A Reasoning Framework for Rule-Based WSML

Stephan Grimm[1], Uwe Keller[2], Holger Lausen[2], and Gábor Nagypál[1]
order of authors is currently alphabetic

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

Abstract. 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. By a use case example, we show how the framework can be applied in a Semantic Web Services setting.

1 Motivation

– motivate the provision of reasoning support for (rule-based) WSML based on existing inference engines
– stress that this is the first specific WSML Reasoner implementation available

2 Reasoning in the WSML Language

– give an overview of the WSML language; focus on semantics of rule-based variants (also show how syntax looks like, e.g. by an example)
– focus on the ontology language part of WSML (only briefly mention WS-specific parts) and sketch its features, such as constraints, datatypes, conceptual

modelling + axiomatic formulations, ...
– describe the reasoning tasks in WSML, i.e. KB satisfiability and entailment
– relate language features and reasoning, e.g. constraints and satisfiability, to emphasise the close connection of these features to reasoning

The Web Service Modeling Language (WSML) is a language for the specification of different aspects of Semantic Web Services. It provides a formal language for the Web Service Modeling Ontology WSMO which is based on well-known logical formalisms, specifying one coherent language framework for the semantic description of Web Services, starting from the intersection of Datalog and the Description Logic $\mathcal{SHIQ}$. This core language is extended in the directions of Description Logics and Logic Programming in a principled manner with strict layering. WSML distinguishes between conceptual and logical modeling in order to support users who are not familiar with formal logic, while not restricting the expressive power of the language for the expert user. IRIs play a central role in WSML as identifiers. Furthermore, WSML defines XML and RDF serializations for inter-operation over the Semantic Web.

The Web Service Modeling Language WSML takes into account all aspects of Web Service description identified by WSMO, i.e. Web services, goals, mediators and ontologies. WSML comprises different formalisms in order to investigate their applicability to the description of SWS. It would have been too restrictive to base our effort on existing language recommendations such as OWL [**?**]. A concrete goal in our development of WSML is to investigate the usage of different formalisms, most notably Description Logics and Logic Programming, in the context of Ontologies and Web services. Within this paper we furthermore focus only on the Ontology part of WSML. Note that we expect that reasoning tasks with the other elements of WSMO, e.g. the over capabilities is expected to be reducible to reasoning tasks within ontologies.

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 people 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, namely, axioms in ontologies and capability descriptions in Goals and Web Services.

2.1 Conceptual Syntax

The WSML conceptual syntax for ontologies allows for the modeling of concepts, instances, relations and relation instances. Additionally, an ontology may have non-functional properties and may import other ontologies. We start the description of WSML ontologies with an example which demonstrates the elements of an ontology in Listing 1.1, and detail the elements below.

Listing 1.1. An Example WSML Ontology

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml−syntax/wsml−flight"
namespace {_"http://example.org/bookOntology#",
      dc _"http://purl.org/dc/elements/1.1/"}
ontology _"http://example.org/bookOntology"
  nonFunctionalProperties
    dc#title hasValue "Example Book ontology"
    dc#description hasValue "Example ontology about books and shopping carts"
  endNonFunctionalProperties
    concept book
        title ofType _string
        hasAuthor ofType author
    concept author subConceptOf person
        authorOf inverseOf(hasAuthor) ofType book
    concept cart
      nonFunctionalProperties
        dc#description hasValue "A shopping cart has exactly one id
          and zero or more items, which are books."
      endNonFunctionalProperties
        id ofType (1) _string
        items ofType book
    instance crimeAndPunishment memberOf book
        title hasValue "Crime and Punishment"
        hasAuthor hasValue dostoyevsky

    relation authorship(impliesType author, impliesType document)
      nonFunctionalProperties
        dc#relation hasValue authorshipFromAuthor
      endNonFunctionalProperties

