# A Reasoning Framework for Rule-Based WSML

Stephan Grimm[1], Uwe Keller[2], Holger Lausen[2], and Gábor Nagypál[1]

[1] FZI Research Center for Information Technologies at the University of Karlsruhe, Germany
{grimm,nagypal}@fzi.de
[2] Digital Enterprise Research Institute (DERI), University of Innsbruck, Austria
{uwe.keller,holger.lausen}@deri.org

**Abstract.** 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 realized 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.

## 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) [6], which comes in variants that follow the rule-based knowledge representation paradigm of logic programming [14]. 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 [8]. 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.

The WSML reasoning framework is jointly developed within, and funded by the European project DIP (IST-FP6-507483) and the Austrian projects SEnSE (FFG 810807) and **R**W$^2$ (FFG 809250) .

## 2 The WSML Language

The Web Service Modeling Language (WSML) [6] 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 (WSMO, `http://www.wsmo.org`) [15] and is based on well-known logic-based knowledge representation (KR) formalisms, namely Description Logics [2] and Logic Programming [14]. In fact, WSML is a family of representation languages that comes in several variants with different expressiveness. Besides various SWS-specific language constructs, such as “goal”, “interface”, “choreography” or “capability”, WSML particularly provides means to formulate the domain ontologies in terms of which SWSs are semantically annotated. Since here we are interested in reasoning with such semantic annotation with respect to the underlying ontology formalism, we focus on the ontology-related part of WSML. Furthermore, we use the human-readable syntax of the language in our presentation, while WSML also specifies XML and RDF serialisations to be compatible with existing web standards.

### 2.1 Language Constructs

WSML makes a clear distinction between the modeling of different conceptual elements on the one hand and the specification of complex axiomatic information on the other. To this end, the WSML syntax is split into two parts: the *conceptual syntax*, and *logical expression syntax*, while elements from both can be combined in a WSML document. We illustrate the interplay of conceptual modelling with logical expressions in WSML by means of an example given in Listing 1.1, which specifies an ontology in the domain of telecommunications taken from a project case study. For a complete account of all WSML syntax elements, we refer to [8].

**Listing 1.1.** WSML Example Ontology

```
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 DialupConnection subConceptOf NetworkConnection
concept DSLConnection subConceptOf NetworkConnection
axiom DialupConnection_DSLConnection_Disjoint definedBy
 !− ?x memberOf DialupConnection and ?x memberOf DSLConnection.
concept OnlineService subConceptOf BundlePart
concept SharePriceFeed subConceptOf OnlineService
axiom SharePriceFeed_requires_bandwidth
definedBy
 !− ?b memberOf ITBundle and ?b[hasOnlineService hasValue ?o]
    and ?o memberOf SharePriceFeed and
    ?b[hasNetwork hasValue ?n] and
    ?n[providesBandwidth hasValue ?x] and ?x < 512.
concept BroadbandBundle subConceptOf ITBundle
    hasNetwork ofType (1 1) DSLConnection
axiom BroadbandBundle_sufficient_condition
definedBy
    ?b memberOf BroadbandBundle :− ?b memberOf ITBundle
    and ?b[hasNetwork hasValue ?n] and ?n memberOf DSLConnection.
instance BritishTelekom memberOf TelecomProvider.
instance UbiqBankShareInfo memberOf SharePriceFeed.
instance MyBundle memberOf ITBundle
    hasNetwork hasValue ArcorDSL
    hasOnlineService hasValue UbiqBankShareInfo
    hasProvider BritishTelekom.
instance MSNDialup memberOf DialupConnection
    providesBandwidth hasValue 10.
instance ArcorDSL memberOf DSLConnection
    providesBandwidth hasValue 1024.
```

**Conceptual Modelling.** The WSML conceptual syntax for ontologies essentially allows for the modeling of concepts, instances and relations.

In ontologies, concepts form the basic elements for describing the terminology of the domain of discourse by means of classes of objects. In the telecommunications domain, a concept like NetworkConnection stands for the class of all network connections. Concepts can be put in a subsumption hierarchy by means of the **subConceptOf**-construct. For example, NetworkConnection is a subconcept of BundlePart, meaning that any network connection is part of some IT product bundle, and has itself the subconcepts DialupConnection and DSLConnection, as can be seen from Listing 1.1.

