hbf"/>
<bahn:Station rdf:ID="Hanau"/>
<bahn:Station rdf:ID="Hildesheim"/>
<bahn:Station rdf:ID="Braunschweig"/>
<bahn:Station rdf:ID="Berlin_Spandau"/>
<bahn:Station rdf:ID="Berlin_ZoologischerGarten"/>
<bahn:Station rdf:ID="Berlin_Ostbahnhof"/>

<bahn:Section rdf:about="#Munich_Hbf-Munich_Pasing">
  <bahn:startPoint rdf:resource="#Munich_Hbf"/>
  <bahn:endPoint rdf:resource="#Munich_Pasing"/>
  <bahn:distance rdf:datatype="&xsd;integer">7</bahn:distance>
  <bahn:duration rdf:datatype="&xsd;integer">8</bahn:duration>
</bahn:Section>
<bahn:Section rdf:about="#Munich_Pasing-Augsburg">
  <bahn:startPoint rdf:resource="#Munich_Pasing"/>
  <bahn:endPoint rdf:resource="#Augsburg"/>
  <bahn:distance rdf:datatype="&xsd;integer">55</bahn:distance>
  <bahn:duration rdf:datatype="&xsd;integer">31</bahn:duration>
</bahn:Section>
<bahn:Section rdf:about="#Augsburg-Wuerzburg">
  <bahn:startPoint rdf:resource="#Augsburg"/>
  <bahn:endPoint rdf:resource="#Wuerzburg"/>
  <bahn:distance rdf:datatype="&xsd;integer">215</bahn:distance>
  <bahn:duration rdf:datatype="&xsd;integer">112</bahn:duration>
</bahn:Section>
<bahn:Section rdf:about="#Wuerzburg-Fulda">
  <bahn:startPoint rdf:resource="#Wuerzburg"/>
  <bahn:endPoint rdf:resource="#Fulda"/>
  <bahn:distance rdf:datatype="&xsd;integer">92</bahn:distance>
  <bahn:duration rdf:datatype="&xsd;integer">35</bahn:duration>
</bahn:Section>
<bahn:Section rdf:about="#Fulda-Kassel_Wilhelmshoehe">
   <bahn:startPoint rdf:resource="#Fulda"/>
  <bahn:endPoint rdf:resource="#Kassel_Wilhelmshoehe"/>
  <bahn:distance rdf:datatype="&xsd;integer">90</bahn:distance>
  <bahn:duration rdf:datatype="&xsd;integer">32</bahn:duration>
</bahn:Section>
<bahn:Section rdf:about="#Kassel_Wilhelmshoehe-Goettingen">
  <bahn:startPoint rdf:resource="#Kassel_Wilhelmshoehe"/>
  <bahn:endPoint rdf:resource="#Goettingen"/>
  <bahn:distance rdf:datatype="&xsd;integer">44</bahn:distance>
  <bahn:duration rdf:datatype="&xsd;integer">21</bahn:duration>
</bahn:Section>
<bahn:Section rdf:about="#Goettingen-Hannover">
  <bahn:startPoint rdf:resource="#Goettingen"/>
  <bahn:endPoint rdf:resource="#Hannover"/>
  <bahn:distance rdf:datatype="&xsd;integer">99</bahn:distance>
```

```
  <bahn:duration rdf:datatype="&xsd;integer">40</bahn:duration>
</bahn:Section>
<bahn:Section rdf:about="#Hannover-Hamburg_Harburg">
  <bahn:startPoint rdf:resource="#Hannover"/>
  <bahn:endPoint rdf:resource="#Hamburg_Harburg"/>
  <bahn:distance rdf:datatype="&xsd;integer">167</bahn:distance>
  <bahn:duration rdf:datatype="&xsd;integer">67</bahn:duration>
</bahn:Section>

<bahn:Section rdf:about="#Hamburg_Harburg-Hamburg_Hbf">
  <bahn:startPoint rdf:resource="#Hamburg_Harburg"/>
  <bahn:endPoint rdf:resource="#Hamburg_Hbf"/>
  <bahn:distance rdf:datatype="&xsd;integer">12</bahn:distance>
  <bahn:duration rdf:datatype="&xsd;integer">11</bahn:duration>
</bahn:Section>
<bahn:Section rdf:about="#Hamburg_Hbf-Hamburg_Dammtor">
  <bahn:startPoint rdf:resource="#Hamburg_Hbf"/>
  <bahn:endPoint rdf:resource="#Hamburg_Dammtor"/>
  <bahn:distance rdf:datatype="&xsd;integer">1</bahn:distance>
  <bahn:duration rdf:datatype="&xsd;integer">7</bahn:duration>
</bahn:Section>
<bahn:Section rdf:about="#Hamburg_Dammtor-Hamburg_Altona">
  <bahn:startPoint rdf:resource="#Hamburg_Dammtor"/>
  <bahn:endPoint rdf:resource="#Hamburg_Altona"/>
  <bahn:distance rdf:datatype="&xsd;integer">5</bahn:distance>
  <bahn:duration rdf:datatype="&xsd;integer">8</bahn:duration>
</bahn:Section>


