RW² Project Deliverable

D1.2 v1.0

Report on reasoning techniques and prototype implementation for the WSML-Core and WSMO-DL languages

Document Version from July 1, 2006

Authors

Uwe Keller, Cristina Feier, Nathalie Steinmetz, Holger Lausen

with contributions from Stephan Grimm and Gábor Nagypál
Partners & Sponsor

<table>
<thead>
<tr>
<th>Partner</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Universität Innsbruck</td>
<td>ICT - Technologie Park</td>
</tr>
<tr>
<td>Digital Enterprise</td>
<td>Technikerstrasse 21a</td>
</tr>
<tr>
<td>Research Institute (DERI)</td>
<td>6020 Innsbruck, Austria</td>
</tr>
<tr>
<td></td>
<td>eMail: <a href="mailto:uwe.keller@deri.org">uwe.keller@deri.org</a></td>
</tr>
<tr>
<td></td>
<td>Tel.: +43 (0)512 507-6464</td>
</tr>
<tr>
<td>Electronic WebService GmbH</td>
<td>Tschiggfreystrasse 36 / Top 1</td>
</tr>
<tr>
<td></td>
<td>6020 Innsbruck, Austria</td>
</tr>
<tr>
<td></td>
<td>eMail: <a href="mailto:dieter.fensel@deri.org">dieter.fensel@deri.org</a></td>
</tr>
<tr>
<td></td>
<td>Tel.: +43 (0)512 900676</td>
</tr>
</tbody>
</table>

Sponsor

FIT-IT Programme

This work has been sponsored under the FIT-IT (Forschung, Innovation, Technologie - Informationstechnologie) Programme, http://www.fit-it.at/
Abstract

The deliverable reports on the progress on developing a generic framework for reasoning with WSML that captures both the rule based variants as well as WSML-DL. First, we describe the general approach and principles underlying the system architecture. Then, both the treatment of WSML-Core and of WSML-DL are described in detail. Finally, we outline our ideas on how to extend our system to cover the remaining parts of the WSML family of representation languages, i.e. WSML-RL (which is currently called WSML-Rule) and WSML-FOL.

This work has been sponsored under the FIT-IT (Forschung, Innovation, Technologie - Informationstechnologie) Programme, http://www.fit-it.at/
## Contents

1 **Introduction** .................................................. 5
   1.1 The Web Service Modeling Language and this Project .......... 5
   1.2 A generic Framework for Reasoning with WSML ................. 5
   1.3 Structure of the Deliverable .................................. 8

2 **Reasoning Framework for WSML Rule** .......................... 9
   2.1 Motivation ................................................. 9
   2.2 The WSML Language ........................................ 10
   2.3 Mapping WSML to Datalog ................................... 14
   2.4 Debugging Support .......................................... 20
   2.5 Reasoning Framework Overview .............................. 23
   2.6 Conclusion & Outlook ...................................... 25

3 **Reasoning Framework for WSML DL** ............................ 27
   3.1 OWL DL .................................................... 27
   3.2 Web Service Modeling Language (WSML) ...................... 36
   3.3 Mapping WSML-DL to OWL-DL ............................... 41
   3.4 DL Reasoning Tasks ........................................ 46
   3.5 DL Reasoner Overview ...................................... 49
   3.6 Reasoner Interfaces ........................................ 54
   3.7 Reasoning With WSML ...................................... 56

4 **Conclusions & Future Work** ..................................... 61
   4.1 Software & Online Demo ..................................... 62

**Bibliography** .................................................. 63
1 Introduction

1.1 The Web Service Modeling Language and this Project.

The goal of what is called semantic web services (SWS) [McIlraith et al., 2001] is the fruitful combination of Semantic Web technology and Web services. By using ontologies as the semantic data model for Web Service technologies Web Services have machine-processable annotations just as static data on the Web. Semantically enhanced information processing empowered by logical inference eventually shall allow the development of high quality techniques for automated discovery, composition, and execution of Services on the Web, stepping towards seamless integration of applications and data on the Web.

Two relevant initiatives have to be considered in the context of Semantic Web services. Chronologically, the first one is OWL-S [Ankolekar and Burstein, 2004], an upper level ontology for describing Web services, specified using OWL. As a result of several deficiencies detected in the previous model [Lara et al., 2004], a new framework for Semantic Web Services called WSMO was proposed as a refinement and extension of the Web Service Modeling Framework (WSMF) [Fensel and Bussler, 2002]. WSMF defines a rich conceptual model for the development and the description of Web Services based on two main requirements: maximal decoupling and strong mediation.

WSMO is accompanied by a formal language, the Web Service Modeling Language (WSML), that allows one to write annotations of Web services according to the conceptual model. In particular, WSML provides a specific language to describe and represent ontologies.

WSML [de Bruijn et al., 2005a] is a family of representation languages for the Semantic Web, taking Description Logics [Baader et al., 2003], Logic Programming [Lloyd, 1987] and First-Order Logic [Fitting, 1996a] as its semantic basis. Furthermore, it has been influenced by F-Logic [Kifer et al., 1995] and frame-based representation systems.

Since the goal of this project is reasoning with Semantic Web Services represented using WSMO, one of the main tasks that has to be solved is to design and implement reasoners for the different variants of WSML. In the context of this document we survey different reasoner implementations of various logical formalisms upon which the WSML variants are conceptually based and evaluate the usability of the implementations for the different WSML variants.

1.2 A generic Framework for Reasoning with the WSML Family of Languages

WSML is a family of formal languages that allows to capture various aspects of Web Services. Most fundamentally, WSML provides a particular ontology language. All other descriptions, such as Web Service functionalities are based on ontologies as the underlying conceptual and formal data model. Ontology reasoning in the context of WSML is thus the most fundamental form of reasoning that is required to realize higher-level tasks such as Web Service Discovery or Composition of Web Services. Therefore, ontology reasoning is the major object of interest and addressed in this deliverable.

[McIlraith et al., 2001] [Ankolekar and Burstein, 2004] [Lara et al., 2004] [Fensel and Bussler, 2002] [Baader et al., 2003] [Lloyd, 1987] [Fitting, 1996a] [Kifer et al., 1995]
Figure 1.1 depicts the WSML variants that we consider in the context of this project:

- **WSML-Core.** This language is defined by the intersection of Description Logic and Horn Logic. It has the least expressive power of all the languages of the WSML family and therefore the most preferable computational characteristics. It provides the basis for extended languages in various directions. Examples are rule based languages that are more expressive and support some form of closed-world reasoning while still being practically tractable.

- **WSML-DL.** This language is an extension of WSML-Core that supports Description Logics. It is closely related to OWL-Lite which is a recommendation of the W3C and has reasonable computational characteristics.

- **WSML-FOL.** This is a very expressive description language which is based on a fully fledged First-order Logic called Frame Logic (F-Logic) and covers WSML-DL. For most applications this language will provide sufficient expressiveness to model the services and all related knowledge like ontologies on a very detailed level. Nonetheless, because of the expressivity of the language one in general can not give computational guarantees in the most general case. Thus, a corresponding reasoner for this language has to be carefully designed and tuned towards the specific characteristics of our problem domain in order to ensure good performance for all relevant practical cases.

- **WSML-RL.** This language is an extension of WSML-Core that supports Horn Logic and thus extends WSML-Core by logical rules. This language in particular is equipped with an operational semantics and supports some form of closed-world reasoning. This language is based on the Horn fragment of F-Logic and thus particularly suited for object-oriented modelling.

All WSML variants share a common syntax, therefore, WSML-DL does not follow the standard Description Logic syntax, but rather the more-intuitive first-order (or rule-based) syntax including variables. Moreover, the syntax can be split into two part, namely the conceptual modelling part that allows to deal with common modelling primitives of ontologies in an intuitive way and the logical expression syntax that allows to write down more elaborate descriptions that go way beyond what can be expressed with the conceptual syntax. The single variants in the family distinguish themselves by the restrictions on the shared conceptual and logical expression syntax. Therefore, they provide different expressivity for modelers and have different computational properties.
Since the start of the project, WSML-RL has split into two languages, namely WSML-Flight and WSML-Rule. Both languages, as well as WSML-Core will be discussed in greater detail in Section 2.

In this deliverable, we consider specifically the WSML-Core and WSML-DL languages. However, instead of taking an isolated perspective on these languages, we particularly take into account the semantic relation to the other languages and their embedding in the WSML family, to come up with a system that leverages syntactic and semantic commonalities between the languages. Ideally, such a system is extensible towards the other languages in the WSML family (such as WSML-Rule and WSML-FOL) with reasonable effort.

Another important design decision with our system is modularity and reuse of existing technologies and flexibility in configuration and customization of a reasoning system for specific tasks. The fact that WSML is based on (theoretically and practically) well-studied knowledge representation paradigms, for which various systems have already been implemented and tested for some time, support this design decision.

Therefore, we decided to proceed in two phases:

- First, we constructed a generic framework (called WSML2Reasoner) which allows to map the various WSML variants to a common (shared) knowledge representation format and then via specific adapters to a concrete systems that are able to deal with the resulting knowledge bases.
  
  In our case, generalized clauses (or rules) including default negation can be seen as the shared knowledge representation format, in which WSML ontologies of all variants can be transformed.

- Second, we provide an own implementation of a reasoning system that is capable of dealing with the shared knowledge representation format.

Both phases are illustrated in Figure 1.2 in regard of the resulting conceptual architecture of our system.
An interesting feature of this approach is that the common knowledge representation format (especially particular subsets thereof) are problem-domain independent and widely used. Hence, our implementation is reusable for domains that are not related to WSML or Web Services. Furthermore, our system can be configured to use specific existing reasoning systems for (subsets) of the common representation format (i.e. generalized clauses or rules). This allows (a) people to use their specific existing reasoner of choice (which is independent of WSML) in the WSML context if they want to and (b) to provide the possibility to exploit systems that are developed already for years and that are therefore well-tuned with respect to performance and stability. We consider the latter as an important aspect that facilitates the use and dissemination of WSML in various domains.

**Current Status.** We have a prototype implementation available for WSML-Core and WSML-DL ontologies. The first phase is completed for WSML-Core, WSML-DL, WSML-Flight and WSML-Rule, whereas the second phase is only completed for the WSML-Core and WSML-Flight languages. In the course of this project, we do not envision to built a system specifically for WSML-DL only, but a prototype that is able to deal with the (more general) common representation format that subsumes WSML-DL. Such a system can then be equipped with specific techniques to address the restricted expressivity of WSML-DL.

The prototype can be configured to work with multiple existing reasoning systems for expressive Description Logics (such as Racer, Pellet, KAON2 and FACT++) as well as datalog engines (such as MINS, KAON2, DLV). Moreover, by design of the framework the integration of other systems (which are not supported yet) is simple and takes only little effort.

For the remainder of the project, the second phase becomes the focus of our interest. We are currently in the process of restructuring and extending our system for datalog reasoning (MINS) for the shared knowledge representation format and thus features such as function symbols, default negation, classical negation, disjunctive rule heads etc.

1.3 **Structure of the Deliverable**

This document is further structured as follows: Section 2 gives a detailed discussion our treatment of the rule-based WSML languages, i.e. WSML-Core, WSML-Flight and WSML-Rule. It shows especially how our transformation to the common knowledge representation format works. Section 3 explains in detail how we deal with WSML-DL ontologies and explains how we reuse existing state-of-the-art Description Logic reasoning systems. A respective mapping is presented which is reusable across various tools. Finally, Section 4 gives conclusions, outlines future work and explains where to find the implementation of the prototype.
2 Reasoning Framework for WSML Rule

The use of ontology languages for semantically annotating Web Services demands for reasoning support in order to facilitate tasks like automated discovery or composition of services based on semantic descriptions of their functionality. WSML is an ontology language specifically tailored to annotate Web Services, and part of its semantics adheres to the rule-based knowledge representation paradigm of logic programming. We present a framework to support reasoning with rule-based WSML language variants based on existing Datalog inference engines. Therein, the WSML reasoning tasks of knowledge base satisfiability and instance retrieval are implemented through a language mapping to Datalog rules and Datalog querying. Part of the WSML semantics is realised by a fixed set of rules that form meta-level axioms. Furthermore, the framework exhibits some debugging functionality that allows for identifying violated constraints and for pointing out involved instances and problem types. Its highly modular architecture facilitates easy extensibility towards other language variants and additional features. The available implementation of the framework provides the first reasoners for the WSML language.

2.1 Motivation

In the Semantic Web, recently Web Services are annotated by semantic descriptions of their functionality in order to facilitate tasks like automated discovery or composition of services. Such semantic annotation is formulated using ontology languages with logical formalisms underlying them. The matching of semantic annotation for discovery or the checking of type compatibility for composition requires reasoning support for these languages. A relatively new ontology language specifically tailored for the description of Web Services is WSML (Web Service Modeling Language) [de Bruijn et al., 2006], which comes in variants that follow the rule-based knowledge representation paradigm of logic programming [Lloyd, 1993]. WSML adds features of conceptual modelling and datatypes, known from frame-base knowledge representation, on top of logic programming rules.

We present a framework for reasoning with rule-based WSML variants that builds on existing infrastructure for inferencing in rule-based formalisms. The framework bases on a semantics-preserving syntactic transformation of WSML ontologies to Datalog programs, as described in the WSML specification [de Bruijn, 2005]. The WSML reasoning tasks of checking knowledge base satisfiability and of instance retrieval can then be performed by means of Datalog querying applied on a transformed ontology. Thus, the framework directly builds on top of existing Datalog inference engines.

Besides these standard reasoning tasks, the framework provides debugging features that support an ontology engineer in the task of ontology development: the engineer is pointed out to violated constraints together with some details on the ontological entities that cause the violation. Such a feature helps to improve the error reporting in situations of erroneous modelling.

Instead of directly mapping WSML entities, i.e. concepts, instances, attributes, to Datalog predicates and constants, we use special meta-level predicates and axioms which form a vocabulary on reified entities for reproducing the WSML language constructs in Datalog. This way of using Datalog as an
underlying formalism facilitates the metamodelling features of WSML.

The framework is implemented and can be readily used to reason about ontologies formulated in rule-based WSML. As such, it is the first implementation of a reasoning tool for this language. In contrast to most of the available rule engines and Datalog implementations, this reasoning framework supports the combination of typical rule-style representation with frame-style conceptual modelling, as offered by WSML.

2.2 The WSML Language

The Web Service Modeling Language (WSML) is a language for the specification of various aspects of Semantic Web Services (SWS), such as what functionality is provided by a SWS or how to interact with the SWS. It provides a formal language for the Web Service Modeling Ontology [1] (WSMO) [Roman et al., 2005] and is based on well-known logic-based knowledge representation (KR) formalisms (i.e. Description Logics [Baader et al., 2003] and Logic Programming [Lloyd, 1993]), specifying one coherent language framework for the semantic description of Web Services. In fact, WSML is a family of representation languages: the least expressive core language represents conceptually the intersection of the two KR formalisms Datalog [Dahr, 1996] and the Description Logic SHIQ (D) [Horrocks et al., 2003]. This core language is extended in the directions of Description Logics and Logic Programming in a principled manner with strict layering.

Internationalized Resource Identifier (IRIs) [Duerst and Suignard, 2005a] play a central role in WSML as (global) identifiers for symbols such as class names, attribute names or individuals. The concept of namespaces is used for logically grouping symbols in a vocabulary. Furthermore, WSML defines XML and RDF serializations for inter-operation over the Semantic Web. Thus, WSML is a Web and Semantic Web compliant KR language.

Although WSML takes into account all aspects of Web Service description identified by WSMO (i.e. Web services, goals, mediators and ontologies) we focus in the following on the WSML ontology description (sub)language. Reasoning with other elements of WSMO (e.g. matching of two Web Service capability descriptions) fundamentally relies on ontology reasoning in WSML and is reduced to ontology reasoning whenever this is possible.

WSML makes a clear distinction between the modeling of the different conceptual elements on the one hand and the specification of complex logical definitions on the other. To this end, the WSML syntax is split into two parts: the conceptual syntax and logical expression syntax. The conceptual syntax was developed from the user perspective, and is independent from the particular underlying logic; it shields the user from the peculiarities of the underlying logic. Having such a conceptual syntax allows for easy adoption of the language, since it allows for an intuitive understanding of the language for users not familiar with logical languages. In case the full power of the underlying logic is required, the logical expression syntax can be used. There are several entry points for logical expressions in the conceptual syntax, e.g. axioms in ontologies or capability descriptions in Goals and Web Services.

Conceptual Syntax – The WSML conceptual syntax for ontologies essentially allows for the modeling of concepts, instances, relations and relation instances.

1http://www.wsmo.org
We illustrate the description of WSML ontologies with an example in Listing 1.

Listing 1: WSML Example Ontology

```xml
concept Product
  hasProvider inverseOf(Provider#provides) impliesType Provider
concept ITBundle subConceptOf Product
  hasNetwork ofType (0 1) NetworkConnection
  hasOnlineService ofType (0 1) OnlineService
  hasProvider impliesType TelecomProvider
concept NetworkConnection subConceptOf BundlePart
  providesBandwidth ofType (0 1) integer
concept DSLConnection subConceptOf NetworkConnection axiom
  DSLConnection DSLConnection_Disjoint definedBy
    1→ 7x memberOf DSLConnection and 7x memberOf DSLConnection.
concept OnlineService subConceptOf BundlePart concept SharePriceFeed
subConceptOf OnlineService axiom SharePriceFeed requires bandwidth definedBy
  1→ 7b memberOf ITBundle and 7b[hasOnlineService hasValue ?o] and
  7o memberOf SharePriceFeed and
  7b[hasNetwork hasValue ?n] and
  7n[providesBandwidth hasValue ?x] and 7x < 512.
concept BroadbandBundle subConceptOf ITBundle
  hasNetwork ofType (1 1) DSLConnection axiom
  BroadbandBundle sufficient condition definedBy
    7b memberOf BroadbandBundle := 7b memberOf ITBundle
    and 7b[hasNetwork hasValue ?n] and
    7n memberOf DSLConnection.
instance GermanTelekom memberOf TelecomProvider. instance
  UbiqBankShareInfo memberOf SharePriceFeed. instance MyBundle
memberOf ITBundle
  hasNetwork hasValue ArcorDSL
  hasOnlineService hasValue UbiqBankShareInfo
  hasProvider GermanTelekom.
instance MSNDialup memberOf DialupConnection
  providesBandwidth hasValue 10.
instance ArcorDSL memberOf DSLConnection
  providesBandwidth hasValue 1024.
```

Concepts & Relations. The notion of concepts (or classes) plays a central role in ontologies. Concepts form the basic terminology of the domain of discourse. A concept may have instances and may have a number of attributes associated with it. Attribute definitions are grouped together in one frame (e.g. concept ITBundle in Listing 1 representing a product bundle (provided by a telecom provider) that consists up to one online network connection and up-to one online service which can be used over the network connection.)