Attributes, i.e. binary relations, are used to relate concepts in a customary way, while they can point to other concepts or datatypes. In our example, NetworkConnection has a datatype attribute providesBandWidth, whereas concept ITBundle has attributes like hasNetwork or hasOnlineService that point to concepts for the single parts which make up the bundle. Attribute definitions can either be *constraining* (using **ofType**) or *inferring* (using **impliesType**)[3]. Constraining attribute definitions define a type constraint on the values for an 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 [3] and OWL [9]. Furthermore, an

[3] The distinction between inferring and constraining attribute definitions is explained in more detail in [7, Section 2].

attribute can be marked as **transitive**, **symmetric**, or **reflexive**, and can be constrained by a minimum and a maximum cardinality (using $(n_{min}\ n_{max})$), as can be seen from Litsing 1.1. Similar constructs are available to define n-ary relations in ontologies.

Instances represent concrete object a the domain, such as MSNDialup as a particular dial-up connection in the telecommunications domain. By means of the **memberOf**-construct, instances are associated with concepts, and using **hasValue** they are are linked to other instances or data values, as can also be seen in Listing 1.1. Notice, that WSML supports metamodelling and allows an entity to be both a concept and an instance.

**Logical Expressions.** By means of the **axiom**-construct, arbitrarily complex logical expressions can be included in a WSML ontology, interfering with the conceptual definitions. In our example, the axiom named BroadbandBundle_sufficient_condition specifies that any IT bundle that has a DSL network connection is concluded to be a broadband bundle.

The general logical expression syntax for WSML has a first-order logic style, in the sense that it has constants, function symbols, variables, predicates and the usual logical connectives. Additionally, WSML provides extensions based on F-Logic [12] as well as logic programming rules and database-style integrity constraints.

Besides standard first-order atoms, WSML provides so-called *molecules*, inspired by F-Logic, that can be used to capture information about concepts, instances, attributes and attribute values. A molecule of the form $I$ **memberOf** $C$ denotes the membership of an instance $I$ in a concept $C$, while a molecule $C_1$ **subConceptOf** $C_2$ denotes the subconcept relationship between concepts $C_1$ and $C_2$. Further molecules have the form $I[A$ **hasValue** $V]$ to denote attribute values of objects, $C[A$ **ofType** $T]$ to denote a type-constraining attribute signature, or $C[A$ **impliesType** $T]$ to denote an inferring attribute signature. Some of these molecule forms appear in Listing 1.1, e.g. in axiom BroadbandBundle_sufficient_condition.

WSML has the usual first-order connectives: the unary (classical) negation operator **neg**, and the binary operators for conjunction **and**, disjunction **or**, right implication **implies**, left implication **impliedBy**, and bi-implication **equivalent**. Variables, preceeded by the ?-symbol may be universally quantified using **forall** or existentially quantified using **exists**. Apart from first-Order constructs, WSML supports logic programming rules of the form $H :- B$ with the typical restrictions on the head and body expressions $H$ and $B$ (see [8]), allowing the symbol **naf** for negation-as-failure on atoms in $B$. A constraint is a special kind of rule with an empty head expression. While the aforementioned axiom is expressed by a rule, the axiom named DialupConnection_DSLConnection_Disjoint comes in form of a constraint, stating that no instance is allowed to be member of both the concepts DialupConnection and DSLConnection at the same time.

**Language Variants.** WSML comes in different variants that map to semantically different target formalisms. Therefore, each variant also defines some restrictions on the use of syntactical constructs: ***WSML-Core*** allows only first-order formulae which conform to DLP [11] as the least common denominator of the description logics and logic programming paradigms, by which its semantics is defined. It allows for most of conceptual modelling but is rather restricted in the use of logical expressions. ***WSML-DL*** allows first-order formulae which can be translated to the description logic $\mathcal{SHIQ}(\mathbf{D})$, that defines its semantics. Thus, WSML-DL is very similar to OWL [9]. ***WSML-Flight*** extends 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. WSML-Flight is based on the well-founded semantics [10] and additionally allows meta-modeling. ***WSML-Rule*** extends WSML-Flight by function symbols and unsafe rules, i.e. variables occurring in the head or in a negative body literal but not in a positive body literal. ***WSML-Full*** does not restrict the use of syntax and allows the full expressiveness of all other WSML variants under a first-order umbrella with nonmonotonic extensions.

In the following, we refer to the WSML-Core, WSML-Flight and WSML-Rule variants jointly as *rule-based WSML* and focus on reasoning in these variants.

### 2.2 Reasoning in Rule-Based WSML

Various reasoning tasks, such as consistency checking or entailment of implicit knowledge, are considered useful in Semantic Web and SWS applications. Here, we sketch the typical reasoning tasks for rule-based formalisms, and thus for rule-based WSML.