<bahn:Section rdf:about="#Stuttgart-Mannheim">
  <bahn:startPoint rdf:resource="#Stuttgart"/>
  <bahn:endPoint rdf:resource="#Mannheim"/>
  <bahn:distance rdf:datatype="&xsd;integer">0</bahn:distance>
  <bahn:duration rdf:datatype="&xsd;integer">40</bahn:duration>
</bahn:Section>
<bahn:Section rdf:about="#Mannheim-Frankfurt_Main_Hbf">
  <bahn:startPoint rdf:resource="#Mannheim"/>
  <bahn:endPoint rdf:resource="#Frankfurt_Main_Hbf"/>
  <bahn:distance rdf:datatype="&xsd;integer">0</bahn:distance>
  <bahn:duration rdf:datatype="&xsd;integer">42</bahn:duration>
</bahn:Section>
<bahn:Section rdf:about="#Frankfurt_Main_Hbf-Hanau">
  <bahn:startPoint rdf:resource="#Frankfurt_Main_Hbf"/>
  <bahn:endPoint rdf:resource="#Hanau"/>
  <bahn:distance rdf:datatype="&xsd;integer">0</bahn:distance>
  <bahn:duration rdf:datatype="&xsd;integer">17</bahn:duration>
</bahn:Section>
<bahn:Section rdf:about="#Hanau-Fulda">
  <bahn:startPoint rdf:resource="#Hanau"/>
  <bahn:endPoint rdf:resource="#Fulda"/>
  <bahn:distance rdf:datatype="&xsd;integer">0</bahn:distance>
  <bahn:duration rdf:datatype="&xsd;integer">42</bahn:duration>
</bahn:Section>

<bahn:Section rdf:about="#Goettingen-Hildesheim">
  <bahn:startPoint rdf:resource="#Goettingen"/>
  <bahn:endPoint rdf:resource="#Hildesheim"/>
```

```
  <bahn:distance rdf:datatype="&xsd;integer">0</bahn:distance>
  <bahn:duration rdf:datatype="&xsd;integer">30</bahn:duration>
</bahn:Section>
<bahn:Section rdf:about="#Hildesheim-Braunschweig">
  <bahn:startPoint rdf:resource="#Hildesheim"/>
  <bahn:endPoint rdf:resource="#Braunschweig"/>
  <bahn:distance rdf:datatype="&xsd;integer">0</bahn:distance>
  <bahn:duration rdf:datatype="&xsd;integer">26</bahn:duration>
</bahn:Section>
<bahn:Section rdf:about="#Braunschweig-Berlin_Spandau">
  <bahn:startPoint rdf:resource="#Braunschweig"/>
  <bahn:endPoint rdf:resource="#Berlin_Spandau"/>
  <bahn:distance rdf:datatype="&xsd;integer">0</bahn:distance>
  <bahn:duration rdf:datatype="&xsd;integer">65</bahn:duration>
</bahn:Section>
<bahn:Section rdf:about="#Berlin_Spandau-Berlin_ZoologischerGarten">
  <bahn:startPoint rdf:resource="#Berlin_Spandau"/>
  <bahn:endPoint rdf:resource="#Berlin_ZoologischerGarten"/>
  <bahn:distance rdf:datatype="&xsd;integer">0</bahn:distance>
  <bahn:duration rdf:datatype="&xsd;integer">11</bahn:duration>
</bahn:Section>

<bahn:Section rdf:about="#Berlin_ZoologischerGarten-Berlin_Ostbahnhof">
  <bahn:startPoint rdf:resource="#Berlin_ZoologischerGarten"/>
  <bahn:endPoint rdf:resource="#Berlin_Ostbahnhof"/>
  <bahn:distance rdf:datatype="&xsd;integer">0</bahn:distance>
  <bahn:duration rdf:datatype="&xsd;integer">18</bahn:duration>
</bahn:Section>


<bahn:Section rdf:about="#Berlin_Ostbahnhof-Berlin_ZoologischerGarten">
  <bahn:startPoint rdf:resource="#Berlin_Ostbahnhof"/>
  <bahn:endPoint rdf:resource="#Berlin_ZoologischerGarten"/>
  <bahn:distance rdf:datatype="&xsd;integer">0</bahn:distance>
  <bahn:duration rdf:datatype="&xsd;integer">15</bahn:duration>
</bahn:Section>