Attribute definitions can take two forms, namely constraining (using ofType) and inferring (using impliesType) attribute definitions. Constraining attribute definitions define a typing constraint on the values for this attribute, similar to integrity constraints in databases; inferring attribute definitions allow that the type of the values for the attribute is inferred from the attribute definition, similar to range restrictions on properties in RDFS [Brickley and Guha, 2004a] and OWL [Dean and Schreiber, 2004]. Each attribute definition may have a number of features associated with it, namely, transitivity, symmetry, reflexivity, and the inverse of an attribute, as well as minimal and maximal cardinality constraints. In Listing 1 e.g. concept Product is defined to have an attribute hasProvider which is considered as the inverse of the attribute provides in concept Provider. As opposed to features of roles in OWL, attribute features such as transitivity, symmetry, reflexivity and inverse attributes are local to a concept in WSML. For instance, the definition of attribute hasProvider in class Product states that for any Product-instance (and only those) we can infer that the respective attribute value is an instance of class Provider. Furthermore, the inverse-relation between hasProvider and provides only holds for pairs of instances from Product and Provider. Similar constructs are available to define (n-ary) relations (denoting logical inter-relation between individuals and values) in WSML.

---

2The distinction between inferring and constraining attribute definitions is explained in more detail in [de Bruijn et al., 2005c] Section 2
ontologies.

**Instances of Concepts and Relations.** Concepts and relations may have an arbitrary number of instances associated with it. Instances explicitly specified in an ontology are those which are shared as part of the ontology. An instance may be member of zero or more concepts (or relations) and may have a number of attribute values associated with it, see for example the instance `MyBundle` in Listing 1 that is an `MyBundle` provided by the `GermanTelekom`. In WSML that the specification of concept membership for instances is optional and the attributes used in the instance specification do not necessarily have to occur in the associated concept definition. Consequently, WSML instances can be used to represent semi-structured data, i.e. set of attributes in a concept definitions do not need to match set of attributes for which values are defined in respective instances, and instances are interwoven into a labeled graph via attribute value definitions.

**Axioms.** Axioms provide a means to add arbitrary logical expressions to an ontology. Such logical expressions can be used to refine concept or relation definitions in the ontology, but also to add arbitrary axiomatic domain knowledge or express constraints. For example, a `SharePriceFeed` instances represent financial services that report in real-time of current prices of certain shares at the stock-market. Thus, a certain bandwidth is required, which is captured by axiom `SharePriceFeed requires bandwidth` in Listing 1 that states that the ontology may not contain an instance of `ITBundle` that provides a `SharePriceFeed` online services over a network which can only provide a bandwidth under a certain limit (here 512). Other examples are axiom `DialupConnection DSLConnection Disjoint` stating there can not be an object which is a dial-up connection and a DSL connection at the same time, or axiom `BroadbandBundle sufficient condition` which specifies that any `ITBundle` that provides a `DSLConnection` as its network connection actually is a `BroadbandBundle`. Thus, the latter axiom together with the (partial) definition of concept `BroadbandBundle` provides an exact characterization of the instances of this class.

**Logical Expression Syntax —** We will first explain the general logical expression syntax, which encompasses all WSML variants, and then describe the restrictions on this general syntax for each of the variants. The general logical expression syntax for WSML has a First-Order Logic (FOL) style, in the sense that it has constants, function symbols, variables, predicates and the usual logical connectives. WSML provides F-Logic [Kifer et al., 1995] based extensions in order to model concepts, attributes, attribute definitions, and subconcept and instance relationships. Finally, WSML has a number of connectives to facilitate the Logic Programming based variants, namely default negation (negation-as-failure), LP-implication (which differs from classical implication) and database-style integrity constraints.

Variables in WSML start with a question mark. Terms are either identifiers, variables, or constructed terms. As usual, an atom is constituted of an n-ary predicate symbol with n terms as arguments. Besides these standard atoms of FOL, WSML has a two special kind of atoms, called *molecules*, which are inspired by F-Logic and can be used to capture information about concepts, instances, attributes and attribute values: (a) An *isa-molecule* is an expression of the form `I memberOf C` (denoting a concept membership) or of the form `C1 subConceptOf C2` (denoting a subconcept relationship) whereby `I, C, Ci` are arbitrary terms. (b) An *object-molecule* is an expression of the form `I[A hasValue V]` (denoting attribute values of objects), of the form `C[A ofType T]` (denoting a constraining attribute signature), or of the form `C[A impliesType T]` (denoting an inferring attribute signature), with `I, A, V, C, T` being arbitrary
WSML has the usual first-order connectives: the unary negation operator \texttt{neg}, and the binary operators for conjunction \texttt{and}, disjunction \texttt{or}, right implication \texttt{implies}, left implication \texttt{impliedBy}, and bi-implication \texttt{equivalent}. Variables may be universally quantified using \texttt{forall} or existentially quantified using \texttt{exists}. First-order formulae are obtained by combining atoms using the mentioned connectives in the usual way.

Apart from First-Order formulae, WSML allows the use of the negation-as-failure symbol \texttt{naf} on atoms, the special Logic Programming implication symbol \texttt{:-} and the integrity constraint symbol \texttt{!-}. A logic programming rule consists of a head and a body, separated by the \texttt{:-} symbol. An integrity constraint consists of the symbol \texttt{!-} followed by a rule body. Negation-as-failure \texttt{naf} is only allowed to occur in the body of a Logic Programming rule or an integrity constraint. The further use of logical connectives in Logic Programming rules is restricted. The following logical connectives are allowed in the head of a rule: \texttt{and}, \texttt{implies}, \texttt{impliedBy}, and \texttt{equivalent}. The following connectives are allowed in the body of a rule (or constraint): \texttt{and}, \texttt{or}, and \texttt{naf}.

Axioms \texttt{BroadbandBundle\_sufficient\_condition} and \texttt{SharePriceFeed\_requires\_bandwidth} in Listing 1 are examples for the use of LP rules and integrity constraints in WSML ontologies.

**Particularities of the WSML Variants** – Each of the WSML variants defines a number of restrictions on the logical expression syntax. For example, LP rules and constraints are not allowed in WSML-Core and WSML-DL. Table 2.1 presents a number of language features and indicates in which variant the feature can occur.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Core</th>
<th>DL</th>
<th>Flight</th>
<th>Rule</th>
<th>Full</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classical Negation (\texttt{neg})</td>
<td>-</td>
<td>X</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td>Existential Quantification</td>
<td>-</td>
<td>X</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td>(Head) Disjunction</td>
<td>-</td>
<td>X</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td>n-ary relations</td>
<td>-</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Meta Modeling</td>
<td>-</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Default Negation (\texttt{naf})</td>
<td>-</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>LP implication</td>
<td>-</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Integrity Constraints</td>
<td>-</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Function Symbols</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Unsafe Rules</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Table 2.1: WSML Variants and Feature Matrix

**WSML-Core** allows only first-order formulae which can be translated to the DLP subset of \texttt{SHIQ(D)}. This subset is very close to the 2-variable fragment of First-Order Logic, restricted to Horn logic. Although WSML-Core might appear in the Table 2.1 featureless, it captures most of the conceptual model of WSML, but has only limited expressiveness within the logical expressions.

**WSML-DL** allows first-order formulae which can be translated to \texttt{SHIQ(D)}. This subset is very close to the 2-variable fragment of First-Order Logic. Thus, WSML DL allows classical negation, and disjunction and existential quantification in the heads of implications.

**WSML-Flight** extends the set of formulae allowed in WSML-Core by allowing variables in place of instance, concept and attribute identifiers and by allowing relations of arbitrary arity. In fact, any such formula is allowed in the head of a WSML-Flight rule. The body of a WSML-Flight rule allows conjunction, disjunction and default negation. The head and body are separated by the LP implication symbol. WSML-Flight additionally allows meta-modeling (e.g., classes-as-instances) and reasoning over the signature, because variables are al-
allowed to occur in place of concept and attribute names. **WSML-Rule** extends WSML-Flight by allowing function symbols and unsafe rules, i.e., variables which occur in the head or in a negative body literal do not need to occur in a positive body literal. **WSML-Full** The logical syntax of WSML-Full is equivalent to the general logical expression syntax of WSML and allows the full expressiveness of all other WSML variants.

In the following, we refer to the WSML-Core, WSML-Flight and WSML-Rule variants of WSML jointly as *rule-based WSML*.

**Reasoning Tasks in WSML** – We refer to any form of symbolic computation based on explicitly represented domain knowledge (such as an ontology) which helps to explicate implicit information as a *reasoning task*. In regard of WSML Ontologies, we consider the following ontology reasoning tasks as particularly useful and relevant to support SW and SWS applications and modelers: Let $O$ denote a WSML ontology and $\pi_{c-free}(O)$ denote the *constraint-free projection* of $O$, i.e., the ontology which can be derived from $O$ by removing all constraining description elements (such as attribute type constraints, cardinality constraints, integrity constraints etc.).

1. **Consistency checking** means checking whether $O$ is satisfiable. More precisely, it is about checking if no constraint in $O$ is violated and if the constraint-free projection $\pi_{c-free}(O)$ has a model $\mathcal{I}$. (2) **Entailment** means given some formula $\phi$, to check if no constraint in $O$ is violated and if in all models $\mathcal{I}$ of $\pi_{c-free}(O)$ it holds that all ground instances $\iota \in \text{ground}(\phi)$ of $\phi$ in $O$ are satisfied. We denote this by $O \models \phi$. (3) **Instance retrieval** means given an ontology $O$ and some formula $Q(\vec{x})$ with free variables $\vec{x} = (x_1, \ldots, x_n)$ to find all suitable terms $\vec{t} = (t_1, \ldots, t_n)$ constructed from symbols in $O$ only, such that the statement $Q(\vec{t})$ is entailed by $O$. We call $\vec{t}$ an *answer* to $Q(\vec{x})$ in $O$ and denote the set of answers by $\text{retrieve}_O(Q) = \{\vec{t} : \vec{t} = (t_1, \ldots, t_n), t_i \in \text{Term}(O), O \models Q(\vec{t})\}$. Rule-based WSML is based on the well-founded model semantics \cite{Gelder1991}. Therefore, the term ,,model" in the reasoning task definitions above stands for to the well-founded model of ontology $O$.

We will demonstrate later, that our framework allows to implement these reasoning tasks almost completely based on existing implementations of efficient datalog reasoning engines.

**2.3 Mapping WSML to Datalog**

The semantics of rule-based WSML is defined via a mapping to Datalog \cite{Dahr1996} with (in)equality and integrity constraints, as described in \cite{deBruin2005}. To make use of existing rule engines, the reasoning framework performs various syntactical transformations to convert an original ontology in WSML syntax into a semantically equivalent Datalog program. The WSML reasoning tasks of knowledge base satisfiability and instance retrieval are then realized by means of Datalog querying via calls to an underlying Datalog inference engine that is fed with the rules contained in this program.

**2.3.1 Ontology Transformations**

The transformation of a WSML ontology to Datalog rules forms a pipeline of single transformation steps which are subsequently applied, starting from the
Table 2.2: Examples for axiomatizing conceptual ontology modeling elements.

<table>
<thead>
<tr>
<th>conceptual syntax</th>
<th>logical expression(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \tau_{\text{axioms}}(\text{concept } C_1 \text{ subConceptOf } C_2) )</td>
<td>( C_1 \text{ subConceptOf } C_2 ).</td>
</tr>
<tr>
<td>( \tau_{\text{axioms}}(\text{concept } C_1 \ A \text{ ofType } (0,1) T) )</td>
<td>( C_1[A \text{ ofType } T]. )</td>
</tr>
<tr>
<td>( \tau_{\text{axioms}}(\text{relation } R_1/n \text{ subRelationOf } R_2) )</td>
<td>( R_1(\vec{x}) \text{ implies } R_2(\vec{x}), ) where ( \vec{x} = (x_1, ..., x_n) ).</td>
</tr>
<tr>
<td>( \tau_{\text{axioms}}(\text{instance } I \text{memberOf } C \ A \text{ hasValue } V) )</td>
<td>( I \text{memberOf } C. )</td>
</tr>
</tbody>
</table>

Axiomatization. In a first step, the transformation \( \tau_{\text{axioms}} \) is applied as a mapping \( O \rightarrow 2^{2^E} \) from the set of all valid rule-based WSML ontologies to the powerset of all logical expressions that conform to rule-based WSML. In this transformation step, all conceptual syntax elements, such as concept and attribute definitions or cardinality and type constraints, are converted into appropriate axioms specified by logical expressions. Table 2.2 shows the details of some of the conversions performed by \( \tau_{\text{axioms}} \), based on [de Bruin, 2005]. During the transformation, for each expression \( e \) in the WSML Ontology \( O \in O \) that matches a pattern on the left-hand side of Table 2.2 the formulae \( \tau_{\text{axioms}}(e) \) are created and added to the resulting theory \( \tau_{\text{axioms}}(O) \).

The meta variables \( C, C_i \) range over identifiers of WSML concepts, \( R_i, A_i \) over identifiers of WSML relations and attributes, \( T \) over identifiers of WSML concepts or datatypes and \( V \) over identifiers of WSML instances or datatype values.

Normalization. The transformation \( \tau_{\text{norm}} \) is applied as a mapping \( 2^{2^E} \rightarrow 2^{2^E} \) to normalize WSML logical expressions. This normalization step reduces the complexity of WSML logical expressions according to [de Bruin, 2005, Section 8.2], to bring the expressions closer to the simple syntactic form of literals in Datalog rules. The reduction includes conversion to negation and disjunctive normal forms as well as decomposition of complex WSML molecules. Table 2.3 shows how the various logical expressions are normalized in detail. The meta variables \( E_i \) range over logical expressions in rule-based WSML, while \( X, Y_i \) range over parts of WSML molecules. After \( \tau_{\text{norm}} \) has been applied, the resulting WSML logical expressions have the form of logic programming rules with no deep nesting of logical connectives.

Lloyd-Topor Transformation. The transformation \( \tau_{lt} \) is applied as a mapping \( 2^{2^E} \rightarrow 2^{2^E} \) to flatten the complex WSML logical expressions, producing simple rules according to the Lloyd-Topor transformations [Lloyd and Topor, 1984], as shown in Table 2.4. Again, the meta variables \( E_i, H_i, B_i \) range over WSML logical expressions, while \( H_i \) and \( B_i \) match the form of valid rule head and body expressions, respectively, according to [de Bruin, 2005].

After this step, the resulting WSML expressions have the form of proper Datalog rules with a single head and conjunctive (possibly negated) body literals.
original expression | normalized expression
--- | ---
\( \tau_{\text{norm}}(\{E_1, \ldots, E_n\}) \) | \{\( \tau_{\text{norm}}(E_1) \), \ldots, \( \tau_{\text{norm}}(E_n) \)\}
\( \tau_{\text{norm}}(E_x \land E_y) \) | \( \tau_{\text{norm}}(E_x) \) and \( \tau_{\text{norm}}(E_y) \)
\( \tau_{\text{norm}}(E_x \lor E_y) \) | \( \tau_{\text{norm}}(E_x) \) or \( \tau_{\text{norm}}(E_y) \)
\( \tau_{\text{norm}}(E_x \land \neg E_y) \) | \( \tau_{\text{norm}}(E_x) \) and \( \tau_{\text{norm}}(\neg E_y) \)
\( \tau_{\text{norm}}(\neg E_x) \) | \( \neg \tau_{\text{norm}}(E_x) \)
\( \tau_{\text{norm}}(E_x \implies E_y) \) | \( \tau_{\text{norm}}(E_x) \) implies \( \tau_{\text{norm}}(E_y) \)
\( \tau_{\text{norm}}(E_x \iff E_y) \) | \( \tau_{\text{norm}}(E_x) \) is impliedBy \( \tau_{\text{norm}}(E_y) \)
\( \tau_{\text{norm}}(X[Y_1, \ldots, Y_n]) \) | \( X[Y_1] \) and \( \ldots \) and \( X[Y_n] \)

Table 2.3: Normalization of WSML logical expressions.

<table>
<thead>
<tr>
<th>original expression</th>
<th>simplified rule(s)</th>
</tr>
</thead>
</table>
\( \tau_{\text{norm}}(\{E_1, \ldots, E_n\}) \) | \( \{\tau_{\text{norm}}(E_1), \ldots, \tau_{\text{norm}}(E_n)\} \)
\( \tau_{\text{norm}}(H). \) | \( \tau_{\text{datalog}}(H) \)
\( \tau_{\text{norm}}(E_x \land \neg E_y) \) | \( \tau_{\text{datalog}}(E_x) \) \land \( \tau_{\text{datalog}}(\neg E_y) \)
\( \tau_{\text{datalog}}(C_x \text{ subConceptOf } C_y) \) | \( \tau_{\text{datalog}}(C_x) \) \land \( \tau_{\text{datalog}}(C_y) \)
\( \tau_{\text{datalog}}(I \text{ memberOf } C) \) | \( \tau_{\text{datalog}}(I) \) \land \( \tau_{\text{datalog}}(C) \)
\( \tau_{\text{datalog}}(I[a \text{ hasValue } V]) \) | \( \tau_{\text{datalog}}(I[a]) \) \land \( \tau_{\text{datalog}}(V) \)
\( \tau_{\text{datalog}}(C[i \text{ impliesType } T]) \) | \( \tau_{\text{datalog}}(C[i]) \) \land \( \tau_{\text{datalog}}(\text{typeof } T) \)
\( \tau_{\text{datalog}}(C[a \text{ ofType } T]) \) | \( \tau_{\text{datalog}}(C[a]) \) \land \( \tau_{\text{datalog}}(\text{typeof } T) \)
\( \tau_{\text{datalog}}(f(X_1, \ldots, X_n)) \) | \( \tau_{\text{datalog}}(f) \) \land \( \tau_{\text{datalog}}(X_1, \ldots, X_n) \)
\( \tau_{\text{datalog}}(X = Y) \) | \( \tau_{\text{datalog}}(\text{eq } X, Y) \)
\( \tau_{\text{datalog}}(X \neq Y) \) | \( \tau_{\text{datalog}}(\text{neq } X, Y) \)

Table 2.4: Lloyd-Topor transformations.

**Datalog Rule Generation.** In a final step, the transformation \( \tau_{\text{datalog}} \) is applied as a mapping \( 2^S \rightarrow P \) from WSML logical expressions to the set of all Datalog programs, yielding generic Datalog rules that represent the content of the original WSML ontology. Rule-style language constructs, such as rules, facts, constraints, conjunction and (default) negation, are mapped to the respective Datalog elements. All remaining WSML-specific language constructs, such as \( \text{subConceptOf} \) or \( \text{ofType} \), are replaced by special meta-level predicates for which the semantics of the respective language construct is encoded in meta-level axioms as described in Section 2.3.2. Table 2.5 shows the mapping from WSML logical expressions to Datalog including the meta-level predicates \( p_{\text{SCO}}, p_{\text{MO}}, p_{\text{hval}}, p_{\text{type}} \) and \( p_{\text{typeof}} \) that represent their respective WSML language constructs as can be seen from the mapping. The meta variables \( E, H, B \) range over WSML logical expressions with a general, a head or a body form, while \( C, I, a \) denote WSML concepts, instances and attributes. Variables \( T \) can either assume a concept or a datatype, and \( V \) stands for either an instance or a data value, accordingly.