Let $O$ denote a WSML ontology and $\pi_{c-free}(O)$ denote the constraint-free projection of $O$, i.e. the ontology which is obtained from $O$ by removing all constraining description elements, such as attribute type constraints, cardinality constraints, integrity constraints etc. (1) **Consistency checking** means to verify whether $O$ is satisfiable, i.e. if $\pi_{c-free}(O)$ has a model in which no constraint in $O$ is violated. (2) **Ground Entailment** means, given some variable-free formula $\phi_g$, to check if $\phi_g$ is satisfied in all models of $\pi_{c-free}(O)$ in which no constraint in $O$ is violated. We denote this by $O \models \phi_g$. (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 $O \models Q(\vec{t})$.

## 3 Mapping WSML to Datalog

The semantics of rule-based WSML is defined via a mapping to Datalog [5, 1] with (in)equality, default negation and integrity constraints, as described in [8]. In the following, we refer to this language simply as Datalog. 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. WSML reasoning tasks are then realized by means of Datalog querying via calls to an underlying Datalog inference engine fed with the rules contained in this program.

### 3.1 Ontology Transformations

The transformation of a WSML ontology to Datalog rules forms a pipeline of single transformation steps that are subsequently applied, starting from the original ontology.

*Axiomatization.* In a first step, the transformation $\tau_{axioms}$ is applied as a mapping $\mathcal{O} \rightarrow 2^{\mathcal{LE}}$ 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

| *Expression $\alpha$ in conceptual syntax* | *Resulting logical expression(s)*: $\tau_{axioms}(\alpha)$ |
|---|---|
| **concept** $C_1$ **subConceptOf** $C_2$ | $C_1$ **subConceptOf** $C_2$. |
| **concept** $C$ $A$ **ofType** $(0,1)$ $T$ | $C$[A **ofType** $T$]. !- ?x **memberOf** $C$ **and** ?x[A **hasValue** ?y, A **hasValue** ?z] **and** ?y != ?z. |
| **concept** $C$ A$_1$ **inverseOf** A$_2$ **impliesType** $T$ | $C$[A **impliesType** $T$]. ?x **memberOf** $C$ **and** ?v **memberOf** $T$ **implies** ( ?x[A$_1$ **hasValue** ?v] **equivalent** ?v[A$_2$ **hasValue** ?x] ). |
| **relation** $R_1/n$ **subRelationOf** $R_2$ | $R_1(\vec{x})$ **implies** $R_2(\vec{x})$. where $\vec{x}$ = (x$_1$,...,x$_n$) |
| **instance** $I$ **memberOf** $C$ A **hasValue** $V$ | $I$ **memberOf** $C$. I[A **hasValue** $V$]. |

**Table 1.** Examples for axiomatizing conceptual ontology modeling elements.

| *original expression* | *normalized expression* |
|---|---|
| $\tau_n(\{E_1,\ldots,E_n\})$ | $\{\tau_n(E_1),\ldots,\tau_n(E_n)\}$ |
| $\tau_n(E_x$ **and** $E_y.)$ | $\tau_n(E_x)$ **and** $\tau_n(E_y)$ |
| $\tau_n(E_x$ **or** $E_y.)$ | $\tau_n(E_x)$ **or** $\tau_n(E_y)$ |
| $\tau_n(E_x$ **and** $(E_y$ **or** $E_z).)$ | $\tau_n(\tau_n(E_x)$ **and** $\tau_n(E_y)$ **or** $\tau_n(E_x)$ **and** $\tau_n(E_z).)$ |
| $\tau_n((E_x$ **or** $E_y)$ **and** $E_z).)$ | $\tau_n(\tau_n(E_x)$ **and** $\tau_n(E_z)$ **or** $\tau_n(E_y)$ **and** $\tau_n(E_z).)$ |
| $\tau_n($ **naf** $(E_x$ **and** $E_y).)$ | **naf** $\tau_n(E_x)$ **or** **naf** $\tau_n(E_y)$. |
| $\tau_n($ **naf** $(E_x$ **or** $E_y).)$ | **naf** $\tau_n(E_x)$ **and** **naf** $\tau_n(E_y)$. |
| $\tau_n($ **naf** $($ **naf** $E_x).)$ | $\tau_n(E_x)$ |
| $\tau_n(E_x$ **implies** $E_y.)$ | $\tau_n(E_y) :- \tau_n(E_x)$. |
| $\tau_n(E_x$ **impliedBy** $E_y.)$ | $\tau_n(E_x) :- \tau_n(E_y)$. |
| $\tau_n(X[Y_1,\ldots,Y_n].)$ | $X[Y_1]$ **and** ... **and** $X[Y_n]$. |