<bahn:Section rdf:about="#Berlin_ZoologischerGarten-Hamburg_Hbf">
  <bahn:startPoint rdf:resource="#Berlin_ZoologischerGarten"/>
  <bahn:endPoint rdf:resource="#Hamburg_Hbf"/>
  <bahn:distance rdf:datatype="&xsd;integer">0</bahn:distance>
  <bahn:duration rdf:datatype="&xsd;integer">90</bahn:duration>
</bahn:Section>

<bahn:Section rdf:about="#Hamburg_Altona-Hamburg_Dammtor">
  <bahn:startPoint rdf:resource="#Hamburg_Altona"/>
  <bahn:endPoint rdf:resource="#Hamburg_Dammtor"/>
  <bahn:distance rdf:datatype="&xsd;integer">5</bahn:distance>
  <bahn:duration rdf:datatype="&xsd;integer">8</bahn:duration>
</bahn:Section>

<bahn:Section rdf:about="#Hamburg_Dammtor-Hamburg_Hbf">
  <bahn:startPoint rdf:resource="#Hamburg_Dammtor"/>
  <bahn:endPoint rdf:resource="#Hamburg_Hbf"/>
  <bahn:distance rdf:datatype="&xsd;integer">1</bahn:distance>
  <bahn:duration rdf:datatype="&xsd;integer">7</bahn:duration>
</bahn:Section>
```

```
<bahn:Section rdf:about="#Hamburg_Hbf-Hamburg_Harburg">
  <bahn:startPoint rdf:resource="#Hamburg_Hbf"/>
  <bahn:endPoint rdf:resource="#Hamburg_Harburg"/>
  <bahn:distance rdf:datatype="&xsd;integer">12</bahn:distance>
  <bahn:duration rdf:datatype="&xsd;integer">11</bahn:duration>
</bahn:Section>

<bahn:Section rdf:about="#Hamburg_Harburg-Hannover">
  <bahn:startPoint rdf:resource="#Hamburg_Harburg"/>
  <bahn:endPoint rdf:resource="#Hannover"/>
  <bahn:distance rdf:datatype="&xsd;integer">167</bahn:distance>
  <bahn:duration rdf:datatype="&xsd;integer">67</bahn:duration>
</bahn:Section>

<bahn:Section rdf:about="#Hannover-Goettingen">
  <bahn:startPoint rdf:resource="#Hannover"/>
  <bahn:endPoint rdf:resource="#Goettingen"/>
  <bahn:distance rdf:datatype="&xsd;integer">99</bahn:distance>
  <bahn:duration rdf:datatype="&xsd;integer">40</bahn:duration>
</bahn:Section>

<bahn:Section rdf:about="#Goettingen-Kassel_Wilhelmshoehe">
  <bahn:startPoint rdf:resource="#Goettingen"/>
  <bahn:endPoint rdf:resource="#Kassel_Wilhelmshoehe"/>
  <bahn:distance rdf:datatype="&xsd;integer">44</bahn:distance>
  <bahn:duration rdf:datatype="&xsd;integer">21</bahn:duration>
</bahn:Section>

<bahn:Section rdf:about="#Kassel_Wilhelmshoehe-Fulda">
  <bahn:startPoint rdf:resource="#Kassel_Wilhelmshoehe"/>
   <bahn:endPoint rdf:resource="#Fulda"/>
  <bahn:distance rdf:datatype="&xsd;integer">90</bahn:distance>
  <bahn:duration rdf:datatype="&xsd;integer">32</bahn:duration>
</bahn:Section>

<bahn:Section rdf:about="#Fulda-Wuerzburg">
  <bahn:startPoint rdf:resource="#Fulda"/>
  <bahn:endPoint rdf:resource="#Wuerzburg"/>
  <bahn:distance rdf:datatype="&xsd;integer">92</bahn:distance>
  <bahn:duration rdf:datatype="&xsd;integer">35</bahn:duration>
</bahn:Section>

<bahn:Section rdf:about="#Wuerzburg-Augsburg">
  <bahn:startPoint rdf:resource="#Wuerzburg"/>
  <bahn:endPoint rdf:resource="#Augsburg"/>
  <bahn:distance rdf:datatype="&xsd;integer">44</bahn:distance>
  <bahn:duration rdf:datatype="&xsd;integer">112</bahn:duration>
</bahn:Section>