<table>
<thead>
<tr>
<th>WSML</th>
<th>Generic Datalog</th>
</tr>
</thead>
</table>
\( \tau_{\text{datalog}}(\{E_1, \ldots, E_n\}) \) | \{\( \tau_{\text{datalog}}(E_1), \ldots, \tau_{\text{datalog}}(E_n)\)\}
\( \tau_{\text{datalog}}(\neg B) \) | \( \square \vdash \neg \tau_{\text{datalog}}(B) \)
\( \tau_{\text{datalog}}(H) \) | \( \tau_{\text{datalog}}(H) \)
\( \tau_{\text{datalog}}(E_x \land \neg E_y) \) | \( \tau_{\text{datalog}}(E_x) \) \land \( \tau_{\text{datalog}}(\neg E_y) \)
\( \tau_{\text{datalog}}(C_x \text{ subConceptOf } C_y) \) | \( \tau_{\text{datalog}}(C_x) \) \land \( \tau_{\text{datalog}}(C_y) \)
\( \tau_{\text{datalog}}(I \text{ memberOf } C) \) | \( \tau_{\text{datalog}}(I) \) \land \( \tau_{\text{datalog}}(C) \)
\( \tau_{\text{datalog}}(I[a \text{ hasValue } V]) \) | \( \tau_{\text{datalog}}(I[a]) \) \land \( \tau_{\text{datalog}}(V) \)
\( \tau_{\text{datalog}}(C[i \text{ impliesType } T]) \) | \( \tau_{\text{datalog}}(C[i]) \) \land \( \tau_{\text{datalog}}(\text{typeof } T) \)
\( \tau_{\text{datalog}}(C[a \text{ ofType } T]) \) | \( \tau_{\text{datalog}}(C[a]) \) \land \( \tau_{\text{datalog}}(\text{typeof } T) \)
\( \tau_{\text{datalog}}(f(X_1, \ldots, X_n)) \) | \( \tau_{\text{datalog}}(f) \) \land \( \tau_{\text{datalog}}(X_1, \ldots, X_n) \)
\( \tau_{\text{datalog}}(X = Y) \) | \( \tau_{\text{datalog}}(\text{eq } X, Y) \)
\( \tau_{\text{datalog}}(X \neq Y) \) | \( \tau_{\text{datalog}}(\text{neq } X, Y) \)

Table 2.5: Transformation WSML logical expressions to Datalog.
The resulting Datalog rules are of the form

$$H : = B_1 \land \ldots \land B_n$$

where $H$ and $B_i$ are literals for the head and the body of the rule, respectively. Body literals can be negated in the sense of negation-as-failure, which is denoted by $\sim B_i$. As usual, rules with an empty body represent facts, and rules with an empty head represent constraints. The latter is denoted by the head being the empty clause symbol $\square$.

Ultimately, we define the basic transformation $\tau$ for converting a rule-based WSML ontology into a Datalog program based on the single transformation steps introduced before by $\tau = \tau_{\text{datalog}} \circ \tau_{\text{lt}} \circ \tau_{\text{norm}} \circ \tau_{\text{axioms}}$.

As a mapping $\tau : O \rightarrow P$, this concatenation of the single steps is applied to a WSML ontology $O \in O$ to yield a semantically equivalent Datalog program $\tau(O) = P \in P$ when interpreted with respect to the meta-level axioms discussed next.

### 2.3.2 WSML Semantics through Meta-Level Axioms

The mapping from WSML to datalog in the reasoning framework works such that each WSML-identifiable entity, i.e. concept, instance, attribute etc., is mapped to an instance (or logical constant) in datalog, as depicted in Figure [2.1](#). There, the concepts $C_1, C_2, C_3$ as well as the instances $I_1, I_2$ and the attribute $a$ are mapped to constants such as $I_{C_1}, I_{I_1}$ or $I_{a}$ in datalog, representing the original WSML entities on the instance level.

Accordingly, the various special-purpose relations that hold between WSML entities, such as $\text{subConceptOf}$, $\text{memberOf}$ or $\text{hasValue}$, are mapped to datalog predicates that form a meta-level vocabulary for the WSML language constructs. These are the meta-level predicates that appear in Table [2.5](#) and which are applied to the datalog constants that represent the WSML entities. The facts listed in Figure [2.1](#) illustrate the use of the meta-level predicates. For example, the predicate $p_{\text{SCO}}$ takes two datalog constants as arguments that represent WSML concepts, to state that the concept represented by the first argument is a subconcept of the one represented by the second argument; on the other hand, the predicate $p_{\text{MO}}$ takes a datalog constant that represents a WSML instance and one that represents a WSML concept, to state that the instance is in the extension of this concept.

In contrast to a direct mapping from WSML to datalog with concepts, attributes and instances mapping to unary predicates, binary predicates and constants, respectively, this indirect mapping allows for the WSML metamodelling facilities. Metamodelling allows an entity to be a concept and an instance at the same time. By representing a WSML entity as a datalog constant, it could, for example, fill both the first as well as the second argument of e.g. the predicate $p_{\text{MO}}$, in which case it is interpreted as both an instance and a concept at the same time.

A fixed set $P_{\text{meta}}$ of datalog rules forms the meta-level axioms which assure that the proper semantics of the WSML language is maintained. In these axioms, the meta-level predicates are interrelated according to the semantics of the different language constructs. Table [2.5](#) shows the rules that make up the meta-level axioms in $P_{\text{meta}}$. Axiom (1) realizes transitivity for the WSML $\text{subConceptOf}$ construct, while axiom (2) ensures that an instance of a subconcept

\[3\] Later on, the transformation pipeline is further extended to support datatypes and debugging features.
is also an instance of its superconcepts. Axiom (3) realizes the semantics for the `impliesType` construct for attribute ranges: any attribute value is concluded to be in the extension of the range type declared for the attribute. Finally, axiom (4) realizes the semantics of the `ofType` construct by a constraint that is violated whenever an attribute value cannot be concluded to be in the extension of the declared range type.

### 2.3.3 WSML Reasoning by Datalog Queries

To perform reasoning over the original WSML ontology $O$ with an underlying datalog inference engine, a datalog program

$$P_O = P_{\text{meta}} \cup \tau(O)$$

is built up that consists of the meta-level axioms together with the transformed ontology. The different WSML reasoning tasks are then realized by performing Datalog queries on $P_O$. Posing a query $Q(\bar{x})$ to a Datalog program $P \in \mathcal{P}$ is denoted by

$$(P, ? - Q(\bar{x}))$$

and yields a set of tuples that instantiate the vector $\bar{x}$ of variables in the query.
Ontology Consistency  – The task of checking a WMSL ontology for consistency is done by querying for the empty clause, as expressed by the following equivalence.

\[ O \text{ is satisfiable} \iff (P_O, ? - \Box) = \emptyset \]

If the resulting set is empty then the empty clause could not be derived from the program and the original ontology is satisfiable, otherwise it is not.

Entailment  – The reasoning task of entailment of ground facts by a WSML ontology can be done by using queries that contain no variables, as expressed in the following equivalence.

\[ O \models \phi \iff (P_O, ? - \tau'(\phi')) \neq \emptyset \]

From the WSML ground fact \( \phi \in \mathcal{E} \) we derive a non-ground formula \( \phi' \in \mathcal{E} \) by replacing the left-most occurrence of a constant by the variable \( x \). \( \phi' \) is then transformed to Datalog with a transformation \( \tau' = \tau_{datalog} \circ \tau_{lt} \circ \tau_{norm} \), similar to the one that is applied to the ontology, and is evaluated together with the Datalog program \( P_O \). If the resulting set is non-empty then \( \phi \) is entailed by the original ontology, otherwise it is not.

 Retrieval  – Similarly, instance retrieval can be performed by posing queries that contain variables to the Datalog program \( P_O \), as expressed in the following equivalence.

\[ \text{retrieve}_{O}(Q) = (P_O, ? - \tau'(Q(\vec{x}))) \]

The query \( Q(\vec{x}) \), formulated as a WSML logical expression with free variables \( \vec{x} \), is transformed to Datalog and evaluated together with the program \( P_O \). The resulting set contains all tuples \( \vec{x} \) for which an instantiation of the query expression is entailed by the original ontology. To give an example, the query \( Q(\vec{x}) = ?x \text{memberOf BroadbandBundle} \) posed to the ontology in Listing 1 yields the set \( \{(\text{MyBundle})\} \) that contains one unary tuple with the instance \( \text{MyBundle} \), which can be inferred to be a broadband bundle due to its high network bandwidth.

2.3.4 Realising Datatype Reasoning

Although most of the generic Datalog rules are understood by practically any Datalog implementation, realizing datatype reasoning has some intricate challenges.

The main challenge in implementing datatype reasoning is related to Axiom (4) in Table 2.3.2, which checks attribute type constraints. The crucial part of the axiom is the literal

\[ \sim p_{\text{mo}}(V, C_2) \]

because for datatype values no explicit membership facts are included in the ontology that could instantiate this literal. Consider, for example, the instance MSNDialup from the WSML ontology in Section 2.2 – there is no fact \( p_{\text{mo}}(10, \text{integer}) \) for the value of the \text{providesBandwidth} attribute. Whenever a value is defined for an attribute constrained by \text{ofType}, Axiom (4) would cause a constraint violation.

To solve this problem, \( p_{\text{mo}} \) facts should be generated for all datatype constants that appear in the ontology. I.e., for each such constant in the ontology, axioms of the following form
should appear:

\[ p_{\text{mo}}(V, D) : - \text{typeOf}(V, D_T) \]

where \( D \) denotes the WSML datatype, \( D_T \) denotes a datatype supported by the underlying Datalog implementation, which is compatible with the WSML datatype, and \( \text{typeOf} \) denotes a built-in predicate implemented by the Datalog tool, which checks whether a constant value belongs to the specified datatype.

These additional meta-level axioms result in a new set of Datalog rules, denoted by \( P_{\text{data}} \), which are no longer in generic Datalog but use tool-specific built-in predicates of the underlying inference engine. The Datalog program \( P_O \) is extended with this new set of rules as follows.

\[ P_O = P_{\text{meta}} \cup P_{\text{data}} \cup \tau(O) \]

In addition to datatypes, WSML also supports some predefined predicates on datatypes, such as numeric comparison\(^4\). For example, the definition of the \texttt{SharePriceFeed.requires.bandwidth} axiom from the WSML ontology in Section 2.2 uses a shortcut of the WSML \texttt{numericLessThan} predicate (denoted by \(<\)). Clearly, these special WSML predicates have to be translated to the corresponding built-in predicates supported by the underlying Datalog reasoner. Therefore, we introduce a new tool-specific transformation step \( \tau_{\text{dpred}} \) as a mapping \( P \to P' \), which translates all predefined WSML datatype predicates in the generic Datalog program to tool-specific built-in predicates. The transformation pipeline \( \tau \) is augmented by this additional step and is redefined as follows.

\[ \tau = \tau_{\text{dpred}} \circ \tau_{\text{datalog}} \circ \tau_{\text{lt}} \circ \tau_{\text{norm}} \circ \tau_{\text{axioms}} \]

To summarize the discussion, the underlying Datalog implementation must fulfill the following requirements to support WSML datatype reasoning: (i) It should provide built-in datatypes that correspond to WSML datatypes. (ii) It should provide a predicate (or predicates) for checking whether a datatype covers a constant and (iii) It should provide built-in predicates that correspond to datatype-related predefined predicates in WSML.

### 2.4 Debugging Support

During the process of ontology development, an ontology engineer can easily construct an erroneous model containing contradictory information. In order to produce consistent ontologies, inconsistencies should be reported to engineers with some details about the ontological elements that cause the inconsistency.

In rule-based WSML, the source for erroneous modelling are always constraints, together with a violating situation of concrete instances related via attributes. The plain Datalog mechanisms employed in the reasoning framework according to Section 2.3 only allow for checking whether some constraint is violated, i.e. whether the empty clause is derived from \( P_O \) indicating that the original ontology \( O \) contains errors – more detailed information about the problem is not reported. Experience shows that it is a very hard task to identify and correct errors in the ontology without such background information.

In our framework, we support debugging features that provide information about the ontology entities which are involved in a constraint violation. We achieve this by replacing constraints with appropriate rules that contain the needed additional information in their heads.

\(^4\)A full list of WSML datatypes can be found in the WSML specification [de Bruin, 2005].
2.4.1 Identifying Constraint Violations

In case of an inconsistent ontology due to a constraint violation, two things are of interest to the ontology engineer: a) the type of constraint that is violated and b) the entities, i.e. concepts, attributes, instances, etc., that are involved in the violation.

To give an example, consider the WSML ontology in Section 2.2. There, the attribute hasOnlineService of the concept ITBundle is constrained to instances of type OnlineService. Suppose we replace the current value of the attribute hasOnlineService for the instance MyBundle by the instance MSNDialup. Then, this constraint would be violated because MSNDialup is not an instance of the concept OnlineService. For an ontology engineer who needs to repair this erroneous modelling, it is important to know the entities that cause the violation, which in this case are the attribute hasOnlineService together with the range concept OnlineService and the non-conforming instance MSNDialup.

For the various types of constraint violations, the information needed by the ontology engineer to track down the problem successfully is different from case to case.

Attribute Type Violation – An attribute type constraint of the form \( C[a \text{ ofType } T] \) is violated whenever an instance of the concept \( C \) has value \( V \) for the attribute \( a \), and it cannot be inferred that \( V \) belongs to the type \( T \). Here, \( T \) can be either a concept or a datatype, while \( V \) is then an instance or a data value, accordingly. In such a situation, an ontology engineer is particularly interested in the instance \( I \), in the attribute value \( V \) that caused the constraint violation, together with the attribute \( a \) and the expected type \( T \) which the value \( V \) failed to adhere to.

Minimum Cardinality Violation – A minimum cardinality constraint of the form \( \text{concept } C a (n^*) \), is violated whenever the number of distinguished values of the attribute \( a \) for some instance \( I \) of the concept \( C \) is less than the specified cardinality \( n \). In such a situation, an ontology engineer is particularly interested in the instance \( I \) that failed to have a sufficient number of attribute values, together with the actual attribute \( a \). (Information about how many values were missing can be learned by querying the ontology separately.)

Maximum Cardinality Violation – A maximum cardinality constraint of the form \( \text{concept } C a (0n) \), is violated whenever the number of distinguished values of the attribute \( a \) for some instance \( I \) of the concept \( C \) exceeds the specified cardinality \( n \). Again, here an ontology engineer is particularly interested in the instance \( I \) for which the number of attribute values was exceeded, together with the actual attribute \( a \).

User-Defined Constraint Violation – Not only built-in WSML constraints, but also user-defined constraints, contained in an axiom definition of the form \( \text{axiom } Ax_{ID} \text{ definedBy } \vdash B \), can be violated. In this case, the information which helps an ontology engineer to repair an erroneous situation is dependent on the arbitrarily complex body \( B \) and cannot be determined in advance. However, a generic framework can at least identify the violated constraint by reporting the identifier \( Ax_{ID} \) of the axiom.

To give an example, consider again the ontology from Section 2.2. Replacing the network connection ArcorDSL of MyBundle by the slower MSNDialup one results in the a violation of the user-defined constraint specified by the axiom
named SharePriceFeed requires bandwidth. This constraint requires a certain bandwidth for connections in bundles with share price feed online services, which is not met by MSNDialup, and thus the ontology engineer is reported the axiom name that identifies the violated constraint.

### 2.4.2 Debugging by Meta-Level Reasoning

In our framework, we realize the debugging features for reporting constraint violations by replacing constraints with a special kind of rules. Instead of deriving the empty clause, as constraints do, these rules derive information about occurrences of constraint violations by instantiating debugging-specific meta-level predicates with the entities involved in a violation. In this way, information about constraint violations can be queried for by means of Datalog inferencing.

The replacement of constraints for debugging is included in the transformation pipeline

$$\tau = \tau_{pred} \circ \tau_{datalog} \circ \tau_{lt} \circ \tau_{norm} \circ \tau_{debug} \circ \tau_{axioms}$$

where the additional transformation step $\tau_{debug}$ is applied after the WSML conceptual syntax has been resolved, replacing constraints on the level of WSML logical expressions. Table 2.7 shows the detailed replacements performed by $\tau_{debug}$ for the different kinds of constraints.

Minimal cardinality constraints (with bodies $B_{\text{mincard}}$) and maximal cardinality constraints (with bodies $B_{\text{maxcard}}$) are transformed to rules by keeping their respective bodies and adding a head that instantiates one of the predicates $p_{v\text{mincard}}$ and $p_{v\text{maxcard}}$ to indicate the respective cardinality violation. The variables for the involved attribute $a$ and instance $I$ are the ones that occur in the respective constraint body $B$.

Similarly, a user-defined constraint is turned into a rule by keeping the predefined body $B_{\text{user}}$ and including a head that instantiates the predicate $p_{v\text{user}}$ to indicate a user-defined violation. The only argument for the predicate $p_{v\text{user}}$ is the identifier $Ax_{ID}$ of the axiom, by which the constraint has been named.

Constraints on attribute types are handled differently because these constraints are not expanded during the transformation $\tau_{axioms}$: they are rather represented by WSML ofType-molecules for which the semantics is encoded in the meta-level axioms $P_{\text{meta}}$. In order to avoid the modification of $P_{\text{meta}}$ in the reasoning framework, such molecules are expanded by $\tau_{debug}$, as shown in Table 2.7.

To maintain the constraining semantics of the replaced constraints, an additional set of meta-level axioms $P_{\text{debug}} \in \mathcal{P}$ is included for reasoning. The rules in $P_{\text{debug}}$ derive the empty clause for any occurrence of a constraint violation, as shown in Table 2.8.

Including the debugging features, the Datalog program for reasoning about the original ontology then turns to

$$P_O = P_{\text{meta}} \cup P_{\text{data}} \cup P_{\text{debug}} \cup \tau(O) \ .$$

Occurrences of constraint violations can be recognized by querying $P_O$ for instantiations of the various debugging-specific meta-level predicates $p_{v\text{otype}}$, $p_{v\text{mincard}}$, $p_{v\text{maxcard}}$ and $p_{v\text{user}}$. For example, the set

$$(P_O, ? - p_{v\text{otype}}(a, T, I, V))$$

5 After this expansion of ofType molecules, the respective axiom (4) in $P_{\text{meta}}$ for realising the semantics of attribute type constraints does not apply anymore.
Table 2.7: Replacing constraints by rules.

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_{debug}({E_1, \ldots, E_n})$</td>
<td>${\tau_{debug}(E_1), \ldots, \tau_{debug}(E_n)}$</td>
</tr>
<tr>
<td>$\tau_{debug}(\neg B_{mincard})$</td>
<td>$p_{user}(a, I) : - B_{mincard}$</td>
</tr>
<tr>
<td>$\tau_{debug}(\neg B_{maxcard})$</td>
<td>$p_{user}(a, I) : - B_{maxcard}$</td>
</tr>
<tr>
<td>$\tau_{debug}(C[a ofType T])$</td>
<td>$p_{otype}(a, T, I, V) : - C[a ofType T]$ and $I$ memberOf $C$</td>
</tr>
<tr>
<td>$\tau_{debug}(\neg B_{user})$</td>
<td>$p_{user}(Ax_ID) : - B_{user}$</td>
</tr>
</tbody>
</table>

Table 2.8: Meta-level axioms for debugging.