| *original expression* | *simplified rule(s)* |
|---|---|
| $\tau_{dlog}(\{E_1,\ldots,E_n\})$ | $\{\tau_{dlog}(E_1),\ldots,\tau_{dlog}(E_n)\}$ |
| $\tau_{dlog}($ !- $B.)$ | $\square :- \tau_{dlog}(B)$ |
| $\tau_{dlog}(H.)$ | $\tau_{dlog}(H)$ . |
| $\tau_{dlog}(H :- B.)$ | $\tau_{dlog}(H) :- \tau_{dlog}(B)$ |
| $\tau_{dlog}(E_x$ **and** $E_y.)$ | $\tau_{dlog}(E_x) \wedge \tau_{dlog}(E_y)$ |
| $\tau_{dlog}($**naf** $E.)$ | $\sim \tau_{dlog}(E)$ |
| $\tau_{dlog}(C_x$ **subConceptOf** $C_y.)$ | $p_{\mathsf{sco}}(C_x, C_y)$ |
| $\tau_{dlog}(I$ **memberOf** $C.)$ | $p_{\mathsf{mo}}(I, C)$ |
| $\tau_{dlog}(I[a$ **hasValue** $V].)$ | $p_{\mathsf{hval}}(I, a, V)$ |
| $\tau_{dlog}(C[a$ **impliesType** $T].)$ | $p_{\mathsf{itype}}(C, a, T)$ |
| $\tau_{dlog}(C[a$ **ofType** $T].)$ | $p_{\mathsf{otype}}(C, a, T)$ |
| $\tau_{dlog}($r$(X_1,\ldots,X_n).)$ | $r(X_1,\ldots,X_n)$ |
| $\tau_{dlog}(X$ = $Y.)$ | $X = Y$ |
| $\tau_{dlog}(X$ != $Y.)$ | $X \neq Y$ |

| *original expr.* | *simplified rule(s)* | *original expression* | *simplified rule(s)* |
|---|---|---|---|
| $\tau_{lt}(\{E_1,\ldots,E_n\})$ | $\{\tau_{lt}(E_1),\ldots,\tau_{lt}(E_n)\}$ | $\tau_{lt}(H_1$ **and** ... **and** $H_n :- B.)$ | $\tau_{lt}(H_1 :- B.), \ldots, \tau_{lt}(H_n :- B.)$ |
| $\tau_{lt}(H_1 :- H_2 :- B.)$ | $\tau_{lt}(H_1 :- H_2$ **and** $B.)$ | $\tau_{lt}(H :- B_1$ **or** , ..., **or** $B_n.)$ | $\tau_{lt}(H :- B_1.), \ldots, \tau_{lt}(H :- B_n.)$ |

**Table 2.** Normalization of WSML logical expressions.

constraints, are converted into appropriate axioms specified by logical expressions. Table 1 shows the details of some of the conversions performed by $\tau_{axioms}$, based on [8]. The WSML conceptual syntax constructs on the left-hand side are converted to the respective WSML logical expressions on the right-hand side. 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 data values.

*Normalization.* The transformation $\tau_n$ is applied as a mapping $2^{\mathcal{LE}} \to 2^{\mathcal{LE}}$ to normalize WSML logical expressions. This normalization step reduces the complexity of formulae according to [8, Section 8.2], to bring 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. The left part of Table 2 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 $\tau_n$ has been applied, the resulting 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 $2^{\mathcal{LE}} \rightarrow 2^{\mathcal{LE}}$ to flatten the complex WSML logical expressions, producing simple rules according to the Lloyd-Topor transformations [13], as shown in the lower part of Table 2. 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 [8]. After this step, the resulting WSML expressions have the form of proper Datalog rules with a single head and conjunctive (possibly negated) body literals.

*Datalog Rule Generation.* In a final step, the transformation $\tau_{dlog}$ is applied as a mapping $2^{\mathcal{LE}} \rightarrow \mathcal{P}$ 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 **subConceptOf** or **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 3.2. The right-hand part of Table 2 shows the mapping from WSML logical expressions to Datalog including the meta-level predicates $p_{\mathsf{sco}}$, $p_{\mathsf{mo}}$, $p_{\mathsf{hval}}$, $p_{\mathsf{itype}}$ and $p_{\mathsf{otype}}$ 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.

The resulting Datalog rules are of the form $H \ :- B_1 \wedge \ldots \wedge B_n$, 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 $\Box$.

Ultimately, we define the basic[4] transformation $\tau$ for converting a rule-based WSML ontology into a Datalog program based on the single transformation steps introduced before by $\tau = \tau_{dlog} \circ \tau_{lt} \circ \tau_n \circ \tau_{axioms}$. As a mapping $\tau : \mathcal{O} \rightarrow \mathcal{P}$, this composition of the single steps is applied to a WSML ontology $O \in \mathcal{O}$ to yield a semantically equivalent Datalog program $\tau(O) = P \in \mathcal{P}$ when interpreted with respect to the meta-level axioms discussed next.