<bahn:Section rdf:about="#Augsburg-Munich_Pasing">
  <bahn:startPoint rdf:resource="#Augsburg"/>
  <bahn:endPoint rdf:resource="#Munich_Pasing"/>
  <bahn:distance rdf:datatype="&xsd;integer">55</bahn:distance>
  <bahn:duration rdf:datatype="&xsd;integer">31</bahn:duration>
</bahn:Section>
```

```
  <bahn:Section rdf:about="#Munich_Pasing-Munich_Hbf">
    <bahn:startPoint rdf:resource="#Munich_Pasing"/>
    <bahn:endPoint rdf:resource="#Munich_Hbf"/>
    <bahn:distance rdf:datatype="&xsd;integer">7</bahn:distance>
    <bahn:duration rdf:datatype="&xsd;integer">8</bahn:duration>
  </bahn:Section>

</rdf:RDF>
```

**Example A.3:** Train Connections

# B Documentation

The prototype consists of two parts:

1. `Florid Server`: The Florid part consists of the Florid system itself and the web service. It is available on request from [Dat]. See Section B.1 for installation instructions.
2. `DLFLorid.war`: This is the Java based part, using servlets, the Jena Framework and Pellet. See Section B.2 for installation instructions.

## B.1 Installation of the Florid Web Service:

1. Unpack the `FloridServer.zip` to an appropriate folder, for example

   `/usr/local/share/FloridServer`

2. Compile the server (gcc4.x needed):

   ```
   make all
   ```

   Refer to the Makefile for further options.

3. Set the following paths (e.g. by using exports in the `.bashrc`):

   ```
   export DEFAULTCFG="/path/to/FloridServer/environment/config.flp"
   export DEFAULTHIS="/path/to/FloridServer/environment/default.his"
   ```

4. Start the server:

   ```
   /path/to/FloridServer/bin/floridServer 127.0.0.1 8900
   ```

   Both, the address and the port can be changed.

5. Now, the server should be ready and the following response should be displayed:

   ```
   Socket connection successful: ...
   ```

## B.2 Installation of DL-Florid

As already mentioned, the prototypical implementation of DL-Florid uses mainly the Java Servlet technology. Therefore, a Java Runtime Environment and a Java Servlet Container

are required. Please make sure, that the `$CATALINA_HOME` path and `$JAVA_HOME` path are set correctly. Furthermore, DL-Florid uses a relational database, e.g., PostgreSQL, for persistence. The following summarized requirements have to be fulfilled:

**Requirements:**

- Apache Tomcat[45] (tested with version 5.5.18, but should also work with 5.0.x or 6.x)
- Java Runtime Environment[46] 1.5.x (Since the implementation makes use of some new features which were introduced in Java 1.5.0, the use of older version is not encouraged.)
- PostgreSQL-Database[47] (or any other relational database, e.g. MySQL)

To install the system, follow the instructions below.

**Installation:**

1. Use the provided sql script to generate the necessary tables.
2. Place the `DLFlorid.war`-file in the `webapps`-directory of the Tomcat installation (`$CATALINA_HOME/webapps/`) and wait for the auto-deployment.
3. Navigate to the `WEB-INF` folder inside the newly created sub folder `DLFlorid` and configure the system as described in Section B.3.
4. Now, the DLFlorid web interface should be accessible under: `http://localhost:8080/DLFlorid/index.html` (depends on the specific Tomcat configuration). You can use the `status`-function to check the system status, particularly to test the connection to the FloridServer and to the database.

Note that a `build.xml` file is provided along with the system which can be used to rebuild the `war` file.

## B.3 Configuration

Basically, the system is configured via the `web.xml` file which should be found in the `$CATALINA_HOME/webapps/DLFlorid/WEB-INF` directory. This place will also be searched for further files, like the log4j-configuration file `log4j.properties`.

[45] Available from `http://tomcat.apache.org/`
[46] Available from `http://java.sun.com/`
[47] Available from `http://www.postgresql.org/`

**Directory Structure.**

```
$CATALINA_HOME/
  webapps/
    DLFlorid/
    WEB-INF/
      log4j.properties
      floridDL.sql
      web.xml

    examples/
      ...