<table>
<thead>
<tr>
<th>Debugging Meta-Level Axioms</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) $\Box : - p_{vtype}(a, T, I, V)$</td>
<td></td>
</tr>
<tr>
<td>(2) $\Box : - p_{vmincard}(a, I)$</td>
<td></td>
</tr>
<tr>
<td>(3) $\Box : - p_{vmaxcard}(a, I)$</td>
<td></td>
</tr>
<tr>
<td>(4) $\Box : - p_{vuser}(Ax_ID)$</td>
<td></td>
</tr>
</tbody>
</table>

contains tuples for all occurrences of attribute type violations in $P_3$, identifying the respective attribute $a$, expected type $T$, involved instance $I$ and violating value $V$ for each violation. This set is empty if there are no attribute types violated.

2.5 Reasoning Framework Overview

The design goals of our framework are modularity for the transformation steps and flexibility with respect to the underlying inference engine. The high modularity allows to reuse transformation functionality across different WSML variants and reduces the effort for accomplishing other reasoning tasks. By reducing WSML to simple Datalog constructs and providing a respective object model we have reduced the effort of integrating new reasoners to a minimum.

The presented framework has been fully implemented in Java. It can be downloaded at http://dev1.deri.at/wsml2reasoner. An online demo is available at http://tools.deri.org/wsml/rule-reasoner.

2.5.1 Architecture and Internal Layering

Figure 2.2 shows the internal architecture of the framework as well as the data flow during a prototypical usage scenario. The outer box outlines a WSML reasoner component that allows a user to register WSML ontologies and to pose queries on them. The inner box illustrates the transformation pipeline introduced in Section 2.3 and shows its subsequent steps in a layering scheme.

Registered ontologies go through all the transformation steps, whereas user queries are injected at a later stage, skipping the non-applicable axiomatization and constraint replacement steps. Here, the internal layering scheme allows for an easy reorganization and reuse of the transformation steps on demand, assuring high flexibility and modularity. A good example for this is the constraint replacement transformation $\tau_{debug}$: if included in the pipeline, it produces the rules that activate the debugging features according to Section 2.4; if excluded, the constraints remain in the resulting Datalog program and are mapped to

---

6In fact, the adaptation of the framework to the MINS rule engine took less than a day.
2.5.2 Interface and Integration with Existing Technology

So far we have not detailed on what data structure the framework operates on. One could implement it directly with a parser and compiler framework that generates an abstract syntax tree for WSML which is then directly transformed to the target format (Datalog). Although this would have performance advantages, it would greatly reduce reusability and would make maintenance harder. Our framework is based on an intermediate object model of the language that is provided by the WSMO4J project. WSMO4J performs the task of parsing and validating WSML ontologies and provides the source object model for our translations. In order to enable the usage of different Datalog engines we additionally implemented a simple object model for Datalog that is independent from any particular engine. The Datalog model has objects to represent Literals and Rules, whereas the term structure is directly reused from WSMO4J (respectively WSML). For each reasoner that has to be connected to the Framework a small adapter class has to be written, that is minimally aware of only Literals, Rules and constants (IRIs) and has to translate them to the equivalent within the representation of the reasoner. If a particular reasoner supports additional built-ins and data types translations for this can iteratively be added.

The WSML reasoner framework currently ships with Facades for two built-in reasoners: KAON2 and MINS. The initial development was done with the

\[^{7}\text{http://wsmo4j.sourceforge.net}\]
KAON2 inference engine\(^8\) \cite{Hustadt2004}. As we have seen in Section 2.3.3, datatype reasoning poses the biggest challenge for the Datalog implementation. KAON2 provides a very flexible type system that allows for user-defined datatypes, together with user-defined predicates on these datatypes, including type checking predicates. Therefore, KAON2 meets the identified requirements easily. As a matter of fact, KAON2 already provided most of the required datatypes and predicates out of the box.

The second reasoner that is currently supported by the framework is MINS\(^9\). Whereas KAON2 is the default reasoner for WSML Flight, MINS can be used for the WSML Rule variant that includes function symbols and unsafe rules. For determining which WSML variant a current ontology is in the user of the framework can use the validation facilities built into WSMO4J\(^10\).

### 2.6 Conclusion & Outlook

In this paper, we presented a transformational framework that enables us to perform important reasoning tasks for rule-based WSML. The key features of the framework are: (1) Reasoning via transformation to the widely used Datalog formalism with numerous implemented systems (2) Modular structure of the transformation allows for adaptation and extension of the overall transformation. The single well-defined transformation steps can be reused across various adaptations for different scenarios (e.g. support for debugging of ontologies) (3) Simple integration and exchange of underlying reasoning components. This allows to customize the framework for specific applications: Application developers can choose to use their desired reasoning system providing the capabilities they need. If an application only uses a less expressive WSML variant, then a more specialized reasoning component tailored towards this language can be used, e.g. to ensure performance and scalability. Such a decision can even be change during runtime (4) Support for debugging of ontologies independent of the underlying reasoning system. This feature can be dynamically added and removed from any reasoning component.

Eventually, the available implementation of the system based on Datalog components such as KAON2 and MINS represent the first available reasoning system for WSML. The presented framework proved to be a flexible and effective way to build reasoners for WSML based on existing, well-designed and efficient systems in a short period of time.

Currently, the framework and our implementation that we discussed in this section focuses on WSML Core, Flight and Rule. However, efforts are ongoing to extend this into the direction of WSML DL and WSML Full: we are working on extending the transformations to allow for transformations to disjunctive datalog and disjunctive logic programs (including default negation) too. The KAON2 natively system already supports disjunctive datalog with stratified default negation and thus can be used for reasoning with WSML DL ontologies, even if they are extended by WSML-Flight-like rules. The DLV system \cite{Citrigno1997} (implementing disjunctive datalog under the stable model semantics) can be used for reasoning the same purpose. Furthermore, we plan to integrate the KRHyper system \cite{Wernhard2003}, which allows reasoning with disjunctive logic programs with stratified default negation. This would then allow to reason with WSML DL, WSML Core, WSML Flight and WSML Rule.

---

\(^8\)KAON2 is available for download from [http://kaon2.semanticweb.org](http://kaon2.semanticweb.org)

\(^9\)http://dev1.deri.at/mins/

\(^10\)A demo of this feature is available at [http://tools.deri.org/wsml/validator](http://tools.deri.org/wsml/validator)
(and combinations) in an integrate manner in the case of stratified negation and safe rules. This way, we expect to be able to support significant parts of WSML Full. Finally, an additional translation specific to WSML-DL ontologies to widely used description logic (DL) system APIs (e.g. DIG Bechhofer et al., 2003b) to support efficient reasoning with WSML-DL based on state-of-the-art DL systems like Racer Haarslev and Möller, 2001, Pellet11 or Fact++12 is implemented and discussed in Section 3.

12http://owl.man.ac.uk/factplusplus/
3 Reasoning Framework for WSML DL

3.1 OWL DL

OWL DL is one of three species of the Web Ontology Language OWL. It is based on the logical framework of Description Logics. Both, OWL and Description Logics are described in the following sections.

3.1.1 Description Logic

Logic, which can be tracked back to the ancient Greek, is the foundation of knowledge representation. One form is predicate logic (also called first-order logic), which provides sound and complete proof systems (see §3.4).

Description Logics (DLs) are a subset of predicate logic and constitute a family of logic-based knowledge representation formalisms. They evolved from semantic networks and frame systems, which were non-logical approaches, based on the use of graphical interfaces, also called network-based structures. DLs differ from those systems in that they provide a precise semantic characterization of the modeling language.

Description Logics have become a cornerstone of the Semantic Web for its use in the design of ontologies. They are based on concepts (classes) and roles. Concepts are designated by unary predicate symbols and represent classes of objects sharing some common characteristics. Roles are designated by binary predicate symbols and are interpreted as relations between objects. The latter can also be defined as attributes attached to objects. The language is compositional, i.e. the concept descriptions are built by combining different subexpressions using constructors.

In the following sections we provide a survey on the DL family, and on the syntax and semantics of DLs.

3.1.1.1 The DL Family

Description Logics form a family of different logics, distinguished by the set of constructors they provide. Usually each constructor is associated to a different capital letter. The name of a language is composed by the prefix $ALC$ (attributive language), introduced in [Schmidt-Schauss and Smolka, 1991], followed by the letters corresponding to the constructors used in the language. The following overview of DL constructors can be found at [Horrocks, 2005]:

The basic description logic is denoted by $ALC$. It enables the use of the conjunction $\land$, the disjunction $\lor$, the negation $\neg$, the existential quantifier $\exists$ and the universal quantifier $\forall$. The $ALC$ language corresponds to the multi-modal logic $K$.

The letter $S$ is used for $ALC$ with transitive roles ($R^+$). Other extensions are indicated by additional letters, e.g.:

- $H$ for role inclusion axioms (role hierarchy)
- $O$ for nominals
- $I$ for inverse roles
- $N$ for number restrictions (of form $\leq nR$, $\geq nR$)
• $Q$ for qualified number restrictions (of form $\leq nR.C$, $\geq nR.C$)

E.g., the DL $SHOIQ$ describes the basic description logic extended with transitive roles ($S$), inverse roles ($I$), role hierarchies ($H$), nominals ($O$) and qualified number restrictions ($Q$).

The DLs have been extended in many other direction, e.g. concrete domains or fixpoints. The extensions are not further explained in this thesis. More detailed information about DLs in general, and about the most known extensions can be found in [Baader et al., 2003].

### 3.1.1.2 DL Syntax and Semantics

Description Logics semantics are defined by an interpretation $I = (\Delta^I, \cdot^I)$. $\Delta^I$ is the domain of the interpretation (a non-empty set) and $\cdot^I$ is the interpretation function. The latter assigns meaning to non-logical symbols: it maps concept names into subsets of the domain, role names into subsets of the cartesian product of the domain, and individual names into elements of the domain.

See Table 3.1 and Table 3.2 for an excerpt of how the interpretation function $\cdot^I$ extends to concept and role expressions. The tables are inferred from [Tessaris, 2001] and [Baader et al., 2003].

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Semantics</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>$A^I \subseteq \Delta^I$</td>
<td>concept name</td>
</tr>
<tr>
<td>$\top$</td>
<td>$\Delta^I$</td>
<td>top</td>
</tr>
<tr>
<td>$\bot$</td>
<td>$\emptyset$</td>
<td>bottom</td>
</tr>
<tr>
<td>$C \cap D$</td>
<td>$C^I \cap D^I$</td>
<td>conjunction</td>
</tr>
<tr>
<td>$\forall R.C$</td>
<td>${ x \mid \forall y. (x, y) \in R^I \Rightarrow y \in C^I }$</td>
<td>universal quantification</td>
</tr>
<tr>
<td>$\exists R.C$</td>
<td>${ x \mid \exists y. (x, y) \in R^I \land y \in C^I }$</td>
<td>existential quantification ($\exists$)</td>
</tr>
<tr>
<td>$\neg C$</td>
<td>$\Delta^I \setminus C^I$</td>
<td>general negation ($C$)</td>
</tr>
<tr>
<td>$C \cup D$</td>
<td>$C^I \cup D^I$</td>
<td>disjunction ($\cup$)</td>
</tr>
<tr>
<td>$\leq nR$</td>
<td>${ x \mid # { y \mid (x, y) \in R^I } \leq n }$</td>
<td>number restriction ($\leq n$)</td>
</tr>
<tr>
<td>$\geq nR.C$</td>
<td>${ x \mid # { y \mid (x, y) \in R^I } \geq n }$</td>
<td>qualified number restriction ($\geq nR.C$)</td>
</tr>
<tr>
<td>${a_1, \ldots, a_n}$</td>
<td>${a_1^I, \ldots, a_n^I}$</td>
<td>nominals, &quot;one-of&quot; ($O$)</td>
</tr>
</tbody>
</table>

Table 3.1: Syntax and Semantics of concept expression constructors

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Semantics</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$</td>
<td>$P^I \subseteq \Delta^I \times \Delta^I$</td>
<td>role name</td>
</tr>
<tr>
<td>$R \cap Q$</td>
<td>$R^I \cap Q^I$</td>
<td>role conjunction</td>
</tr>
<tr>
<td>$R \cup Q$</td>
<td>$R^I \cup Q^I$</td>
<td>role disjunction</td>
</tr>
<tr>
<td>$R^{-}$</td>
<td>${ (y, x) \mid (x, y) \in R^I }$</td>
<td>inverse role ($I$)</td>
</tr>
<tr>
<td>$R \circ Q$</td>
<td>$R^I \circ Q^I$</td>
<td>role composition</td>
</tr>
<tr>
<td>$R^*$</td>
<td>$\bigcup_{i \geq 0} (R^I)^i$</td>
<td>reflexive transitive closure</td>
</tr>
</tbody>
</table>

Table 3.2: Syntax and Semantics of role expression constructors

### 3.1.1.3 DL Knowledge Base

Within a knowledge base, there is a clear distinction between intensional knowledge (general knowledge about the problem domain) and extensional knowledge (specific to a particular problem). Analogously, the DL knowledge
A DL knowledge base is separated into two components, TBoxes and ABoxes (see Figure 3.1, Horrocks, 2005).

**Figure 3.1: DL knowledge base**

**TBox** A TBox contains the terminological knowledge of a knowledge base. The basic form of declarations in a TBox are concept definitions, where new concepts are defined in terms of previously defined concepts. Such a concept definition describes general properties of a concept.

Two sorts of axioms describe the structure of a domain in a TBox:

- **definition axioms** introduce macros/names for concepts. The left-hand side of a definition can only be an atomic concept name. Example: \( \text{Woman} \equiv \text{Human} \cap \text{Female} \), denoting that a Woman is equivalent to the intersection of Human and Female.

- **(general) inclusion axioms** (GCI, General Concept Inclusion) assert subsumption relations. Both sides of the definition can be complex concept expressions. Example: \( \exists \text{hasChild}.\text{Female} \sqsubseteq \exists \text{hasChild}.\text{Human} \), denoting that the concept of having at least one female child is a subconcept of having at least one human child.

Most of the early DL systems did not allow TBoxes to contain terminological cycles. But as recursive definitions are very common in modeling application domains, modern systems all provide unrestricted support for cyclic concept definitions.

The basic task in constructing a terminology is classification, which amounts to placing a new concept expression in the proper place in a taxonomic hierarchy of concepts (see Section 3.4.2).

**ABox** The ABox contains assertional knowledge (knowledge about the individuals of a domain). Individuals are introduced and the ABox asserts their properties using concept definitions.

Two sorts of axioms describe concrete situations in an ABox:

- **concept assertions** Example: \( \text{Mary} : (\text{Woman} \cap \exists \text{hasChild}.\text{Female}) \), denoting that Mary is a Woman and has at least one female child.
• **role assertions** Example: `< Mary, Jack >: hasChild`, denoting that Jack is the child of Mary.

When the used Description Logic is extended with nominals, ABox assertions can as well be described as TBox concept subsumptions: \( a : C \) equivalent to \( \{ a \} \sqsubseteq C \).

DL often adopts to the Unique Name Assumption, what means that different names always denote different individuals.

DL knowledge bases adopt an open world semantics. The open world assumption entails that the given information can be incomplete. Thus, what we cannot prove, must not necessarily be wrong. Analogously a closed world assumption would entail that the information from a knowledge base is regarded as complete. This means that everything, that cannot be proven from the available information is regarded as false.

Example: Given a knowledge base that contains the axioms \( \text{Jack} : \text{Man}, \text{Bob} : \text{Man} \) and \(< \text{Jack}, \text{Bob} >: \text{hasChild} \). The query "are all of Jack’s children male?" results to "yes", if the knowledge base adopts a closed world semantics, and to "unknown" under an open world semantics, as there is no information available that tells us if Bos is the only child of Jack. So we do not know whether there is some more information missing about Jack and eventual other children.

### Reasoning
A knowledge representations system based on DLs is not only useful for storing concept definitions and assertions. It allows specific kinds of reasoning over the contained knowledge. More information about DL reasoning tasks can be found at 3.4.

### 3.1.2 Web Ontology Language (OWL)

The OWL Web Ontology Language is a formal language for representing ontologies in the Semantic Web. Ontologies can be defined as the information about categories of objects and how they are interrelated. OWL can not only represent objects, but also information about these objects (RDF annotation properties).

OWL, which today is a World Wide Web Consortium (W3C) standard, was developed by the W3C Web Ontology Working Group. The language is influenced by representation languages as XML and RDF, OIL and DAML+OIL, as well as by Description Logics and frames. This diversity of influences is due to the multiplicity of requirements for OWL, defined in [Antoniou and van Harmelen, 2004] as:

- a well-defined syntax - OWL has the same kind of syntax than RDF and RDFS. A well-defined syntax allows the machine-processing of information.

- a formal semantics - A formal semantics describes the exact meaning of knowledge, what allows reasoning over this knowledge.

- convenience of expression

- efficient reasoning support - Reasoning allows one, e.g., to check the consistency of an ontology and a knowledge base, or/and to build a classification of the ontology objects.

- sufficient expressive power - In general, the more expressive a language is, the more inefficient and difficult the reasoning support becomes.
OWL uses URI references as names (analogously to RDF). Commonly qualified names are used as shorthands for URI references in OWL, using for example the qualified name owl:Thing for the URI reference `http://www.w3.org/2002/07/owl#Thing`.

The semantics of OWL adopt an open-world assumption: Even though a statement cannot be proved to be true, it cannot be assumed false. This assumption is correct in regard of the huge and only partially knowable web. [Horrocks et al., 2003] states that two of the three species of OWL, OWL DL and OWL Lite, can be viewed as very expressive Description Logics. Accordingly, an OWL ontology is equivalent to a Description Logic Knowledge Base (Tbox + Abox).

OWL provides four different syntaxes:

- an RDF-XML based syntax,
- an XML based syntax that does not follow the RDF syntax,
- an abstract syntax, which is much easier to read by human users,
- a graphic syntax, based on the conventions of the Unified Modeling Language (UML).

The following sections provide an overview of the three OWL species and a more precise description of OWL DL. More complete and detailed information about OWL can be found at [McGuinness and van Harmelen, 2004], [Smith et al., 2004], [Bechhofer et al., 2004] and [Patel-Schneider et al., 2004].

Furthermore, Qualified Cardinality Restrictions, which are supported in DL and in WSML, but not in OWL, will be discussed.

### 3.1.2.1 Three Species of OWL

The diversity of requirements for the Web Ontology Language mentioned above, was also responsible for the definition of the three species of OWL. Each one fulfills different aspects of the set of requirements.