## 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 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 **subConceptOf, memberOf** or **hasValue**, are mapped to Datalog predicates that form a meta-level vocabulary for the WSML language constructs. These are the meta-level

[4] Later on, the transformation pipeline is further extended to support datatypes and debugging.

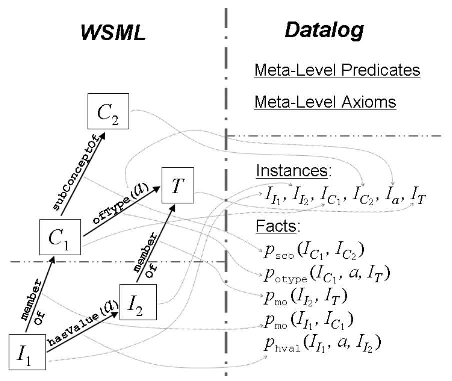
WSML
Datalog
Meta-Level Predicates
Meta-Level Axioms
$C_2$
$C_1$
$T$
$I_2$
$I_1$
subConceptOf
ofType (d)
memberOf
memberOf
hasValue (d)
Instances:
$I_{I_1}, I_{I_2}, I_{C_1}, I_{C_2}, I_a, I_T$
Facts:
$p_{\mathtt{sco}}(I_{C_1}, I_{C_2})$
$p_{\mathtt{otype}}(I_{C_1}, a, I_T)$
$p_{\mathtt{mo}}(I_{I_2}, I_T)$
$p_{\mathtt{mo}}(I_{I_1}, I_{C_1})$
$p_{\mathtt{hval}}(I_{I_1}, a, I_{I_2})$

**Fig. 1.** Usage of meta-level predicates.

| *Meta-Level Axioms* |
|---|
| (1) $p_{\mathsf{sco}}(C_1, C_3)\ :-\ p_{\mathsf{sco}}(C_1, C_2) \wedge p_{\mathsf{sco}}(C_2, C_3)$ |
| (2) $p_{\mathsf{mo}}(I, C_2)\ :-\ p_{\mathsf{mo}}(I, C_1) \wedge p_{\mathsf{sco}}(C_1, C_2)$ |
| (3) $p_{\mathsf{mo}}(V, C_2)\ :-\ p_{\mathsf{itype}}(C_1, a, C_2) \wedge p_{\mathsf{mo}}(I, C_1) \wedge p_{\mathsf{hval}}(I, a, V)$ |
| (4) $\Box\ :-\ p_{\mathsf{otype}}(C_1, a, C_2) \wedge p_{\mathsf{mo}}(I, C_1) \wedge p_{\mathsf{hval}}(I, a, V) \wedge \sim p_{\mathsf{mo}}(V, C_2)$ |

**Fig. 2.** WSML semantics in Datalog.

predicates that appear in Table 2 for $\tau_{dlog}$, and which are applied to the Datalog constants that represent the WSML entities. The facts listed in Figure 1 illustrate the use of the meta-level predicates. For example, the predicate $p_{\mathsf{mo}}$ 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 $p_{\mathsf{mo}}$.

A fixed set $P_{meta}$ of Datalog rules, shown in Figure 2, forms the meta-level axioms which assure that the original WSML semantics is properly maintained. Axiom (1) realizes transitivity for the WSML **subConceptOf** construct, while axiom (2) ensures that an instance of a subconcept 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.

## 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 $P_O = P_{meta} \cup \tau(O)$ 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 $Q(\vec{x})$ to a Datalog program $P \in \mathcal{P}$ is denoted by $(P,\ ?-\ Q(\vec{x}))$ and yields a set of tuples that instantiate the vector $\vec{x}$ of variables in the query. If $Q(\vec{x})$ contains no variables, in fact a boolean query $Q$ is posed that instead evaluates either to $\{Q\}$ if $Q$ is satisfied in the well-founded model of $P$ or $\emptyset$ otherwise.

*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: $O$ is satisfiable $\Leftrightarrow$ $(P_O,\ ?-\ \Box) = \emptyset$ . 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 ground entailment by a WSML ontology is done by using queries that contain no variables, as expressed in the following equivalence: $O \models \phi_g \Leftrightarrow (P_O,\ ?-\ \tau'(\phi_g))) \neq \emptyset$. The WSML ground fact $\phi_g \in \mathcal{LE}$ is transformed to Datalog with a transformation $\tau' = \tau_{dlog} \circ \tau_{lt} \circ \tau_n$, 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 $\phi_g$ is entailed by the original ontology, otherwise it is not.