    log/
      floridAdditions.log
```

Please adjust the following settings in the `web.xml` file:

- `DB_URL`: URL of database server
- `DB_USER`: database user id
- `DB_PASSWD`: database password
- `DB_TYPE`: database type (e.g. `PostgreSQL`)
- `DB_DRIVER`: name of JDBC driver class (e.g. `org.postgresql.Driver`)
- `MODEL_NAME`: Name of the "default" model
- `FLORID_URL`: URL of Florid WebService (e.g. `127.0.0.1`)
- `FLORID_TABLE_NAME`: Tablename to store Florid rules
- `FLORID_COMMIT_LENGTH`: the size of the "batch commits" in number of characters

The configuration file has to contain a valid database URL and user account information for the application to be able to open a database connection to the underlying relational database. Furthermore, the address of the `FloridServer` must be correct (see Section B.1).

# Bibliography

[AAB+05] José Júlio Alferes, Ricardo Amador, Erik Behrends, Mikael Berndtsson, François Bry, Gihan Dawelbait, Andreas Doms, Michael Eckert, Oliver Fritzen, Wolfgang May, Paula-Lavinia Pătrânjan, Loïc Royer, Franz Schenk, and Michael Schroeder. Specification of a Model, Language and Architecture for Reactivity and Evolution. deliverable I5-D4, Centro de Inteligência Artificial - CENTRIA, Universidade Nova de Lisboa, 2005.

[AAB+07] José Júlio Alferes, Ricardo Amador, Erik Behrends, Oliver Fritzen, Tobias Knapke, Wolfgang May, Franz Schenk, and Daniel Schubert. Reactive rule ontology: RDF/OWL level. *Deliverables I5-D6, REWERSE, March*, (I5-D6), 2007. Available from `http://rewerse.net/deliverables/m36/i5-d6.pdf`.

[ADG+05] Grigoris Antoniou, Carlos Viegas Damásio, Benjamin Grosof, Ian Horrocks, Michael Kifer, Jan Maluszynski, and Peter F. Patel-Schneider. Combining Rules and Ontologies: A survey. *Deliverables I3-D3, REWERSE, March*, 2005.

[AL04] Jürgen Angele and Georg Lausen. Ontologies in F-logic. In Steffen Staab and Rudi Studer, editors, *Handbook on Ontologies*, pages 29–50. Springer, 2004.

[Bal95] Mira Balaban. The F-logic approach for description languages. *Annals of Mathematics and Artificial Intelligence*, 15(1):19–60, 1995.

[BCM+03] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider, editors. *The Description Logic Handbook*. Cambridge University Press, 2003.

[BGT05] Harold Boley, Benjamin N. Grosof, and Said Tabet. RuleML Tutorial, 2005. Available from `http://www.ruleml.org/papers/tutorial-ruleml.html`.

[BH06] Jos de Bruijn and Stijn Heymans. Translating Ontologies from Predicate-based to Frame-based Languages. In *Proc. of the Second International Conference on Rules and Rule Markup Languages for the Semantic Web (RuleML-2006)*, Athens, Georgia, USA, November 10-11 2006. IEEE.

[BHL99] Tim Bray, Dave Hollander, and Andrew Layman. Namespaces in XML. Available from `http://www.w3.org/TR/REC-xml-names`, 1999.

[BL98a] Tim Berners-Lee. Notation 3. Available from `http://www.w3.org/DesignIssues/Notation3.html`, 1998.

[BL98b] Tim Berners-Lee. Semantic Web Road map. Available from `http://www.w3.org/DesignIssues/Semantic.html`, 1998.

[BL00] Tim Berners-Lee. Semantic Web - XML2000. Available from `http://www.w3.org/2000/Talks/1206-xml2k-tbl/Overview.html`, 2000.

[BLF99] Tim Berners-Lee and Mark Fischetti. *Weaving the Web.* Orion Business Books, 1999.

[BLFM98] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. Uniform Resource Identifier (URI): Generic Syntax. *Internet Engineering Task Force (IETF), Request for Comments (RFC) 2396*, 1998.

[BLH01] Tim Berners-Lee and James Hendler. Publishing on the Semantic Web. *Nature*, 410(6832):1023–1024, 2001.

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):28–37, 2001.

[Bor96] Alex Borgida. On the Relative Expressiveness of Description Logics and Predicate Logics. *Artificial Intelligence*, 82(1-2):353–367, 1996.

[BPLF05] Jos de Bruijn, Axel Polleres, Rubén Lara, and Dieter Fensel. OWL DL vs. OWL Flight: Conceptual Modeling and Reasoning on the Semantic Web. In *Proc. of the 14th International World Wide Web Conference (WWW2005)*, pages 623–632, Chiba, Japan, 2005. ACM.