**OWL Full** OWL Full is the most expressive of the OWL species. It uses all OWL language primitives and is fully upward-compatible with RDF. It provides all metamodeling facilities of RDF Schema (e.g. defining classes of classes, attaching properties to classes). This means that any legal RDF document is also a valid OWL Full document.

Unfortunately OWL Full has one big disadvantage, which is that it is undecidable, so that there is no complete reasoning support for it (in the sense that reasoning over OWL Full does not always terminate).

**OWL DL** OWL DL (short for Description Logic) is a sublanguage of OWL Full with restricted use of OWL and RDF constructors. It combines a maximum expressiveness with computational completeness and decidability. OWL DL (as well as OWL Lite) corresponds to Description Logics, and allows, unlike OWL Full, both complete and sound reasoning. The underlying DL is $SHOIN(D)$ (the $D$ stands for datatypes).

The fact that OWL DL is a sublanguage of OWL Full entrains that every legal OWL DL document is also a legal OWL Full document. The disadvantage of OWL DL is, that it is not upward-compatible with RDF. Although every valid OWL DL document is a valid RDF document, the vice-versa is not generally true.
OWL Lite  OWL Lite has a still more restricted expressivity than OWL DL. This makes the language easy to learn and easy to implement. It is especially useful for people who need a classification hierarchy and only simple constraints.

OWL Lite being a sublanguage of OWL DL, every valid OWL Lite document is also a valid OWL DL document. The underlying DL of OWL Lite is $SHIF(D)$.

The following section will introduce the OWL DL language. The examples will be presented in the OWL abstract syntax. The tables in Section 3.1.3 provide a quick overview of OWL DL constructors and axioms and the corresponding DL syntax.

### 3.1.2.2 The OWL DL Language

Whereas in OWL Full all OWL language constructors can be used, there exist some restrictions for OWL DL. In OWL DL all resources must explicitly be defined. This means that, e.g., a class cannot only be specified as superclass by the rdfs:subClassOf statement, but must be explicitly stated as class.

Another constraint concerns the vocabulary partitioning. Any resource can only be a class, a data type, an individual, etc., and not more than one of these. For example, a class cannot at the same time be an individual.

#### Ontologies

An OWL ontology contains an rdf:RDF element as root element, which also specifies a number of namespaces. The owl:Ontology element contains comments, version control and the import of other ontologies.

For example:

```xml
Ontology(
  Annotation(rdfs:comment "An example OWL ontology")
  Annotation(rdfs:label "University Ontology")
)
```

#### Classes

Classes are defined by an owl:Class element. OWL classes can be defined to be subclasses of other classes, to be disjoint from other classes or to be equivalent to other classes. OWL provides also an empty class, owl:Nothing, and a most general class, owl:Thing.

Examples for OWL class elements:

```xml
Class( assistantProfessor partial academicStaffMember)
Class( professor partial )
DisjointClasses( professor assistantProfessor )
```

#### Properties

There are two kinds of properties in OWL: Object Properties, which relate objects to objects and Datatype Properties, which relate objects to datatype values (OWL has no own predefined datatypes but allows the use of XML Schema datatypes [Biron and Malhotra, 2004]). Properties can be related to inverse properties or can be defined to be equivalent or inverse to other properties, or to be subproperties of other properties.

Some OWL property elements examples:

```xml
ObjectProperty(isTaughtBy
  inverse (teaches) domain(course) range(academicStaffMember))
SubPropertyOf(isTaughtBy involves)
EquivalentProperties( lecturesIn teaches)
DatatypeProperty(age range(xsd:nonNegativeInteger))
```

#### Special Properties

Special properties designate properties of property elements. Two rather self-explaining properties are owl:TransitiveProperty and owl:SymmetricProperty. owl:FunctionalProperty defines a property that has at most one value for each object and owl:InverseFunctionalProperty defines a property for which two different objects cannot have the same value.
OWL DL restricts the use of inverse of, inverse functional, symmetric or transitive features on properties: they can only be applied on Object Properties, not on Datatype Properties.

An Example for an OWL Special Property:

ObjectProperty(hasSameGradeAs Transitive Symmetric domain(student) range(student))

Property Restrictions A property restriction is a special kind of class description. It describes an anonymous class, namely a class of all individuals that satisfy certain conditions. This anonymous class is not explicitly stated as OWL class. OWL distinguishes two kinds of property restrictions: value constraints and cardinality constraints.

In combination with owl:onProperty, the elements owl:allValuesFrom, owl:someValuesFrom and owl:hasValue allow to build some value restrictions. owl:hasValue declares a specific value that the property specified by owl:onProperty must have. owl:allValuesFrom is a universal quantification in terms of logic, whereas owl:someValuesFrom is an existential quantification in terms of logic.

Cardinality restrictions put constraints on the number of values a property can take. A precise number can be specified by using owl:cardinality. Minimum and maximum cardinalities are defined by owl:minCardinality and owl:maxCardinality. OWL DL does not allow transitive cardinality restrictions, so that cardinality restrictions cannot be placed on transitive properties or their subproperties.

OWL DL does not support qualified cardinality restrictions, although these are representable in DL (see Section 3.1.2.3).

OWL property restrictions can be like the following:

Class(course partial restriction (isTaughtBy minCardinality(1)))
Class(academicStaffMember partial restriction (teaches someValuesFrom(undergraduateCourse)))
Class(postgraduateCourse partial restriction (isTaughtBy allValuesFrom(Professor)))
Class(semanticWebCourse partial restriction (isTaughtBy hasValue(PeterPan)))

Boolean Combinations OWL allows the definition of the Boolean combinations union, intersection and complement. The complement statement can be used to designate the disjointness of two classes. The same meaning could thus be expressed by using disjoint classes.

For example:

Class( peopleAtUniversity complete unionOf(staffMember student))
Class(adminStaff complete intersectionOf (staffMember complementOf(unionOf(faculty techSupportStaff))))

Instances Instances are declared the same way as in RDF, through RDF description and typing.

OWL does not adopt the Unique Name Assumption (UNA). I.e. if two individuals (or classes or properties) have different names, we may still derive that they must be the same. To designate two individuals as different, we need to explicitly state this by a owl:differentFrom statement. OWL also provides a possibility to assert the pairwise inequality of all individuals in a given list.

Example for OWL instances:

Individual (PeterPan type(academicStaffMember))
Individual (SemanticWeb type(course) value(isTaughtBy PeterPan))
DifferentIndividuals (PeterPan CaptainHook)

Data Types As already mentioned earlier, OWL, having no own predefined datatypes, allows the use of XML Schema datatypes [Biron and Malhotra, 2004].
In addition, OWL provides an enumerated datatype (owl:oneOf), which is a construct for defining a range of data values.

The OWL reference Bechhofer et al., 2004 lists in detail all the datatypes that can be used.

An example for an enumerated class is:

```xml
EnumeratedClass(weekdays Monday Tuesday Wednesday Thursday Friday Saturday Sunday)
```

Anonymous Classes OWL DL restricts the use of anonymous classes (explicitly defined anonymous classes, not property restrictions). They are only allowed as domain and range of the owl:equivalentClass or the owl:disjointWith statements, and as range of the rdfs:subClassOf statement.

Facts (axioms) about individual equality and difference must be about named individuals.

### 3.1.2.3 Qualified Cardinality Restrictions

Cardinality restrictions are used to constrain the number of values of a particular property, irrespective of the value type. If we also want to specify the values to be of a particular type, we need "qualified cardinality restrictions" (QCR). The syntax and semantics of such qualified cardinality (number) restrictions in Description Logics are shown in Table 3.1.

One famous example for QCRs in the human anatomy use case is mentioned by Alain Rector in Rector, 2003: "The normal hand has exactly five fingers of which one is a thumb". The fact that a hand has exactly five fingers is easy to describe via a simple cardinality constraint:

```xml
Class(NormalHand
    restriction (hasFinger cardinality (5)))
```

But without QCRs we cannot describe that one of these fingers must be a thumb.

The Web Ontology Working Group has postponed the issue of full representation of QCRs, but has already proposed an OWL representation for them Schreiber, 2003 and discussed a possible workaround that Alain Rector described in Rector, 2003.

A W3C working draft Rector and Schreiber, 2006 discusses a partial workaround within the OWL standard and a non endorsed extension of OWL, that allows to express QCRs correctly.

**Use owl:someValuesFrom** The construct owl:someValuesFrom is equivalent to a QCR with a minimum cardinality of 1. This means that the specified property should have at least one value of this type.

E.g., an Italian dinner should contain at least one antipasto:

```xml
Class( ItalianDinner partial
    restriction (hasCourse someValuesFrom(AntiPasto)))
```

**Workaround using rdfs:subPropertyOf** The idea of the workaround is to express QCRs by having extra subproperties for each component whose number is to be specified. So we introduce a subProperty for the main property and then introduce an unqualified cardinality restriction on that subProperty.

Using this workaround, the "normal hand" example from human anatomy use case, mentioned before, would be represented by:
Unfortunately this workaround is not complete (see [Rector and Schreiber, 2006]) and, although it may be ok for simple cases, it can lead to vast and cumbersome numbers of subProperties.

**Use a non-endorsed OWL extension** Qualified Restrictions resemble regular restrictions but contain one extra triple in the RDF representation and an extra argument in the abstract syntax. This statement would be valuesFrom, which points to the value type being restricted.

Using this extension, the normal hand example can be described by:

```xml
Class(NormalHand partial
  restriction (hasFinger cardinality (5))
  qualifiedRestriction (hasFinger valuesFrom (Thumb) cardinality(1)))
```

This representation is legal in OWL Full, but the semantics will only be treated correctly by parsers and classifiers that support QCRs (the four Description Logic parsers presented in Section 3.5 all support QCRs).

According to [Rector and Schreiber, 2006] it is not unlikely that this extension will be incorporated in a future version of OWL.

As well as DLs, the Web Service Modeling Language WSML (3.2) contains Qualified Cardinality Restrictions. Section 3.3.2 explains the mapping from WSML DL to OWL DL concerning QCRs.

### 3.1.3 OWL DL vs. $SHOIN(D)$

The following tables, illustrating the relationship between the OWL DL and the DL syntax, are inferred from [Horrocks, 2006] and [Horrocks et al., 2003].

Table 3.3 shows the OWL DL constructors, the corresponding DL syntax, and an illustrating example. In the table, $C$ is a concept (class), $P$ is a role (property) and $x$ is an individual name.

Table 3.4 shows the OWL ontology axioms, their corresponding DL syntax and some more illustrating examples.

<table>
<thead>
<tr>
<th>OWL Constructor</th>
<th>DL Syntax</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>intersectionOf</td>
<td>$C_1 \cap \ldots \cap C_n$</td>
<td>$\text{Human} \cap \text{Female}$</td>
</tr>
<tr>
<td>unionOf</td>
<td>$C_1 \cup \ldots \cup C_n$</td>
<td>$\text{Male} \cup \text{Female}$</td>
</tr>
<tr>
<td>complementOf</td>
<td>$\neg C$</td>
<td>$\neg \text{Female}$</td>
</tr>
<tr>
<td>oneOf</td>
<td>${x_1} \sqcup \ldots \sqcup {x_n}$</td>
<td>${\text{John}} \sqcup {\text{Mary}}$</td>
</tr>
<tr>
<td>allValuesFrom</td>
<td>$\forall P . C$</td>
<td>$\forall \text{hasChild} . \text{Female}$</td>
</tr>
<tr>
<td>someValuesFrom</td>
<td>$\exists P . C$</td>
<td>$\exists \text{hasChild} . \text{Male}$</td>
</tr>
<tr>
<td>hasValue</td>
<td>$R : o$</td>
<td>$\text{hasChild} : \text{Bob}$</td>
</tr>
<tr>
<td>maxCardinality</td>
<td>$\leq n P$</td>
<td>$\leq 1 \text{hasChild}$</td>
</tr>
<tr>
<td>minCardinality</td>
<td>$\geq n P$</td>
<td>$\geq 2 \text{hasChild}$</td>
</tr>
</tbody>
</table>

Table 3.3: OWL DL descriptions - DL syntax
### OWL DL Axioms - DL Syntax

<table>
<thead>
<tr>
<th>OWL Axiom</th>
<th>DL Syntax</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>subClassOf</td>
<td>$C_1 \sqsubseteq C_2$</td>
<td>$Mother \sqsubseteq Female \sqcap Parent$</td>
</tr>
<tr>
<td>equivalentClass</td>
<td>$C_1 \equiv C_2$</td>
<td>$Female \equiv Human \sqcap Male$</td>
</tr>
<tr>
<td>subPropertyOf</td>
<td>$P_1 \sqsubseteq P_2$</td>
<td>$hasDaughter \sqsubseteq hasChild$</td>
</tr>
<tr>
<td>equivalentProperty</td>
<td>$P_1 \equiv P_2$</td>
<td>$teaches \equiv lecturesIn$</td>
</tr>
<tr>
<td>transitiveProperty</td>
<td>$P^+ \sqsubseteq P$</td>
<td>$ancestor^+ \sqsubseteq ancestor$</td>
</tr>
<tr>
<td>type</td>
<td>$a : C$</td>
<td>$Mary : Mother$</td>
</tr>
<tr>
<td>property</td>
<td>$&lt; a, b &gt; : R$</td>
<td>$&lt; Mary, John &gt; : hasChild$</td>
</tr>
</tbody>
</table>

Table 3.4: OWL DL axioms - DL syntax

## 3.2 Web Service Modeling Language (WSML)

Semantically enriched Web Services are called Semantic Web Services and enable the access of Web services for software agents without, or with a minimal, human interaction.

A language for describing such Semantic Web Services is the Web Service Modeling Ontology WSMO [Roman et al., 2004]. Providing a conceptual model for the description of various aspects related to Semantic Web Services, it distinguishes four top-level elements:

- **Ontologies**: Ontologies define an agreed common terminology by providing concepts and relationships between the concepts. In order to capture semantic properties of relations and concepts, an ontology generally also provides a set of axioms, which are expressions in some logical language.

- **Goals**: They describe the functionality and interaction style in the requester view. The requester describes what would potentially satisfy his desires.

- **Web Service descriptions**: They specify the functionality and the means of interacting with the Web Service in order to achieve the requested functionality.

- **Mediators**: Mediators have different jobs: they can import ontologies, resolve possible representation mismatches between ontologies, link two goals, two Web Services or Web Services to goals.

The Web Service Modeling Language WSML [de Bruijn et al., 2005b] is a family of formal Web languages based on the conceptual model of WSMO. Its Semantic is based on Description Logics [Baader et al., 2003], Logic Programming [Lloyd, 1987] and First-Order Logic [Fitting, 1996b], with influences from F-Logic [Kifer et al., 1990] and frame-based representation systems.

Conforming to the different influences, there exist five variants of WSML:

- **WSML-Core** is based on the intersection of the $SHIQ$ DL and Horn Logic, itself based on Description Logic Programs [Grosol et al., 2003]. It has the least expressive power of all the WSML variants.

- **WSML-DL** is an extension of WSML-Core and captures the expressive Description Logic $SHI(Q(D))$. According to [de Bruijn et al., 2005b] it covers that part of OWL DL which is efficiently implementable. It will be presented in more detail in Section 3.2.1.
• **WSML-Flight** is (like WSML-DL) an extension of WSML-Core that provides a powerful rule language.

• **WSML-Rule** extends WSML-Flight with further features from Logic Programming.

• **WSML-Full** unifies all WSML variants under a First-order umbrella. Its semantics is currently an open research issue.

WSML makes a clear distinction between the modeling of conceptual elements (Ontologies, Web Services, Goals and Mediators) and the specification of logical definitions. Therefore the WSML syntax is split in two parts: the conceptual syntax and the logical expression syntax.

The following sections provide an overview of the WSML-DL syntax and the WSML-DL reasoning requirements.

### 3.2.1 WSML-DL Syntax

A WSML ontology distinguishes two parts. The first contains meta-information about the specification, the second consists of elements such as concepts, attributes, relations, etc.

The remainder of this section provides an insight to the WSML Syntax Basics, the meta-information elements, the ontology elements and the logical expression specification. The provided examples are taken partly from [de Bruijn et al., 2005b], where a more detailed description of the WSML-DL syntax can be found.

#### 3.2.1.1 WSML-DL Syntax Basics

Some syntax basics, as the use of namespaces, identifiers and datatypes are explained hereafter.

**Namespaces** WSML adopts the namespace mechanism of RDF. A namespace can be seen as part of an IRI. Namespaces can be used to syntactically distinguish elements of multiple WSML specifications and, more general, resources on the Web. The WSML keywords belong to the namespace ‘http://www.wsmo.org/wsml/wsml-syntax#’ (commonly abbreviated as ‘wsml’).

**Identifiers** An identifier in WSML is either a data value, an IRI, or an anonymous ID.

- **Datatypes** - Primitive data values in WSML are strings, integers and decimals. They can be written using syntactical shortcuts: Data values of type string can be written between double quotes “”, integer and decimal values can be written simply as such (1 and 0.1 as shortcuts for integer("1") and decimal("0.1")).

  These datatypes can be used to build structured datatypes, using datatype wrappers. All datatypes correspond to XML Schema datatypes [Biron and Malhotra, 2004]. [de Bruijn et al., 2005b] Appendix C gives a detailed overview of datatypes in WSML.

- **IRIs** - The Internationalized Resource Identifier (IRI) mechanism provides a way to identify resources (e.g. on the web) [Duerst and Suignard, 2005b]. The IRI proposed standard is a successor to the popular URI standard. In fact, every URI can be mapped to an IRI. An IRI can be abbreviated to an
sQName (short for serialized QName, RDF), what enhances legibility. sQ-Names consist of two parts: the namespace prefix and the local part (e.g. dc#title stands for ('http://purl.org/dc/elements/1.1#title')).

- Anonymous identifiers - Anonymous identifiers can be used whenever the concrete identifier is not relevant and when we require the identifier to be different from existing ones. An anonymous identifier represents an IRI which is globally unique.

WSML distinguishes unnumbered anonymous identifiers and numbered anonymous identifiers. The use of numbered anonymous identifiers is however limited to logical expressions and can therefore not be used to denote objects in the conceptual syntax.

Note that the use of identifiers in the specification of WSML elements is optional. If no identifier is specified, the identifier of an ontology is assumed to be the same as the locator of the specification, whereas for WSML elements as concepts, relations, etc. unnumbered anonymous identifiers are used.

3.2.1.2 WSML-DL Meta-information Elements

The meta-information part of a WSML specification is strictly ordered. It consists of the WSML variant identification, namespace references, non-functional properties (annotations), the import of ontologies, etc.

**WSML Variant** Every WSML specification should start with the `wsmlVariant` keyword, followed by an identifier for the WSML variant. For WSML-DL this is `http://www.wsmo.org/wsml/wsml-syntax/wsml-dl`. The specification of the `wsmlVariant` is optional, but recommended, as it facilitates the work for tools (by recognizing the intention of the author and react to it).

**Namespaces** Next comes an optional block for namespace references, preceded by the `namespace` keyword. Each namespace reference, except the default namespace, consists of a namespace prefix and the IRI which identifies the namespace. The default namespace does not contain a namespace prefix.