*Retrieval* – Similarly, instance retrieval can be performed by posing a WSML query $Q(\vec{x})$ with free variables $\vec{x}$ to the Datalog program $P_O$, which yields the following set: $\{\vec{o}\,|\,O \models Q(\vec{o})\} = (P_O,\ ?-\ \tau'(Q(\vec{x})))$. The query $Q(\vec{x})$ is transformed to Datalog by $\tau'$ and evaluated together with the program $P_O$. The resulting set contains all object tuples $\vec{o}$ for which an instantiation of the query expression is entailed by the original ontology, while the objects in $\vec{o}$ can be identifiable WSML entities or data values. To give an example, the query $Q($?x$)$ = ?x **memberOf** BroadbandBundle posed to the ontology in Listing 1.1 yields the set $\{(\textit{MyBundle})\}$ that contains one unary tuple with the instance *MyBundle*, which can be inferred to be a broadband bundle due to its high network bandwidth.

### 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 is related to Axiom (4) in Figure 2, which checks attribute type constraints. The crucial part of the axiom is the literal

$$\sim p_{\mathsf{mo}}(V, C_2)$$

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 – there is no fact $p_{\mathsf{mo}}(10, \_\mathsf{integer})$ for the value of the providesBandwidth attribute. Whenever a value is defined for an attribute constrained by **ofType**, Axiom (4) would cause a constraint violation.

To solve this problem, $p_{\mathsf{mo}}$ facts should be generated for all datatype constants that appear as values of attributes having **ofType** constraints in the ontology. I.e., for each such constant in the ontology, axioms of the following form should appear:

$$p_{\mathsf{mo}}(V, D)\ :-\ \textit{typeOf}(V, D_T)$$

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 *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 $P_{data}$, which are no longer in generic Datalog but use tool-specific built-in predicates of the underlying inference engine. The program $P_O$ is extended by these rules as follows.

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

In addition to datatypes, WSML also supports some predefined predicates on datatypes, such as numeric comparison (see [8] for a full list of WSML datatypes). The definition of the axiom SharePriceFeed_requires_bandwidth from the WSML ontology in Section 2, for example, uses a shortcut of the WSML **numericLessThan** predicate (denoted by $<$). For translation of these special predicates to the corresponding tool-specific built-in predicates supported by the underlying Datalog reasoner, we introduce a new tool-specific transformation step $\tau_{dpred}$ as a mapping $\mathcal{P} \rightarrow \mathcal{P}$. This affects the transformation pipeline $\tau$ as follows.

$$\tau = \tau_{dpred} \circ \tau_{dlog} \circ \tau_{lt} \circ \tau_{n} \circ \tau_{axioms}$$

In summary, 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.

## 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 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 derive debugging-relevant information.

### 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. 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 $C[a$ **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 **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 separate querying.)

*Maximum Cardinality Violation* – A maximum cardinality constraint of the form **concept** $C\ a\ (\mathbf{0}\ n)$, 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 **axiom** $Ax_{ID}$ **definedBy !-** $B$, 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. 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.

### 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

$$\tau = \tau_{dpred} \circ \tau_{dlog} \circ \tau_{lt} \circ \tau_{n} \circ \tau_{debug} \circ \tau_{axioms}$$

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 3 shows the detailed replacements performed by $\tau_{debug}$ for the different kinds of constraints.

Minimal cardinality constraints (with bodies $B_{mincard}$) and maximal cardinality constraints (with bodies $B_{maxcard}$) are transformed to rules by keeping their respective bodies and adding a head that instantiates one of the predicates $p_{\mathsf{v\_mincard}}$ and $p_{\mathsf{v\_maxcard}}$ 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 $B_{user}$ and including a head that instantiates the predicate $p_{\mathsf{v\_user}}$ to indicate a user-defined violation. The only argument for the predicate $p_{\mathsf{v\_user}}$ 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 $P_{meta}$. In order to avoid the modification of $P_{meta}$ in the reasoning framework, such molecules are expanded by $\tau_{debug}$, as shown in Table 3.[5]

To maintain the semantics of the replaced constraints, an additional set of meta-level axioms $P_{debug} \subset \mathcal{P}$ is included for reasoning. The rules in $P_{debug}$ have the form $\Box \ :- p_{\mathsf{v}}$ and derive the empty clause for any type and occurrence of a constraint violation.