[BSB04] Bryan Basham, Kathy Sierra, and Bert Bates. *Head First Servlets and JSP: Passing the Sun Certified Web Component Developer Exam (SCWCD).* O'Reilly Media, Inc., 2004.

[BvH$^+$] Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. OWL Web Ontology Language Reference. *W3C Recommendation 10 February 2004.* Available from `http://www.w3.org/TR/owl-ref/`.

[CDD$^+$04] Jeremy J. Carroll, Ian Dickinson, Chris Dollin, Dave Reynolds, Andy Seaborne, and Kevin Wilkinson. Jena: Implementing the Semantic Web Recommendations. In *Proc. of the 13th international World Wide Web conference on Alternate track papers & posters (WWW Alt. '04)*, pages 74–83. ACM Press, 2004.

[CGT90] Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic programming and Databases.* Springers, 1990.

[Cla78] Keith L. Clark. Negation as Failure. *Logic and Data Bases*, 1978.

[CSHD03] M.F. Cohen, J. Shade, S. Hiller, and O. Deussen. Wang Tiles for image and texture generation. *ACM Transactions on Graphics (TOG)*, 22(3):287–294, 2003.

[CvH+] Dan Connolly, Frank van Harmelen, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. Daml+oil (march 2001) reference description. Available from `http://www.w3.org/TR/daml+oil-reference`.

[Dat] Databases and Information Systems group, University Freiburg. The Florid Project. Available from `http://dbis.informatik.uni-freiburg.de/index.php?project=Florid`.

[DCv+02] Mike Dean, Dan Connolly, Frank van Harmelen, James Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. OWL Web Ontology Language 1.0 Reference. *W3C Working Draft*, 2002.

[dig] DL Implementation Group (DIG). Available from `http://dl.kr.org/dig/`.

[DLNS98] Francesco M. Donini, Maurizio Lenzerini, Daniele Nardi, and Andrea Schaerf. AL-log: Integrating Datalog and Description Logics. *Journal of Intelligent Information Systems*, 10(3):227–252, 1998.

[EIP+06] Thomas Eiter, Giovambattista Ianni, Axel Polleres, Roman Schindlauer, and Hans Tompits. Reasoning with Rules and Ontologies. In *Proc. of Summer School Reasoning Web 2006*, volume 4126 of *LNCS*, pages 93–127, Lisbon, Portugal, 2006. REWERSE.

[FFB04] Elisabeth Freeman, Eric Freeman, and Bert Bates. *Head First Design Patterns (Head First)*. O'Reilly, 2004.

[FH] Ernest Friedman-Hill. Jess - the Rule Engine for the Java Platform. Available from `http://www.jessrules.com/jess/index.shtml`.

[FHK+97] Jürgen Frohn, Rainer Himmeröder, Paul-Thomas Kandzia, Georg Lausen, and Christian Schlepphorst. FLORID: A Prototype for F-Logic. In *Proc. of the Thirteenth International Conference on Data Engineering*, page 583. IEEE Computer Society, 1997.

[GG95] Nicola Guarino and Pierdaniele Giaretta. Ontologies and Knowledge Bases. *Towards Very Large Knowledge Bases*, 1995.

[GHVD03] Benjamin N. Grosof, Ian Horrocks, Raphael Volz, and Stefan Decker. Description Logic Programs: Combining Logic Programs with Description Logic. In *Proc. of the 12th international conference on World Wide Web (WWW '03)*, pages 48–57, Budapest, Hungary, May 2003. ACM.

[Gol04] Christine Golbreich. Combining Rule and Ontology Reasoners for the Semantic Web. In *Proc. of RuleML 2004*, volume 3323 of *LNCS*, pages 6–22. Springer, 2004.

[Gru93] Thomas R. Gruber. A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.

[Gua98] Nicola Guarino. Formal Ontology and Information Systems. *Proceedings of the 1st International Conference June 6-8, Trento, Italy*, 1998.

[Guh] Ramanathan V. Guha. rdfdb query language. Available from `http://www.guha.com/rdfdb/query.html`.

[Hay03] Peter Hayes. RDF Semantics. *W3C Working Draft*, 23, 2003.

[HKS06] Ian Horrocks, Oliver Kutz, and Ulrike Sattler. The Even More Irresistible $\mathcal{SROIQ}$. In *Proc. of the 10th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 2006)*, pages 57–67. AAAI Press, 2006.

[Hor] Ian Horrocks. The FaCT System - Fast Classification of Terminologies. Available from `http://www.cs.man.ac.uk/~horrocks/FaCT/`.

[HPPSH05] Ian Horrocks, Bijan Parsia, Peter Patel-Schneider, and James Hendler. Semantic Web Architecture: Stack or Two Towers? In Francois Fages and Sylvain Soliman, editors, *Principles and Practice of Semantic Web Reasoning (PPSWR'05)*, number 3703 in LNCS, pages 37–41. Springer, 2005.