**Header** The WSML header may contain non-functional properties, import ontologies and use mediators.

Non-functional properties in WSML are not part of the logical language.

**Example** The following is an example prologue of a WSML-DL file:

```xml
wsmlVariant "http://www.wsmo.org/wsml/wsml-syntax/wsml-dl"
namespace { "http://www.example.org/ontologies/example#", dc: "http://purl.org/dc/elements/1.1#" }
```

3.2.1.3 WSML-DL Conceptual Syntax - Ontology Specification

A WSML ontology specification is designated by the `ontology` keyword, optionally followed by an IRI as ontology identifier. Such an ontology specification may look like:

```
ontology "http://www.example.org/ontologies/example"
```

An ontology specification may contain concepts, relations, instances, relation instances and axioms.
Concepts  In ontologies, concepts form the basic terminology of the domain of discourse. A concept may have instances and associated attributes.

In WSML, a concept definition starts with the keyword `concept`, followed by an identifier. This concept can contain attribute definitions and can be defined as subconcept of another concept. In this case, a concept inherits all attribute definitions of its superconcept.

There are two sorts of attribute definitions that a concept may contain: constraining definitions with the keyword `ofType` and inferring definitions with the keyword `impliesType`. Constraining attribute definitions may only be used for datatypes ranges. This means that attribute definitions of the form `A ofType D` are only allowed if `D` is a datatype identifier. Inferring attribute definitions are similar to range restrictions on properties in RDFS [Brickley and Guha, 2004b] and OWL [Bechhofer et al., 2004].

Example:
```plaintext
concept Human
  hasChild impliesType Human
  hasName ofType String
concept Man subConceptOf { Human }
```

Relations  A relation definition starts with the keyword `relation`, followed by an identifier. A relation can be defined as subrelation of another relation.

WSML-DL allows only the specification of binary relations. The parameters of a relation are strictly ordered. The usage of `impliesType` and `ofType` for parameter type definitions correspond with the usage in attribute definitions (the `ofType` keyword is only allowed in combination with a datatype and only the second parameter may have a datatype at its range).

Example:
```plaintext
relation ageOfHuman (impliesType Human, ofType Integer)
```

Instances  A concept may have an arbitrary number of instances associated to it. Instances can either be explicitly defined in an ontology or they can exist outside the ontology in a private database.

An instance starts with the `instance` keyword, followed by an identifier, the `memberOf` keyword and the name of the concept to which the instance belongs (all these definitions being optional). The instance definition can be followed by the attribute values associated with the instance.

Example:
```plaintext
instance Mary memberOf { Human }
  hasName hasValue "Maria Smith"
  hasBirthdate hasValue date(1976,08,16)
```

Relation Instances  A relation instance starts with the `relationInstance` keyword, followed by an identifier, the `memberOf` keyword and the name of the relation to which the relation instance belongs. This definition is followed by the values of the parameters associated with the relation instance.

Relation instances are only allowed for binary relations. Both values of the relation have to be specified and have to correspond to its signature.

Example:
```plaintext
relationInstance ageOfHuman(Mary, 31)
```

Axioms  An axiom definition starts with the keyword `axiom`, followed by an identifier, the `definedBy` keyword and a logical expression.

Such axioms can be used to refine the definitions already given in the conceptual syntax, e.g. the subconcept and attribute definitions of concepts. By
defining respective axioms one can define global transitivity, symmetricity and
inversity of attributes, just like in DLs or OWL.

Example:

```
axiom maryDefinition
  definedBy
  Mary[ageOfHuman hasValue 31].
```

The logical expression syntax is explained in the following Section 3.2.1.4.

### 3.2.1.4 WSML-DL Logical Expression Syntax

The form of the WSML-DL logical expressions and their expressiveness is
based on the Description Logic SHIQ(D).

The WSML-DL logical expression syntax has constants, variables, predicates
and logical connectives, which all come from First-order Logic style. F-Logic
based extensions help to model concepts, attributes, attribute definitions, sub-
concept and membership relationships.

Variables in WSML-DL start with a question mark, followed by an arbi-
trary number of alphanumeric characters (e.g. ?x). Terms are either identifiers,
variables or constructed terms.

An atom in WSML-DL is a predicate symbol with one or two terms as
arguments. WSML has a special kind of atoms, called molecules. Those are
used to capture information about concepts, instances, attributes and attribute
values. There are two types of molecules:

- **isa molecule** - An isa molecule is a concept membership molecule of the
  form $A$ memberOf $B$ (also called a-molecule) or a subconcept molecule
  of the form $A$ subConceptOf $B$ with $A$ and $B$ arbitrary terms.

- **object molecule** - An object molecule is an attribute value expression
  of the form $A$[B hasValue $C$] or $B(A, C)$ (also called b-molecule), a
  constraining attribute expression of the form $A$[B ofType $C$] ($C$ must
  be a datatype, corresponding to the usage in attribute definitions), or an
  inferring attribute expression of the form $A$[B impliesType $C$], with $A,$
  $B$ and $C$ arbitrary terms.

These molecules build the set of atomic formulae in WSML-DL. Using first-
order connectives, one can combine the atomic formulae to descriptions and
formulae.

WSML-DL Descriptions are built using the connectives: unary negation
operator neg and binary operators for conjunction and and disjunction or.
Variables may be universally quantified using forall or existentially quantified
using exists.

The set of formulae in WSML-DL contains any atomic formulae and the
combination of descriptions, using the connectives right implication implies,
left implication impliedBy and dual implication equivalent.

How exactly the molecules can be combined to build descriptions and
formulae, can be seen in detail in [de Bruijn et al., 2005b], chapter 3.6.

Examples of WSML-DL logical expressions:

```
// the concepts Man and Woman are disjoint
\forall x \forall y (x memberOf Man \implies \neg y memberOf Woman).
// every Human has a father, which is a Human and every father is a Human
\forall x memberOf Human
  implies
  \exists y (\forall x (father hasValue y) \land \forall y (memberOf Human).
and
\forall y (\forall x (father hasValue y) \implies \forall y (memberOf Human).
// globally symmetric attribute / relation
\forall x (p hasValue y) \implies \forall y (p hasValue x).
```
3.2.2 WSML-DL vs. SHIQ(D)

Table 3.5 illustrates the relationship between the WSML-DL semantics, the Description Logics syntax and the OWL DL syntax. The table is inferred by [de Bruijn et al., 2005b], [Volz, 2004] and [Borgida, 1996].

In the table, \( id \) can be any identifier, \( dt \) is a datatype identifier, \( X \) can be either a variable or an identifier and \( Y \) is a variable.

Table 3.5: OWL DL descriptions - DL syntax

<table>
<thead>
<tr>
<th>WSML-DL</th>
<th>DL Syntax</th>
<th>OWL DL</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \tau( \text{expr \ impliedBy \ rexpr} ) )</td>
<td>( \text{rexpr} \subseteq \text{expr} )</td>
<td>subClassOf</td>
</tr>
<tr>
<td>( \tau( \text{expr \ or \ rexpr} ) )</td>
<td>( \text{expr} \cup \text{rexpr} )</td>
<td>unionOf</td>
</tr>
<tr>
<td>( \tau( \text{expr \ and \ rexpr} ) )</td>
<td>( \text{expr} \cap \text{rexpr} )</td>
<td>intersectionOf</td>
</tr>
<tr>
<td>( \tau(\neg \text{expr}) )</td>
<td>( \neg \text{expr} )</td>
<td>complementOf</td>
</tr>
<tr>
<td>( \tau(\forall Y_1, \ldots, Y_n \text{ expr} ) )</td>
<td>( \forall R.\text{expr} )</td>
<td>allValuesFrom</td>
</tr>
<tr>
<td>( \tau(\exists Y_1, \ldots, Y_n \text{ expr} ) )</td>
<td>( \exists R.\text{expr} )</td>
<td>someValuesFrom</td>
</tr>
<tr>
<td>( \tau(\text{X memberOf} \ id) )</td>
<td>( X : id )</td>
<td>type</td>
</tr>
<tr>
<td>( \tau(\text{id1 subConceptOf} \ id2) )</td>
<td>( \text{id1} \sqsubseteq \text{id2} )</td>
<td>subClassOf</td>
</tr>
<tr>
<td>( \tau(X1[\text{id hasValue} X2]) )</td>
<td>( &lt; X1, X2 &gt; : id )</td>
<td>property</td>
</tr>
<tr>
<td>( \tau(\text{id1}[\text{d2 impliesType} \ id3]) )</td>
<td>( \text{id1} \sqsubseteq \forall \text{id2}.\text{id3} )</td>
<td>subPropertyOf</td>
</tr>
<tr>
<td>( \tau(\text{id1}[\text{d2 ofType} \ dt]) )</td>
<td>( \text{id1} \sqsubseteq \forall \text{id2}.\text{dt3} )</td>
<td>subPropertyOf</td>
</tr>
<tr>
<td>( \tau(\text{id}(X_1, \ldots, X_n)) )</td>
<td>( &lt; X_1, \ldots, X_n &gt; : p )</td>
<td>type</td>
</tr>
<tr>
<td>( \tau(X_1 := X_2) )</td>
<td>( X_1 \equiv X_2 )</td>
<td>equivalentClass</td>
</tr>
</tbody>
</table>

3.3 Mapping WSML-DL to OWL-DL

In this section we provide a mapping from WSML-DL to the OWL DL abstract syntax. The mapping is based on a mapping from WSML-Core to OWL DL, which can be found in [de Bruijn et al., 2005b]. The mapping to OWL DL can be applied to WSML ontologies and logical expressions.

3.3.1 Pre-processing Steps

In order to simplify the translation from WSML-DL to OWL DL, we try to build simpler and less expressions. Therefore we perform the following pre-processing steps:

- Replace all sQNames with full IRIs. For example, “dc#title” is substituted by “http://purl.org/dc/elements/1.1#title”.

• Replace all unnumbered anonymous identifiers by \( http://www.wsmo.org/wsml/wsml-syntax#anonymousId \).

• Rewrite all data term shortcuts ("string" := _string("string"); integer := _integer(integer); decimal := _decimal(decimal)).

• Replace all WSML datatype constructors according to [de Bruijn et al., 2005b, Table C.1] to their corresponding XML Schema datatype.
  For example, _string is changed to \( http://www.w3.org/2001/XMLSchema#string \).

• Replace idlists with single ids (in the case of ofType, impliesType, hasValue, memberOf, subConceptOf and subRelationOf).
  For example, "hasChild hasValue \{Bob, Anna\}" is replaced by "hasChild hasValue \{Bob\}" and "hasChild hasValue \{Anna\}".

• Replace all relations of the form "B(impliesType A, impliesTyp C)" by concept attributes of the form "concept A[impliesTyp C]".
  For example, the relation "relation ageOfHuman(impliesType Human, ofType integer)" is substituted by the concept attribute "concept Human [ageOfHuman ofType integer]".

• Replace all subrelations of the form "B subRelationOf D" by axioms of the form "?x[B hasValue ?y] implies ?x[D hasValue ?y].".
  For example, "relation hasMother subRelationOf hasParent" is replaced by "axiom hasMother_is_hasParent, definedBy, ?x[hasMother hasValue ?y] implies ?x[hasParent hasValue ?y].".

• Replace all relation instances of the form " relationInstance B(A, C)" by attribute values of the form "instance A[hasValue C]".
  For example, the relation instance relationInstance ageOfHuman(Mary, 31) is substituted by the instance and attribute value "instance Mary[ageOfHuman hasValue 31]".

• Replace the remaining conceptual syntax, consisting of concepts and instances, by logical expressions. The transformation is done according to [de Bruijn et al., 2005b] Table 8.1.

• Within logical expressions, the following pre-processing steps are applied:
  - Equivalence and right implication is replaced by left implication.
    For example, "lexpr implies rexpr." is replaced by "rexpr impliedBy lexpr." and "lexpr equivalent rexpr" is substituted by "lexpr impliedBy rexpr and rexpr impliedBy lexpr."
  - Within left implications, left-side conjunctions and right-side disjunctions are eliminated by splitting the expressions. This normalizing step is only applied if the operands of the respective conjunctions or disjunctions do not contain any dependencies from each other.
    For example, the left implication "A and B impliedBy C" is replaced by "(A impliedBy C) and (B impliedBy C) and the left implication "A impliedBy B or C" is replaced by "(A impliedBy B) and (A impliedBy C)".
### 3.3.2 Mapping Table

Table 3.6 contains the mapping between the WSML-DL syntax and the OWL DL abstract syntax. The mapping is described through the mapping function \( \tau \).

The table also indicates a mapping for Qualified Cardinality Restrictions (QCRs). In WSML-DL the QCRs are represented by a combination of WSML-DL descriptions. The mapping to OWL DL is done according the workaround with OWL subproperties, described in Section 3.1.2.3.

Boldfaced words in the table refer to keywords in the WSML language. X and Y are meta-variables and are replaced with actual identifiers and variables during the translation, while DES stands for a WSML-DL description. IRIs are abbreviated by sQNames. The prefix ‘wsml’ stands for ‘http://wsmo.org/wsml/wsml-syntax#’ and ‘owl’ stands for ‘http://www.w3.org/2002/07/owl#’.

<table>
<thead>
<tr>
<th>WSML-DL</th>
<th>OWL-DL</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \tau(\text{ontology id}) )</td>
<td>Ontology(id)</td>
<td>A header can contain nonFunctionalProperties, usesMediator and importsOntology statements. An ontology_element can be a concept, a relation, an instance, a relation instance or an axiom.</td>
</tr>
<tr>
<td>( \cdots )</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>header_n</td>
<td>( \tau(\text{header}_n) )</td>
<td></td>
</tr>
<tr>
<td>ontology_element_1</td>
<td>( \tau(\text{ontology_element}_1) )</td>
<td></td>
</tr>
<tr>
<td>( \cdots )</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>ontology_element_n</td>
<td>( \tau(\text{ontology_element}_n) )</td>
<td></td>
</tr>
</tbody>
</table>

| \( \tau(\text{nonFunctionalProperties}) \) | Annotation(id_1, \( \tau(\text{value}_1) \)) | For non functional properties on the ontology level “Annotation” instead of “annotation” has to be written. |
| \( \cdots \) | ... | |
| id_1 hasValue value_1 | \( \tau(\text{value}_1) \) | |
| \( \cdots \) | ... | |
| id_n hasValue value_n | \( \tau(\text{value}_n) \) | |
| endNonFunctionalProperties | | |

| \( \tau(\text{importsOntology id}) \) | Annotation(owl#import id) | “id” stands for the identifier of a WSML file. |
| \( \tau(\text{usesMediator id}) \) | Annotation(wsml#usesMediator id) | As OWL doesn’t have the concept of a mediator, a wsml#usesMediator annotation is used. |

\[
\tau(\text{datatype id}(x_1, \ldots, x_n)) \rightarrow \text{datatype_id}(x_1, \ldots, x_n) \wedge \tau(\text{datatypes})
\]

\( \tau(\text{datatypes}) \) maps WSML datatypes to XML Schema datatypes, according to [de Bruijn et al., 2005b, Table C.1].

| \( \tau(\text{id}) \) | id | In WSML an IRI is enclosed by ‘‘ and ‘‘, which are omitted in OWL abstract syntax. |

Mapping for axioms

<p>| ( \tau(\text{axiom id log expr nfp}) ) | ( \tau(\text{log expr}) ) | A log expr can be a logical expression like the following. The axiom does not keep its non functional properties. |</p>
<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>τ(id[att_id] impliesType range_id)</td>
<td>Class(id restriction (att_id allValuesFrom range_id) ) ObjectProperty(att_id)</td>
</tr>
<tr>
<td>τ(id[att_id ofType range_id])</td>
<td>Class(id restriction (att_id allValuesFrom range_id) ) DatatypeProperty(att_id)</td>
</tr>
<tr>
<td>τ(id1 subConceptOf id2)</td>
<td>Class(id1 partial id2)</td>
</tr>
<tr>
<td>τ(id[att_id hasValue value])</td>
<td>Individual (id value( att_id τ(value)))</td>
</tr>
<tr>
<td>τ(id1 memberOf id2)</td>
<td>Individual (id1 type(id2))</td>
</tr>
<tr>
<td>τ(?x[att_id hasValue ?y] impliedBy ?y[ att_id hasValue ?z])</td>
<td>SubProperty(att_id att_id2)</td>
</tr>
<tr>
<td>τ(?y[att_id hasValue ?y] impliedBy ?x[ att_id hasValue ?x] and ?y[att_id hasValue ?z])</td>
<td>ObjectProperty(att_id Transitive)</td>
</tr>
<tr>
<td>τ(?x[att_id hasValue ?y] impliedBy ?y[ att_id hasValue ?x])</td>
<td>ObjectProperty(att_id Symmetric)</td>
</tr>
<tr>
<td>τ(?x[att_id hasValue ?y] impliedBy ?y[ att_id2 hasValue ?x])</td>
<td>ObjectProperty(att_id inverseOf(att_id2))</td>
</tr>
<tr>
<td>τ(?x memberOf concept_id2 impliedBy ?x memberOf concept_id)</td>
<td>Class(concept_id partial concept_id2)</td>
</tr>
<tr>
<td>τ(?x memberOf concept_id impliedBy ?x[ att_id hasValue ?y])</td>
<td>ObjectProperty(att_id domain(concept_id) )</td>
</tr>
<tr>
<td>τ(?y memberOf concept_id impliedBy ?x[ att_id hasValue ?y])</td>
<td>ObjectProperty(att_id range(concept_id) )</td>
</tr>
<tr>
<td>τ(DES1 impliedBy DES2)</td>
<td>subClassOf(τ(DES2), τ(DES1) )</td>
</tr>
</tbody>
</table>
If $\tau$ is applied for a non-occurring production no translation has to be made.