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

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

Occurrences of constraint violations can be recognized by querying $P_O$ for instantiations of the various debugging-specific meta-level predicates $p_{\mathsf{v\_otype}}$, $p_{\mathsf{v\_mincard}}$, $p_{\mathsf{v\_maxcard}}$ and $p_{\mathsf{v\_user}}$. For example, the set

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

contains tuples for all occurrences of attribute type violations in $P_O$, 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.

[5] After this expansion of **ofType** molecules, the respective axiom (4) in $P_{meta}$ for realising the semantics of attribute type constraints does not apply anymore.

| *Constraint* | *Rule* |
|---|---|
| $\tau_{debug}(\{E_1,\ldots,E_n\})$ | $\{\tau_{debug}(E_1),\ldots,\tau_{debug}(E_n)\}$ |
| $\tau_{debug}(\,!\!-\,B_{mincard}.)$ | $p_{\mathsf{v\_mincard}}(a,I) :- B_{mincard}.$ |
| $\tau_{debug}(\,!\!-\,B_{maxcard}.)$ | $p_{\mathsf{v\_maxcard}}(a,I) :- B_{maxcard}.$ |
| $\tau_{debug}(\,!\!-\,B_{user}.)$ | $p_{\mathsf{v\_user}}(Ax_{ID}) :- B_{user}.$ |
| $\tau_{debug}(C[a \textbf{ ofType } T].)$ | $p_{\mathsf{v\_otype}}(a,T,I,V) :-$ $C[a \textbf{ ofType } T]$ **and** $I$ **memberOf** $C$ **and** $I[a \textbf{ hasValue } V]$ **and naf** $V$ **memberOf** $T$. |

**Table 3.** Replacing constraints by rules.

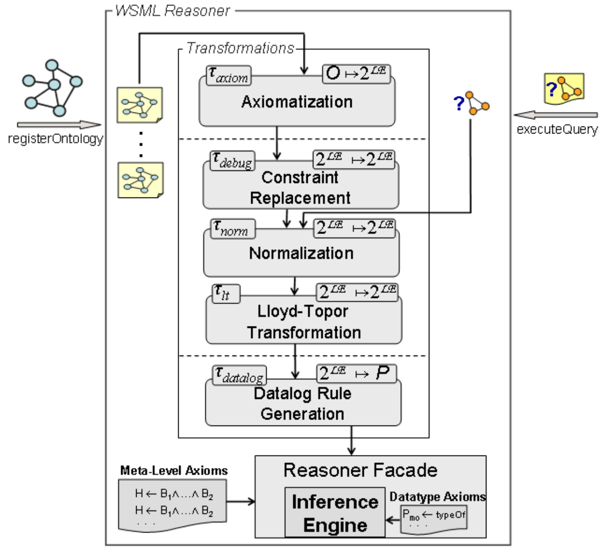
WSML Reasoner
Transformations
registerOntology
executeQuery
$\tau_{axiom}$
$O \mapsto 2^{\mathcal{LE}}$
Axiomatization
$\tau_{debug}$
$2^{\mathcal{LE}} \mapsto 2^{\mathcal{LE}}$
Constraint Replacement
$\tau_{norm}$
$2^{\mathcal{LE}} \mapsto 2^{\mathcal{LE}}$
Normalization
$\tau_{lt}$
$2^{\mathcal{LE}} \mapsto 2^{\mathcal{LE}}$
Lloyd-Topor Transformation
$\tau_{datalog}$
$2^{\mathcal{LE}} \mapsto P$
Datalog Rule Generation
Meta-Level Axioms
$H \leftarrow B_1 \wedge \ldots \wedge B_2$
$H \leftarrow B_1 \wedge \ldots \wedge B_2$
Reasoner Facade
Inference Engine
Datatype Axioms
$P_{mo} \leftarrow$ typeOf

**Fig. 3.** Internal framework architecture.

## 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 realizing WSML on top of a generic Datalog layer, we have also reduced the effort of integrating other reasoners to a minimum The presented framework has been fully implemented in Java and can be downloaded and tested online at `http://dev1.deri.at/wsml2reasoner`.

**Architecture and Internal Layering.** Figure 3 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 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 4; if excluded, the constraints remain in the resulting Datalog program and are mapped to native constraints of the underlying reasoning engine.

The core component of the framework is an exchangeable Datalog inference engine wrapped by a reasoner facade which embeds it in the framework infrastructure. This facade mediates between the generic Datalog program produced in the transformations and the external engine's tool-specific Datalog implementation and built-in predicates.