[hps] HP Labs Semantic Web Research. Available from `http://www.hpl.hp.com/semweb/`.

[HPS03] Ian Horrocks and Peter F. Patel-Schneider. Reducing OWL Entailment to Description Logic Satisfiability. In Dieter Fensel, Katia Sycara, and John Mylopoulos, editors, *Proc. of the 2003 International Semantic Web Conference (ISWC 2003)*, number 2870 in LNCS, pages 17–29. Springer, 2003.

[HPS04] Ian Horrocks and Peter F. Patel-Schneider. A Proposal for an OWL Rules Language. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 723–731. ACM, 2004.

[HPSB+04] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. W3C, May 2004.

[HPSBT05] Ian Horrocks, Peter F. Patel-Schneider, Sean Bechhofer, and Dmitry Tsarkov. OWL rules: A proposal and prototype implementation. *Journal of Web Semantics*, 3(1):23–40, 2005.

[HPSvH03] Ian Horrocks, Peter F. Patel-Schneider, and Frank van Harmelen. From $\mathcal{SHIQ}$ and RDF to OWL: The Making of a Web Ontology Language. *Journal of Web Semantics*, 1(1):7–26, 2003.

[jen] Jena - A semantic Web Framework for Java. Available from `http://jena.sourceforge.net/`.

[KC04] Graham Klyne and Jeremy J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. *W3C Recommendation*, 2004. Available from `http://www.w3.org/TR/rdf-concepts/`.

[Kif05] Michael Kifer. Rules and Ontologies in F-Logic. *Reasoning Web, First International Summer School, Tutorial Lectures*, pages 22–34, 2005.

[KL89] Michael Kifer and Georg Lausen. F-logic: a Higher-Order Language for Reasoning about Objects, Inheritance, and Scheme. In James Clifford, Bruce G. Lindsay, and David Maier, editors, *Proc. of the 1989 ACM SIGMOD International Conference on Management of Data*, pages 134–146, New York, NY, USA, 1989. ACM Press.

[KLW95] Michael Kifer, Georg Lausen, and James Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of ACM*, 42:741–843, July 1995.

[KP88] Phokion G. Kolaitis and Christos H. Papadimitriou. Why not negation by fixpoint? In *ACM Symposium on Principles of Database Systems (PODS)*, pages 231–239, 1988.

[LC00] Dongwon Lee and Wesley W. Chu. Comparative analysis of six xml schema languages. *SIGMOD Rec.*, 29(3):76–87, 2000.

[LR96] Alon Y. Levy and Marie-Christine Rousset. CARIN: A Representation Language combining Horn Rules and Description Logics. *European Conference on Artificial Intelligence*, pages 323–327, 1996.

[Mar] Maryland Information and Network Dynamics Lab. Pellet: An OWL DL Reasoner. Available from `http://pellet.owldl.com/`.

[May99] Wolfgang May. Information extraction and integration with FLORID: The MONDIAL case study. Technical Report 131, Universität Freiburg, Institut für Informatik, 1999. Available from `http://dbis.informatik.uni-goettingen.de/Mondial`.

[May00a] Wolfgang May. Florid version 3.0: User manual. Technical report, Institute of Computer Science, University Freiburg, Germany, 2000. Available from `http://dbis.informatik.uni-freiburg.de/content/projects/Florid/florid_manual.pdf`.

[May00b] Wolfgang May. How to Write F-Logic Programs in Florid. Technical report, Institute of Computer Science, University Freiburg, Germany, 2000. Available from `http://dbis.informatik.uni-freiburg.de/content/projects/Florid/florid_tutorial.pdf`.

[McB02] Brian McBride. Jena: A Semantic Web Toolkit. *IEEE Internet Computing*, 6(6):55–59, 2002.

[MHRS06] Boris Motik, Ian Horrocks, Riccardo Rosati, and Ulrike Sattler. Can OWL and Logic Programming Live Together Happily Ever After? In Isabel F. Cruz, Stefan Decker, Dean Allemang, Chris Preist, Daniel Schwabe, Peter Mika, Michael Uschold, and Lora Aroyo, editors, *Proc. of the 5th International Semantic Web Conference (ISWC 2006)*, volume 4273 of *LNCS*, pages 501–514. Springer, 2006.

[MSS04] Boris Motik, Ulrike Sattler, and Rudi Studer. Query Answering for OWL-DL with Rules. In Sheila A. McIlraith, Dimitris Plexousakis, and Frank van Harmelen, editors, *Proc. of the 3rd International Semantic Web Conference (ISWC 2004)*, volume 3298 of *LNCS*, pages 549–563. Springer, 2004.