### Mapping for descriptions (DES)

<table>
<thead>
<tr>
<th>$\tau()$</th>
<th>Class(id)</th>
<th>Description. Membership molecule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau(\text{memberOf id})$</td>
<td>ObjectProperty( att_id )</td>
<td>Description. Attribute-value molecule</td>
</tr>
<tr>
<td>$\tau(\text{DES}_1 \text{ and } ... \text{ and } \text{DES}_n)$</td>
<td>intersectionOf ($\tau(\text{DES}_1),... ,\tau(\text{DES}_n)$)</td>
<td>Description. Conjunction</td>
</tr>
<tr>
<td>$\tau(\text{DES}_1 \text{ or } ... \text{ or } \text{DES}_n)$</td>
<td>unionOf($\tau(\text{DES}_1),...,\tau(\text{DES}_n)$)</td>
<td>Description. Disjunction</td>
</tr>
<tr>
<td>$\tau(\text{neg DES})$</td>
<td>complementOf($\tau(\text{DES})$)</td>
<td>Description. Negation</td>
</tr>
<tr>
<td>$\tau(\text{forall } ?x (\text{DES impliedBy } ?y [\text{att_id hasValue } ?y]))$</td>
<td>restriction ( att_id allValuesFrom ( $\tau(\text{DES})$))</td>
<td>Description. Universal quantification</td>
</tr>
<tr>
<td>$\tau(\text{exists } ?x (?y[\text{att_id hasValue } ?x] \text{ and } \text{DES}))$</td>
<td>restriction ( att_id someValuesFrom( $\tau(\text{DES})$))</td>
<td>Description. Existential quantification</td>
</tr>
<tr>
<td>$\tau(\text{forall } ?y_1,...,?y_n (?x[\text{att_id hasValue } ?y_1] \text{ and } ... \text{ and } \text{DES} \text{ and } \text{neg(?y}_2) \text{ and } ... \text{ and } \text{neg(?y}_n-1 \text{ :: : } ?y}_n))$</td>
<td>Class($\tau(\text{DES})$) ObjectProperty( att_id ) ObjectProperty( att_id' range($\tau(\text{DES})$)) SubPropertyOf(att_id' att_id) restriction ( att_id minCardinality cardinality</td>
<td>maxCardinality(n) )</td>
</tr>
<tr>
<td>$\tau(\text{forall } ?y_1,...,?y_n (\text{neg(?y}_1 \text{ :: : } ?y}_2) \text{ and } ... \text{ and } \text{neg(?y}_n-1 \text{ :: : } ?y}_n))$</td>
<td>Class($\tau(\text{DES})$) ObjectProperty( att_id ) ObjectProperty( att_id' range($\tau(\text{DES})$)) SubPropertyOf(att_id' att_id) restriction ( att_id minCardinality cardinality</td>
<td>maxCardinality(n) )</td>
</tr>
</tbody>
</table>

Table 3.6: Mapping WSML-DL to OWL DL

### Architecture and Implementation

In the following, we will discuss the approach and architecture of our prototype that allows us to perform reasoning with WSML-DL ontologies. In Section 3.4, we explain various standard and non-standard reasoning tasks that have been considered for Description Logic knowledge bases and that are relevant to ontology reasoning in principle. In Section 3.5, we survey on four powerful state-of-the-art Description Logic reasoners that we aim at integrating in our framework, namely: Fact++, KAON2, Pellet and Racer. We discuss common
interfaces that allow for a uniform integration of the single systems in Section 3.6 and finally show in Section 3.7 how to realize reasoning in WSML-DL on top of this common interface and the available state-of-the-art systems for particular expressive DLs by means of a wrapper component.

3.4 DL Reasoning Tasks

Reasoning describes the task of inferring new (i.e. not explicitly stated) knowledge from a given set of statements (respectively a knowledge base). This is also called logic inferencing.

Related to the Semantic Web, there are two main application fields for reasoning:

- **ontology design** - Checking for class consistency and for (unexpected) implied relationships helps in building and modeling ontologies.
- **ontology deployment** - Checking sets of facts w.r.t. ontologies for consistency and offering answer queries w.r.t ontologies are both useful for the deployment of information from a knowledge base.

In Description Logics, there are different basic reasoning tasks for reasoning with TBoxes or ABoxes. As described in [Baader et al., 2003], the main inference procedures with TBoxes are Concept subsumption and Concept satisfiability. With ABoxes, the main reasoning tasks are ABox consistency and Instance checking. These different reasoning tasks are not independant, often one of them can be reduced to another.

The Semantic Web community focuses on entailment and query answering as the key inference services. Entailment can be reduced to satisfiability, while query answering amounts to compute the result of a query over a database, or an ABox respectively. This is based on working with a database-style conjunctive query language.

For the main DL reasoning tasks, mentioned above, there mostly exist both sound and complete algorithms given a knowledge base. A sound proof procedure for entailment proves only entailed sentences, whereas a complete proof procedure for entailment can prove every entailed sentence. In the usage of knowledge-based systems, it is often necessary to have a guarantee that the inferencing algorithms are sound and complete. Most state-of-the-art reasoners today are based on tableaux-calculi techniques.

Reasoning algorithms are not only evaluated by their effectiveness but also by their complexity. [Baader et al., 2003] talks about the tradeoff between the expressiveness of a representation language and the computational complexity. The more expressive the language is, the harder the reasoning over it is.

More detailed information about reasoning tasks and algorithms in expressive Description Logics and about the reasoning complexity are available at [Baader et al., 2003] and at the official Description Logics page, [http://dl.kr.org/](http://dl.kr.org/).

The following sections deal with the main reasoning tasks, as described above, and offer a quick overview of some main non-standard inference problems.

3.4.1 Knowledge Base Consistency

Checking knowledge base consistency is about ensuring that an ontology does not contain any contradictory facts. It is checked if a given ABox \( \mathcal{A} \) and
3.4.2 Concept Subsumption

Subsumption is usually written as $C \sqsubseteq D$. Determining subsumption is about checking whether the subsumer concept $D$ is more general than the subsumee concept $C$. This means that $C$ must denote a subset of the set denoted by $D$.

Example: $Mother \sqsubseteq Woman$.

Two other relationships between concepts are Equivalence and Disjointness. Both of them can be reduced to subsumption:

- $C$ and $D$ are equivalent $\iff C$ is subsumed by $D$ and $D$ is subsumed by $C$.
- $C$ and $D$ are disjoint $\iff C \cap D$ is subsumed by $\bot$.

Subsumption is also used to compute a subsumption hierarchy (taxonomy) of all named concepts. This helps for classification, which means to place a new concept expression in the proper place in a taxonomic hierarchy of concepts. Classification is a basic task in constructing a terminology.

3.4.3 Concept Satisfiability

Concept satisfiability is the problem of checking whether there exists a model of the knowledge base in which $C$ is interpreted non-empty (has an individual).

So a concept $C$ is satisfiable if and only if there exists an interpretation $I$ such that $C^I \neq \emptyset$. $I$ is then called a model of $C$.

Satisfiability can be reduced to subsumption: $C$ is unsatisfiable $\iff C$ is subsumed by $\bot$. Subsumption can be reduced to satisfiability by: $C$ is subsumed by $D$ $\iff C \cap \neg D$ is unsatisfiable.

Example: $Mother \sqsubseteq Woman \iff Mother \cap \neg Woman$ is unsatisfiable.

3.4.4 ABox Consistency

ABox consistency is the problem of checking whether there is a nonempty model for $A$. In general, it is checked w.r.t. a TBox (see Section 3.4.1).

3.4.5 Instance Checking

Instance checking verifies whether a given individual is an instance of a specified concept. Other reasoning services can be defined in terms of instance checking (e.g. Knowledge Base Consistency (see Section 3.4.1), Realization (see Section 3.4.6.2), Retrieval (see Section 3.4.6.3)). They are described in the following Section 3.4.6 about Non-Standard Inference Problems.

Instance checking itself can be reduced to ABox consistency.

3.4.6 Non-Standard Inference Problems

All DL systems provide the standard inference services described above. According to [Baader et al., 2003], it has turned out however that these services are
not quite sufficient for optimally building and maintaining large DL knowledge bases.

Non-standard reasoning tasks can support the building and maintainence of knowledge bases, as well as the retrieval of information about the knowledge in it. Hereafter some more prominent non-standard inference problems are briefly described.

3.4.6.1 Least Common Subsumer (LCS)

The Least Common Subsumer of two concepts $C$ and $D$ is the minimal concept that subsumes both of them, thus a concept describing the commonalities of $C$ and $D$.

$E$ is the LCS of $C$ and $D$ if:

1. $C \sqsubseteq E$ and $D \sqsubseteq E$
2. for every $F$ with $C \sqsubseteq F$ and $D \sqsubseteq F$, we have $E \sqsubseteq F$.

3.4.6.2 Most Specific Concept (MSC) and Realization

The Most Specific Concept is the least concept description that an individual is an instance of, given the assertions in the knowledge base. The problem of determining the MSC of a given individual is called realization.

$C$ is the MSC of an individual $a$ in an ABox $A$ if:

1. $A \models a : C$
2. for each $D$ with $A \models a : D$, we have $C \sqsubseteq D$

3.4.6.3 Retrieval

Retrieval is about retrieving all instances of a given concept.

3.4.6.4 Unification of Concept Terms

Unification of concept terms extends, according to [Baader and Narendran, 1998], the equivalence problem by allowing to replace certain concept names by concept terms before testing for equivalence. This is necessary, because often testing for equivalence is not sufficient to find out whether, for a given concept term, there already exists another concept term in the knowledge base describing the same notion.

Although the following concept terms are not equivalent, e.g., they are representing the same concept:

- $\text{Woman} \sqcap \forall \text{child.Woman}$
- $\text{Female} \sqcap \text{Human} \sqcap \forall \text{child.}(\text{Female} \sqcap \text{Human})$

The two terms can be made equivalent by replacing the atomic concept $\text{Woman}$ by the concept term $\text{Female} \sqcap \text{Human}$. So those two terms obviously unify.

3.4.6.5 Matching of Concepts

Matching of concepts with variables is a special case of unification. It was initially meant to help to discard unimportant aspects of large concepts in knowledge bases.

Given a concept description $D$, containing variables, and a concept description $C$, without variables, the matching problem asks for a substitution $\sigma$ (of
the variables by concept descriptions) such that $C \sqsubseteq \sigma(D)$. More detailed information about the concept matching problem can be found at [Baader et al., 1998].

### 3.4.6.6 Concept Rewriting

The idea of Rewriting is, given a concept expression, to find a concept which is related to the given concept according to equivalence, subsumption, or some other relation. This can be used to reformulate concepts during maintenance of a knowledge base or to translate concepts from one knowledge base to another.

According to [Baader et al., 2000], the problem of rewriting a concept can be described as follows: given a set of concept definitions (TBox) $\mathcal{T}$ and a concept description $C$ that does not contain concept names defined in $\mathcal{T}$, can this description be rewritten into a related description $E$ by using some of the names defined in $\mathcal{T}$?

### 3.4.6.7 Absorption

Absorption is a rewriting optimisation that tries to eliminate General Concept Inclusion axioms (GCIs) ([Tsarkov and Horrocks, 2006]).

### 3.4.6.8 Explanation

We use explanations all the time in our daily life to justify our opinions and decisions. Depending on the context in which they are used, those explanations can have many different forms.

While a DL reasoner can be used to derive inferences from, or detect contradictions in an ontology, most users have difficulties in understanding inferences and/or fixing errors in an ontology. This is because most reasoners only report inferences (or errors) in the ontology without explaining how or why they are derived.

So an approach to explanation in knowledge-based systems is the following: the user asks why a conclusion has been reached and gets presented the reasoning trace of the system [Sørmo and Cassens, 2004]. This helps the user in understanding how the system works, to be confident in the system’s output and to eventually debug an ontology.

### 3.5 DL Reasoner Overview

This section provides a survey on four state-of-the-art Description Logic reasoners, namely: FaCT++, KAON2, Pellet and Racer. We do not provide performance evaluations of the reasoners, these are usually available at the reasoner sources.

#### 3.5.1 FaCT++

FaCT++\(^1\) is a DL reasoner for the $\mathcal{SHIQ}(\mathcal{D})$ Description Logic, and thus for OWL-DL. It is a C++ re-implementation of the DL reasoner FaCT ([Fast Classification of Terminologies] Horrocks, 1998), which has been implemented in Lisp. FaCT++ system descriptions can be found in [Tsarkov and Horrocks, 2003], [Tsarkov and Horrocks, 2004] and [Tsarkov and Horrocks, 2006].

\(^1\)http://owl.man.ac.uk/factplusplus/
The reasoner implements tableaux algorithms, which were already established in FaCT. During the re-implementation new optimizations were introduced. One of these is a new “ToDo list” architecture, that, according to [Tsarkov and Horrocks, 2005] is better suited to more complex tableaux algorithms.

Unfortunately FaCT++ has currently only limited support for datatypes - only integers and strings are supported.

3.5.1.1 Features

- **Satisfiability/Subsumption** - The reasoner checks the satisfiability and subsumption of given concepts.

- **Consistency check** - FaCT++ can be used to check the consistency of an ontology.

- **Classification** - By classifying the whole ontology, a taxonomy can be created.

- **First-Order logic interface** - FaCT++ can create First-Order logic problems for subsumption and satisfiability checking, using the standard syntax TPTP[^2].

- **General axioms** - FaCT++ uses the absorption technique to efficiently deal with general axioms. This technique was introduced in [Horrocks, 1997].

3.5.1.2 Interfaces

The FaCT++ reasoner is available as standalone software component, which works only in batch mode, and provides a DIG interface and a servlet, that implements HTTP DIG reasoner functionality.

3.5.1.3 License

Fact++ is released under the GNU public license and thus freely available.

3.5.2 KAON2

KAON2[^3] is an infrastructure for managing OWL-DL, SWRL and F-Logic ontologies. It is the successor of KAON (Karlsruhe ontologie), which used a proprietary extension of RDFS, whereas KAON2 is based on OWL-DL. KAON2 provides a hybrid reasoner that allows datalog-style rules to interact with structural Description Logics knowledge bases. It supports the $SHIQ(D)$ description logic and disjunctive datalog programs.

So w.r.t. OWL, KAON2 is able to reason with the $SHIQ(D)$ subset of OWL-DL. This includes all features of OWL-DL apart from nominals. Since nominals are not a part of OWL Lite, KAON2 supports all of OWL Lite.

Unlike most currently available DL reasoners, the Java-based KAON2 reasoner does not implement the tableaux calculus. It implements algorithms which reduce a $SHIQ(D)$ knowledge base to a disjunctive datalog program. These algorithms allow applying well-known deductive database techniques, such as magic sets or join-order optimizations, to DL reasoning. More detailed information can be found at [http://kaon2.semanticweb.org/](http://kaon2.semanticweb.org/)

3.5.2.1 Features

- **Query answering** - KAON2 provides a conjunctive query answering machinery (without support for non-distinguished variables). Queries can be formulated using SPARQL\(^4\).

- **Built-in functions** - KAON2 supports basic built-in functions, such as adding and subtracting, that deal with integers and strings.

- **SWRL** - The so-called DL-safe subset of the Semantic Web Rule Language (SWRL) is supported.

3.5.2.2 Open Issues

- **Nominals** - KAON2 cannot handle nominals, so that each reasoning task over an ontology that contains a nominal concept will throw an error.

- **Cardinality statements** - Currently KAON2 cannot handle large numbers in cardinality statements.

3.5.2.3 Interfaces

The KAON2 reasoner provides a DIG interface and a KAON2 API for programmatic management of OWL-DL, SWRL and F-Logic ontologies. It also offers a standalone server providing access to ontologies using RMI.

3.5.2.4 License

KAON2 is owned by the company Ontoprise and is available free of charge for research and academic purposes. To use KAON2 in a commercial setting, please look at [www.ontoprise.de](http://www.ontoprise.de).

3.5.3 Pellet

Pellet\(^5\) is a sound and complete DL reasoner for $\mathcal{SHOIN(D)}$ Description Logics [Sirin and Parsia, 2006]. The Java-based reasoner provides an option to enable the Unique Name Assumption (UNA) and it includes reasoning about nominals. It is based on the tableaux algorithms developed for expressive Description Logics and incorporates the decision procedure for $\mathcal{SHOIQ}$ (the expressivity of OWL DL plus qualified cardinality restrictions) [Horrocks and Sattler, 2005]. More details about Pellet’s features, its architecture and implemented optimization techniques, as well as its future directions can be found in [Sirin et al., 2006].

3.5.3.1 Features

- **Ontology analysis and repair** - Pellet incorporates a number of heuristics to detect OWL Full documents that can be repaired as to get a valid OWL DL document.

- **Species Validation** - During the species validation it is checked - through different structural checks - whether the document is within OWL DL or OWL Full.

\(^4\)http://www.w3.org/TR/rdf-sparql-query/

D1.2 – Reasoning Techniques and Prototype Implementation for WSML

- **Classification** - By classifying the whole ontology, a taxonomy can be created.
- **Consistency check** - Check the consistency of ontologies.
- **Conjunctive ABox query** - The ABox query answering module in Pellet uses the "rolling-up" technique [Tessaris, 2001]. Queries can be formulated by both SPARQL\(^6\) and RDQL\(^7\).
- **Entailment** - Entailment support, using a technique for reducing entailment to Knowledge Base consistency [Horrocks and Patel-Schneider, 2004], is implemented in Pellet.
- **Datatype Reasoning** - Pellet can reason with XML Schema datatypes and also provides support for simple user-defined datatypes that are derived from XML Schema datatypes.
- **E-Connections** - Using E-Connections, Pellet enables multi-ontology reasoning. E-Connection, a knowledge representation language, is used as a language for defining and instantiating combinations of OWL DL ontologies. This approach provides an alternative to owl:imports.
- **SWRL** - Pellet supports the Semantic Web Rules Language (SWRL).
- **Ontology debugging** - An explanation support for understanding inferences and for fixing errors in an OWL ontology is provided by pinpointing axioms that cause an inconsistency or relations between unsatisfiable concepts.

### 3.5.3.2 Interfaces

The Pellet reasoner can be used with a simple command line interface, with the Jena and OWL-API libraries and it also provides a DIG interface.

### 3.5.3.3 License

Pellet is an open-source reasoner which is released under the MIT license and thus freely available.

### 3.5.4 Racer

RACER\(^8\) (Renamed ABox and Concept Expression Reasoner) is a DL reasoner for the SHIQ Description Logic. It implements tableau calculus for a very expressive Description Logic and offers reasoning for both TBoxes and ABoxes. It provides full reasoning support for OWL Lite and OWL DL ontologies, but only with approximations for nominals in OWL DL. Although RACER is not fixed to the Unique Name Assumption (UNA), the latter can be globally enabled. The reasoner also provides facilities for algebraic reasoning including concrete domains for dealing with:

- min/max restrictions over integers,
- linear polynomial (in-)equations over reals or cardinals with order relations,

\(^6\)http://www.w3.org/TR/rdf-sparql-query/
\(^7\)http://www.w3.org/Submission/RDQL/
\(^8\)http://www.racer-systems.com/
• equalities and inequalities of strings.

No support is yet provided for user-defined XML datatypes.

### 3.5.4.1 Features

- **Satisfiability/Subsumption** - The reasoner checks the satisfiability and subsumption of given concepts.
- **Consistency check** - RACER can be used to check the consistency of an ontology.
- **Classification** - By classifying the whole ontology, a taxonomy can be created.
- **Instance retrieval** - Retrieve instances from specified concepts or from certain query concepts. The retrieval from tuples of individuals, that satisfy certain conditions, is also possible.
- **Realization** - The reasoner finds the most specific concepts from a given individual.
- **Find synonyms** - RACER can find synonyms for resources (either classes or instance names).
- **OWL-QL querying** - an OWL-QL query processing system is available as an open-source project for RACER.
- **Query language** - RACER provides a new expressive ABox query language, nRQL (new Racer Query Language).
- **Incremental query answering** - RACER supports incremental query answering for information retrieval tasks (retrieve the n next results of a query). It also allows for a resource bounded or a resource-aware computation (maximum query answering times specified or “easy” answers are computed first and with minimum resources).
- **General axioms** - RACER supports the specification of general terminological axioms.
- **Concrete domains** - Check if certain concrete domains are entailed by an ABox and a TBox.
- **Filler of a role** - RACER supports the computation of the fillers of a role with reference to an individual w.r.t. an ABox and a TBox.
- **Retrieve imported resources** - An HTTP client can retrieve imported resources from the web. Multiple resources can be imported into one ontology.