**Interface and Integration with Existing Technology.** Our framework is based on the WSMO4J (`http://wsmo4j.sourceforge.net`) project, which provides an API for the programmatic handling of WSML documents. WSMO4J performs the task of parsing and validating WSML ontologies and provides the source object model for our translations. For a reasoner to be connected to the Framework, a small adapter class needs to be written, that translates generic Datalog elements to their equivalent constructs within the internal representation layer of the underlying reasoner. Our framework currently comes with facades for two built-in reasoners: KAON2 (`http://kaon2.semanticweb.org`) and MINS (`http://dev1.deri.at/mins`). The initial development was done with the KAON2 inference engine that, with respect to the challenges for datatype reasoning, provides a very flexible type system that allows for user-defined datatypes, together with predicates on these datatypes, including type checking predicates. However, KAON2 cannot be used for reasoning in WSML-Rule as it does not support function symbols and unsafe rules. The second reasoner, MINS, can be used for the WSML-Rule variant but has limited support for datatype reasoning. (For determining the WSML variant of an ontology, one can use the validation facilities built into WSMO4J .)

## 6 Conclusion & Outlook

We have presented a framework for reasoning in rule-based WSML that builds on a mapping to Datalog and on querying a generic Datalog layer. The single well-defined transformation steps can be reused across various adaptations for different scenarios in a highly modular way. We have incorporated debugging features by replacing native constraints with rules to derive debugging-relevant information that can be queried by an ontology engineer. We have implemented our framework with two existing reasoner tools, namely KAON2 and MINS, as alternative implementations of the generic Datalog layer, by which we provide the first available reasoning system for the WSML language.

While the current framework focuses on WSML-Core, -Flight and -Rule, efforts are ongoing to extend the transformations to disjunctive Datalog and description logics. The KAON2 system natively supports disjunctive Datalog and DL reasoning, the latter even extended by WSML-Flight-like rules. Also the DLV system [4] (implementing disjunctive Datalog under the stable model semantics) can be used to realise a similar reasoning. Furthermore, we plan to integrate the KRHyper system [16], which allows reasoning with disjunctive logic programs with stratified default negation. Transformations to DL additionally allow to incorporate description logic system APIs to support efficient reasoning with WSML-DL.

**Acknowledgements.** We would like to thanks our colleagues that wrote [8], especially Jos de Bruijn, for fruitful discussions on the approach discussed here, as well as the contribution of mapping definitions.

## References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook*. Cambridge University Press, 2003.
3. D. Brickley and R. V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. Recommendation 10 February 2004, W3C, 2004.
4. S. Citrigno, T. Eiter, W. Faber, G. Gottlob, C. Koch, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The DLV System: Model Generator and Advanced Frontends. In *Workshop LP*, 1997.
5. M. Dahr. *Deductive Databases: Theory and Applications*. International Thomson Publishing, December 1996.
6. J. de Bruijn, H. Lausen, A. Polleres, and D. Fensel. The Web Service Modeling Language WSML: An Overview. In *Proc. of the 3rd Euro. Semantic Web Conference (ESWC)*, 2006.
7. J. de Bruijn, A. Polleres, R. Lara, and D. Fensel. OWL DL vs. OWL Flight: Conceptual Modeling and Reasoning on the Semantic Web. In *Proceedings of the 14th International World Wide Web Conference (WWW2005)*, Chiba, Japan, 2005. ACM.
8. J. de Bruin. The Web Service Modeling Language (WSML) Specification. Tech. Report, Digital Enterprise Research Institute (DERI), Feb. 2005. http://www.wsmo.org/TR/d16/.
9. M. Dean and G. Schreiber, editors. *OWL Web Ontology Language Reference*. 2004. W3C Recommendation 10 February 2004.
10. A. V. Gelder, K. Ross, and J. S. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, 38(3):620–650, 1991.
11. B. Grosof, I. Horrocks, R. Volz, and S. Decker. Description Logic Programs: Combining Logic Programs with Description Logics. In *Proceedings of WWW-2003*, 2003.
12. M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *JACM*, 42(4):741–843, 1995.
13. J. Lloyd and R. Topor. Making Prolog More Expressive. *Journal of Logic Programming*, 3:225–240, 1984.
14. J. W. Lloyd. *Foundations of Logic Programming; (2nd extended ed.)*. Springer, New York, NY, USA, 1987. ISBN 3-540-18199-7.
15. D. Roman, U. Keller, H. Lausen, R. L. Jos de Bruijn, M. Stollberg, A. Polleres, C. Feier, C. Bussler, and D. Fensel. Web Service Modeling Ontology. *Applied Ontology*, 1(1):77–106, 2005.
16. C. Wernhard. System Description: KRHyper. Technical report, Fachberichte Informatik 14–2003, Universitat Koblenz-Landau, Institut fur Informatik., 2003.