[Ont] Ontoprise. OntoBroker: The Power of Inferencing. Available from `http://www.ontoprise.de`.

[Proa] Protégé ontology editor and knowledge-base framework. Available from `http://protege.stanford.edu/`.

[Prob] The Apache XML Project. Xerces Java Parser. Available from `http://xerces.apache.org/xerces-j/`.

[PS04] Bijan Parsia and Evren Sirin. Pellet: An OWL DL Reasoner. *Proceedings of the International Workshop on Description Logics*, 2004.

[Rac] Racerpro. Available from `http://www.racer-systems.com`.

[Sea04] Andy Seaborne. RDQL - A Query Language for RDF. *W3C Member Submission*, 9, 2004.

[SH01] Aaron Swartz and James Hendler. The Semantic Web: A Network of Content for the Digital City. *Proceedings of the 2nd Annual Digital Cities Workshop*, 2001.

[SSS91] Manfred Schmidt-Schauß and Gert Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48(1):1–26, 1991.

[Sto] FLORA-2: An Object-Oriented Knowledge Base Language. Available from `http://flora.sourceforge.net/`.

[Suna] Sun Microsystems, Inc. Java. Available from `http://java.sun.com/`.

[Sunb] Sun Microsystems, Inc. Java Servlets. Available from `http://java.sun.com/products/servlet/`.

[Sunc] Sun Microsystems, Inc. JavaServer Pages. Available from `http://java.sun.com/products/jsp/`.

[The] The Rule Markup Initiative. RuleML. Available from `http://www.ruleml.org`.

[VDO03] Raphael Volz, Stefan Decker, and Daniel Oberle. Bubo-Implementing OWL in rule-based systems. *Proceedings of WWW*, 2003.

[VMHG03] Raphael Volz, Boris Motik, Ian Horrocks, and Benjamin Grosof. Description Logics Programs: An Evaluation and Extended Translation. 2003. Available from `http://www.cs.man.ac.uk/~horrocks/Publications/download/2003/VMHG03a.pdf`.

[Vol04] Raphael Volz. *Web Ontology Reasoning with Logic Databases*. PhD thesis, University of Karlsruhe, 2004.

[W3Ca] The World Wide Web Consortium. Available from `http://www.w3c.org/`.

[W3Cb] W3C. Getting into RDF & Semantic Web using N3. Available from `http://www.w3.org/2000/10/swap/Primer`.

[W3Cc] W3C. RDF Validation Service. Available from `http://www.w3.org/RDF/Validator/`.

[W3C93] W3C. Naming and Addressing: URIs, URLs, ... Available from `http://www.w3.org/Addressing/`, 1993.

[W3C00] W3C. Resource Description Framework (RDF). Available from `http://www.w3.org/RDF`, 2000.

[W3C01a] W3C. N-Triples. Available from `http://www.w3.org/TR/rdf-sparql-query`, 2001.

[W3C01b] W3C. Semantic Web. Available from `http://www.w3.org/2001/sw`, 2001.

[W3C01c] W3C. XQuery 1.0: An XML Query Language. Available from `http://www.w3.org/TR/xquery`, 2001.

[W3C03] W3C. Web ontology (webont) working group charter. `http://www.w3.org/2002/11/swv2/charters/WebOntologyCharter`, 2003.

[W3C04a] W3C. OWL Web Ontology Language Guide. Available from `http://www.w3.org/TR/owl-guide/`, 2004.

[W3C04b] W3C. OWL Web Ontology Language Overview. Available from `http://www.w3.org/TR/owl-features/`, 2004.

[W3C04c] W3C. RDF Vocabulary Description Language 1.0: RDF Schema. Available from `http://www.w3.org/TR/rdf-schema`, 2004.

[W3C04d] W3C. RDF/XML Syntax Specification (Revised). Available from `http://www.w3.org/TR/rdf-syntax-grammar`, 2004.

[W3C04e] W3C. SPARQL Query Language for RDF. Available from `http://www.w3.org/TR/rdf-sparql-query`, 2004.

[W3C04f] W3C. SPARQL Query Results XML Format. Available from `http://www.w3.org/TR/rdf-sparql-XMLres/`, 2004.

[W3C06a] W3C. Extensible Markup Language (XML) 1.0 (Fourth Edition). W3C Recommendation, 2006. Available from `http://www.w3.org/TR/REC-xml/`.

[W3C06b] W3C. SPARQL Protocol for RDF. Available from `http://www.w3.org/TR/rdf-sparql-protocol/`, 2006.

[W3C07a] W3C. OWL 1.1 Web Ontology Language. Available from `http://webont.org/owl1.1/`, 2007.

[W3C07b] W3C. Uniform Resource Identifier (URI) Activity Statement. Available from `http://www.w3.org/Addressing/Activity`, 2007.

Ich erkläre hiermit, dass ich die Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen oder anderen Quellen entnommen sind, sind als solche kenntlich gemacht.

Göttingen, den 30. Mai 2007