### 3.5.4.2 Interfaces

The RACER reasoner provides a command line file interface, a DIG interface, a Java API, a Lisp API and a Web Service interface.

Both the Java API and the Lisp API are based on TCP sockets. These socket interfaces enables client application programs to access the services of the RACER server.
3.5.4.3 License

RACER is owned by the company Racer Systems GmbH & Co. KG and is provided free of charge for universities and research labs (RacerPro free educational and trial license agreement). The RACER source code is not publicly available. To use RACER in a commercial setting, please look at http://www.racer-systems.com/index.phtml.

3.6 Reasoner Interfaces

The four DL reasoners described above provide different interfaces to make their services accessible for users and applications. Two of these interfaces are the Description Logic Interface DIG and the OWL API. Their features are described and compared in the following sections.

3.6.1 DIG

The DIG Description Logic Interface\(^9\) specifies a common standardized XML interface for DL Reasoners. It has been developed by the DL Implementation Group\(^10\). All of the four reasoners described in the previous chapter provide support for DIG.

The DIG interface defines a simple protocol (based on HTTP PUT/GET) along with an XML Schema that describes a concept language and accompanying operations (offering an ask/tell functionality). DIG’s concept language is based on $\text{SHIQD}^-_n$.

The following sections briefly describe the HTTP interface and the accompanying operations of DIG. More details can be found in [Bechhofer et al., 2003a].

3.6.1.1 HTTP Interface

DIG’s XML Schema is to be used over an HTTP interface to a DL reasoner. The reasoner thus must provide basic HTTP support and be able to parse and generate XML content.

To interact with a DIG reasoner, a DIG client initiates one or more POST requests, with the body of the requests being messages encoded using the DIG XML Schema. The server responds with an HTTP response code and message. In the body of the response, an XML encoded message corresponds to a DIG response.

The connection to a reasoner is stateless: clients are not identified to the reasoner and the reasoner doesn’t maintain any kind of consistency checking or informations about which client is adding information or making requests.

Using persistent connections, it is possible to pipeline requests. A single TCP connection can be used for multiple requests without waiting for each response.

3.6.1.2 Message Formats

The DIG specification provides three classes of action to the clients: management, tell and ask operations.

\(^9\)http://dig.sourceforge.net/
\(^10\)http://dl.kr.org/dig/
Management operations allow the creation and management of knowledge bases by the DIG reasoner.

Tell operations are used to make assertions in the reasoner’s knowledge base (KB). This information can never be removed from the KB, unless by releasing the whole KB. The response to a tell statement will be a response message containing an “ok” or an error element.

Ask operations are used to query the KB. One request may represent multiple queries. The response to an ask statement will be a response message containing one response for each query in the ASK request.

### 3.6.1.3 Open Issues

The expressivity of DIG 1.1, the current release, is not sufficient to capture general OWL-DL ontologies, in particular datatype support is lacking: the specification only supports Integers and Strings. Another problem are the poor fit between DIG’s notion of relations and OWL’s properties.

A working group has been formed to address these issues and a revised DIG specification should be available in 2006.

### 3.6.2 OWL API

The WonderWeb OWL API\(^{11}\) provides a high-level Java-based programmatic interface for accessing, manipulating and reasoning over OWL ontologies. Along with the interface comes a simple concrete reference implementation.

Pellet (see Section 3.5.3) supports the OWL API directly by providing an OWL API interface.

The following sections briefly describe the OWL API functionalities and architecture. More details can be found in [Bechhofer et al., 2003c].

#### 3.6.2.1 Functionality Overview

The OWL API provides different aspects of support for OWL, which is reflected in the different packages the API contains:

- **Modeling** - The API provides a number of classes and interfaces to represent OWL ontologies. The model package provides read-only access to an OWL ontology, offering mainly methods for accessing existing OWL elements.

  A client application can use the API without being concerned about the particular implementation behind. An ontology can thus be stored simply in-memory or provide, e.g., a relational database or an RDF store.

- **Manipulation** - The API’s change package provides mechanisms for manipulation of OWL documents. It allows the addition, change and removal of entities, definitions, axioms, etc. It also provides support for composite changes (being decomposed to chains of changes), which allow to indicate a context for this change.

- **Validation** - The Species Validator checks OWL-RDF documents for basic syntactic errors and determines which particular sub-species of the language the document belongs to. An online validator is available at: [http://phoebus.cs.man.ac.uk:9999/OWL/Validator](http://phoebus.cs.man.ac.uk:9999/OWL/Validator)

\(^{11}\)http://owl.man.ac.uk/api.shtml
• **Parsing** - The parser builds an internal representation of a concrete representation from an OWL document. Parsing is supported for RDF-XML syntax and for OWL abstract syntax.

• **Serializing** - The Serializer writes an internal data structure to OWL concrete syntax (e.g. RDF-XML syntax or OWL abstract syntax).

• **Inference** - The inference package is used to encapsulate OWL DL reasoners and provides access to functionality related to reasoning with OWL ontologies. Although the method signatures are provided, there is no guarantee that the semantics are always correctly implemented.

### 3.6.3 DIG vs. OWL API

Both DIG and the OWL API allow a programmatic access to knowledge base systems. DIG is a protocol-centric approach assuming a client-server architecture for application development. The OWL API is component-based and thus offers a reusable component to OWL-based applications.

The OWL API offers a higher-level abstraction which helps to isolate application developers from underlying issues of syntax and presentation. It also allows to abstract from the underlying Description Logic, increasing the usage of DL knowledge bases in the Semantic Web area.

DIG, on the other hand, is a lower-level interface, which helps to isolate developers from languages as RDF, OWL and a specific programming language of a client application. Thus DIG sits below the OWL API and can provide a reasoning functionality that the OWL API expects. Actually the OWL API provides an inference interface for usage with DIG reasoners.

Both interfaces do not provide a complete support for reasoning with WSML-DL. DIG does not support all WSML datatypes (only Integers and Strings) and the OWL API cannot model Qualified Cardinality Restrictions, as these are not part of the OWL specification.

In the context of this work and of the corresponding implementation of a WSML-DL reasoner wrapper (see Section 3.7.3), we have decided to primarily use the OWL API. This decision derives mostly from the fact that the OWL API provides a higher-level interface and complete support for datatypes. After the translation from WSML-DL to OWL DL, we used the OWL API reasoner interface with Pellet. Through the DIG export functionality of the OWL API, we can, in a later stage, also use reasoners that do not support the OWL API, but DIG.

### 3.7 Reasoning With WSML

Offering reasoning support for a Semantic Web Services Language is crucial for the usage and acceptance of the language in the Semantic Web area. Reasoning allows, e.g. to check the consistency of an ontology and of ontology elements and to build classifications of the ontology objects.

Concerning reasoning with WSML (see Section 3.2), a main task is to design and implement reasoners for the different variants of WSML. Instead of implementing new reasoners, existing reasoner implementations can be used for WSML through a wrapper that translates WSML expressions into the appropriate syntax for the reasoner.
In the following sections we briefly describe WSMO4J and the WSML2Reasoner framework. We talk in more detail about the newly added wrapper for the WSML-DL reasoner.

3.7.1 WSMO4J

WSMO4J\footnote{http://wsmo4j.sourceforge.net/} is a Java-based API (WSMO API) and Reference Implementation for the Web Service Modeling Ontology (WSMO) as defined in \cite{Roman et al., 2004}. It is a low level infrastructure layer that can be used by WSMO applications.

3.7.1.1 Functionality Overview

The WSMO API provides the following core functionalities:

- **Modeling** - WSMO4J supports modeling of the conceptual syntax and of the logical expression syntax. Modeling allows adding, changing and removing WSMO elements from an ontology.

- **Validation** - The WSML Validator checks WSML documents for syntactic errors and determines which particular variant of the language the document belongs to. An online validator is available at: \url{http://tools.deri.org/wsml/validator/v1.2/}.

- **Parsing** - Building an internal representation of a WSML document.

- **Serializing** - Serializing an internal data structure to the different WSML variants syntax.

- **Import/Export** - WSMO4J supports import and export from/to other formats (e.g. XML, RDF).

3.7.2 WSML2Reasoner

The WSML2Reasoner framework\footnote{http://dev1.deri.at/wsml2reasoner/} is a flexible architecture for easy integration of external reasoning components.

So far the framework has two external reasoners integrated: KAON2 (WSML-Flight reasoner) and MINS (WSML-Rule reasoner).

3.7.3 Adding WSML-DL Wrapper to WSML2Reasoner

For the WSML-DL reasoner we have implemented a translation from WSML-DL to OWL DL (see Section 3.3.2). This is the appropriate syntax for the OWL DL reasoner Pellet (see Section 3.5.3), which completely supports OWL DL. Pellet, wrapped by a Pellet reasoner facade, has been embedded as the first DL reasoner in the framework infrastructure. The facade mediates between the OWL DL ontology produced by the transformation and the Pellet-specific internal representation.

With regard to WSML-DL, the reasoner architecture consists of five different components:
• **Web Interface** - A web interface for the WSML-DL reasoner will be available in the near future.

• **DL Reasoner** - Pellet provides the reasoning services. In the future more reasoners will be added to the framework, e.g. FaCT++, KAON2, RACER.

• **Reasoner Interface** - The reasoner interface implements the translation from WSML-DL to OWL DL. In a later stage there will be added an export from OWL DL to the Description Logic Interface DIG.

• **OWL API** - The OWL API is used by the reasoner interface for transforming the WSML-DL ontology to an OWL DL ontology.

• **WSMO4J** - WSMO4J is used by the reasoner interface for parsing and internal representation of WSML documents.

For each DL reasoner that is embedded into the WSML2Reasoner framework, an adapter facade has to be written.

### 3.7.3.1 Dependencies

The WSML-DL reasoner requires the following libraries:

• Pellet 1.3 (MINDSWAP)
• OWL API (Wonderweb)
• econn OWL API (MINDSWAP)
• xsdlib (Sun Microsystems)
• relaxngDatatype (Sun Microsystems)
• commons-logging 1.1 (Apache)
• log4j (Apache)
• rdfapi (KAON)
• aterm (Centrum voor Wiskunde en Informatica (CWI))

All libraries except for the original OWL API and the rdfapi are included in the Pellet distribution. They are available at [http://www.mindswap.org/2003/pellet/download/pellet-1.3.zip](http://www.mindswap.org/2003/pellet/download/pellet-1.3.zip). Please note that Pellet uses an extended version of the original OWL API; when you download Pellet, it comes with this extended version (you need the “api.jar” and the “impl.jar” from this extended version) and should work fine. The OWL API is available from [http://sourceforge.net/projects/owlapi](http://sourceforge.net/projects/owlapi) and the rdfapi from [http://sourceforge.net/projects/kaon](http://sourceforge.net/projects/kaon).

### 3.7.3.2 Ontology Normalization

Before translating WSML-DL to OWL DL, we apply some preprocessing steps to the WSML-DL ontology to build simpler and less expressions (see Section 3.3.1).

We use the following normalizers and logical expression rules that were already implemented in an earlier stage of the WSML2Reasoner framework:

• **AnonymousId Translator** - Translates anonymous identifiers to globally unique identifiers by creating new identifiers with the system’s time in milliseconds.
• **Axiomatization Normalizer** - Transforms conceptual syntax to logical expressions.

• **Implication Reduction Rules** - Transform equivalences and right implications to left implications.

• **Molecule Decomposition Rules** - Replace complex molecules by conjunctions of simple ones.

Additionally we have implemented some new normalizers and logical expression rules:

• **Relation2Attribute Normalizer** - Replace relations, subrelations and relation instances according to the preprocessing steps described in Section 3.3.1. This normalizer needs to be applied before the Axiomatization Normalizer. The latter transforms relations into logical expressions containing Negation-as-failure `naf`, which is not a legal WSML-DL construct. Furthermore it does not support the transformation of relations with impact on more than one type (e.g. `ageOfHuman(impliesType {Man, Woman}, ofType integer)`).

• **InverseImplication Rules** - Replace conjunctions on the left side and disjunctions on the right side of an inverse implication by left implications.

### 3.7.3.3 DL Reasoner Functionality

The reasoner offers the following functionality:

• Set the ontology that the reasoner will reason with.

• Print a class hierarchy with individuals of the ontology.

• Get information about the registered ontology. E.g. the expressivity, the number of concepts, attributes and instances.

• Check Ontology|Concept|Logical Expression consistency.

• Get all concepts|instances|attributes from the ontology.

• Get all constraint|infering attributes from the ontology.

• Get all subconcepts|superconcepts of a specified concept.

• Get all concepts|attributes equivalent to a specified concept|attribute.

• Check if two concepts are equivalent.

• Check if a specified concept is a subconcept of another specified concept.

• Check if a specified instance is a member of a specified concept.

• Get all instances of a specified concept.

• Get all concepts a specified instance is member of.

• Get all sub|superrelations(-attributes) of a specified relation(attribute).

• Get all relations(attributes) inverse to a specified relation(attribute).

• Get all concepts from a specified attribute.

• Get all instance|datavalue ranges from a specified infering|constraint attribute.
• Get all infering\textit{constraint attributes with the corresponding values|datavalues from a specified instance.}

• Get all instances that have values for a specified constraint\textit{infering attribute, with the corresponding values.}

• Check if a specified instance has a specified constraint\textit{infering attribute value.}

• Get values\textit{datavalues for a specified instance and a specified constraint\textit{infering attribute.}

\subsection*{3.7.3.4 Limitations}

The limitations of the current version of the WSML-DL reasoner and of the transformation from WSML-DL to OWL DL are:

• \textbf{Web access} - The WSML-DL reasoner web interface will be available in the near future.

• \textbf{Query answering} - There is currently no support for query answering.

This feature will be added in a future step. The user should be able to send WSML logical expression queries, which then need to be translated to RDQL or SPARQL queries.

• \textbf{QCRs} - The transformation currently does not support Qualified Cardinality Restrictions (QCRs) (see Section 3.1.2.3 and 3.3.2). In a next step we will implement the workaround for QCRs in OWL DL, proposed in Section 3.1.2.3. We intend to extend the OWL API for support of the non-endorsed OWL extension (see Section 3.1.2.3) later.

• \textbf{Annotations} - The mapping from WSML \textit{importsOntology, usesMediator and Non Functional Properties on ontology level is not yet implemented in the transformation.}

• \textbf{Ontology Import} - Ontology imports are currently ignored. If a WSML file imports other ontologies, these should also be transformed and imported to the resulting OWL ontology.
4 Conclusions & Future Work

We have presented our work on a generic framework for WSML ontology reasoning. The framework is guided by the following design principles:

- The WSML Reasoning Framework is based on a common knowledge representation format that is independent of WSML and in which all WSML variants can be embedded. This shared format is based on generalized clauses (or rules) which can include default negation.

- Transformations allow to map ontologies that belong to either of the WSML variants into the shared format.

- Adapter allow to integrate arbitrary external components (which are not specifically developed for WSML or which are not WSML-aware) to realize particular reasoning tasks over WSML ontologies. The only requirement for the integration of an external tool is that it implements a relevant reasoning task (such as e.g. satisfiability checking) for a subset (or all) of the shared knowledge representation format.

- Applications that need some reasoning support (such as a Web Service discovery component) are not aware of the transformation or particular underlying reasoners. They are shielded from these technical details by means of a generic and extensible WSML-Reasoning API. Applications do not have to be adapted when changing either the transformation or the underlying reasoning system. Changes (in particular of the latter) can even be done at runtime in a transparent way.

Therefore, it provides enough flexibility allow for smooth and effortless integrate existing systems that can deal with specific subsets the common knowledge representation format of generalized clauses.

We have provided a prototype implementation for ontology reasoning that can deal with the WSML-Core and the WSML-DL language. For both languages, we have integrated various reasoning systems. Furthermore, for rule-based WSML we provide an own implementation of a datalog engine. This engine is under constant development and will provide the basis a reasoner that can deal with the shared general clause format later on. This way, we expect to be able to deal with the major reasoning tasks that are considered in this project (i.e. logical entailment checking, satisfiability checking and instance retrieval) in an efficient and integrated way. In regard of the latter, we expect the Hyper-Tableaux calculus [Baumgartner et al., 1996][Baumgartner, 1997] (with suitable adaptations and refinements) to provide an adequate algorithmic basis for a system that can deal efficiently with generalized clauses and smoothly integrates with specific techniques for reasoning with important subsets of this shared representation format at the same time. Furthermore, we are optimistic that based on Hyper-Tableau methods we can implement a system which posses an extremely desirable property, namely the principle of graceful degradation, i.e. in ontology reasoning computational costs only go along with the concrete language features that are actually used in an ontology, not with an underlying implementation that aims at covering all the expressive language features that might occur in an ontology of the most expressive language variant. Hence, within one system that covers all the variants, computational costs are bound to ontologies and not to the ontology language for which the system has been developed.
4.1 Software & Online Demo

To use our prototype for reasoning with WSML, one has to obtain the following software libraries:

- **WSML2Reasoner Framework** implements the transformation from WSML to the common shared knowledge representation format. Download the most recent release from [http://dev1.deri.at/wsml2reasoner/](http://dev1.deri.at/wsml2reasoner/). The library includes the WSML Reasoning API. Instructions on how to use the API in applications as well as examples can be found at this website.

- **WSMO4j** implements an in-memory representation of WSML ontologies for Java. The most recent release can be downloaded at [http://wsmo4j.sourceforge.net/snapshot](http://wsmo4j.sourceforge.net/snapshot).

- **MINS** implements a datalog engine in Java for datalog with support for stratified default negation, negation under well-founded semantics and function symbols. The latest release can be downloaded at [http://dev1.deri.at/mins/](http://dev1.deri.at/mins/).

- **Pellet** is a Description Logic reasoning system that supports all features of OWL-DL. It can be downloaded at [http://www.mindswap.org/2003/pellet](http://www.mindswap.org/2003/pellet).

![WSML Reasoner Online Demo](image)

Figure 4.1: The Web Interface of the WSML Reasoner Online Demo

Furthermore, we provide an online demo (see Figure 4.1 and Figure ?? for the instance retrieval reasoning task) of our system at [http://tools.deri.org/wsml/rule-reasoner/](http://tools.deri.org/wsml/rule-reasoner/) as well as a Web Service interface for demonstration purposes at [http://tools.deri.org/services/reasoner.wsdl](http://tools.deri.org/services/reasoner.wsdl).

Acknowledgements

This project was funded by the Austrian Federal Ministry for Transport, Innovation, and Technology under the project **RW²** (FIT-IT contract FFG 809250).

We would like to thank to all the members of the WSML working group for their advice and input into this document.
Query Result Page

Your Query

עליים[ין.Float and hasValue ﹧]?have

Query answer:

 Husband Wife
 umer Marge

Figure 4.2: WSML Reasoner Online Demo: Result

Bibliography