    axiom authorshipFromAuthor
      definedBy
        authorship(?x,?y) :− ?x[authorOf hasValue ?y] memberOf author.
```

Concepts The notion of concepts (sometimes also called '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. The non-functional properties, as well as the attribute definitions, are grouped together in one frame, as can be seen from the example concept book in Listing 1.1.

Attribute definitions can take two forms, namely *constraining* (using **ofType**) and *inferring* (using **impliesType**) attribute definitions[3]. Constraining attribute definitions define a typing constraint on the values for this attribute, similar to integrity constraints in Databases; inferring attribute definitions imply that the type of the values for the attribute is inferred from the attribute definition, similar to range restrictions on properties in RDFS [**?**] and OWL [**?**]. 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.

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

Constraining attribute definitions, as well as cardinality constraints, require closed-world reasoning and are thus not allowed in WSML-Core and WSML-DL. As opposed to features of roles in Description Logics, attribute features such as transitivity, symmetry, reflexivity and inverse attributes are local to a concept in WSML. Thus, none of these features may be used in WSML-Core and WSML-DL. For a motivation on the use of constraining attributes, see [**?**].

Relations Relations in WSML can have an arbitrary arity, may be organized in a hierarchy using **subRelationOf** and the parameters may be typed using parameter type definitions of the form (**ofType** *type*) and (**impliesType** *type*), where *type* is a concept identifier. The usage of **ofType** and **impliesType** correspond with the usage in attribute definitions. Namely, parameter definitions with the **ofType** keyword are used to check the type of parameter values, whereas parameter definitions with the **impliesType** keyword are used to infer concept membership of parameter values.

The allowed arity of the relation may be constrained by the underlying logic of the WSML language variant. WSML-Core and WSML-DL allow only binary relations and, similar to attribute definitions, they allow only parameter typing using the keyword **impliesType**.

Instances A concept may have a number of instances associated with it. Instances explicitly specified in an ontology are those which are shared as part of the ontology. However, most instance data exists outside the ontology in private databases. WSML does not prescribe how to connect such a database to an ontology, since different organizations will use the same ontology to query different databases and such corporate databases are typically not shared.

An instance may be member of zero or more concepts and may have a number of attribute values associated with it, see for example the instance crimeAndPunishment in Listing 1.1. Note that the specification of concept membership 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, since without concept membership and constraints on the use of attributes, instances form a directed labelled graph. Because of this possibility to capture semi-structured data, most RDF graphs can be represented as WSML instance data, and vice versa.

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. The axiom authorshipFromAuthor in Listing 1.1 states that the relation authorship exists between any author and any book of which he is an author; consequently, $\langle \mathsf{dostoyesksy}, \mathsf{crimeAndPunishment} \rangle$ is in the relation authorship. Logical expressions are explained in more detail in Section 2.2.

2.2 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 style, in the sense that it has constants, function symbols, variables, predicates and the usual logical connectives. Furthermore, WSML has F-Logic [**?**] based extensions in order to model concepts, attributes, attribute definitions, and subconcept and concept membership 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, followed by an arbitrary number of alphanumeric characters, e.g., ?x, ?name, ?123. Free variables in WSML (i.e., variables which are not explicitly quantified), are implicitly universally quantified outside of the formula (i.e., the logical expression in which the variable occurs is the scope of quantification), unless indicated otherwise, through the **sharedVariables** construct (see the previous Section).

Terms are either identifiers, variables, or constructed terms. An atom is, as usual, a predicate symbol with a number of terms as arguments. Besides the usual atoms, WSML has a special kind of atoms, called *molecules*, which are used to capture information about concepts, instances, attributes and attribute values. The are two types of molecules, analogous to F-Logic:

- An *isa* molecule is a concept membership molecule of the form *A* **memberOf** *B* or a subconcept molecule of the form *A* **subConceptOf** *B* with *A* and *B* arbitrary terms
- An *object* molecule is an attribute value expressions of the form *A*[*B* **hasValue** *C*], a constraining attribute signature expression of the form *A*[*B* **ofType** *C*], or an inferring attribute signature expression of the form *A*[*B* **ofType** *C*], with *A*,*B*,*C* arbitrary terms

WSML has the usual first-order connectives: the unary negation operator **neg**, and the binary operators for conjunction **and**, disjunction **or**, right implication **implies**, left implication **impliedBy**, and dual implication **equivalent**. Variables may be universally quantified using **forall** or existentially quantified using **exists**. First-order formulae are obtained by combining atoms using the mentioned connectives in the usual way. The following are examples of First-Order formulae in WSML:

```
//every person has a father
 forall ?x (?x memberOf Person implies exists ?y (?x[father hasValue
?y])).
//john is member of a class which has some attribute called 'name'
exists ?x,?y (john memberOf ?x and ?x[name ofType ?y]).
```

Apart from First-Order formulae, WSML allows the use of the negation-as-failure symbol **naf** on atoms, the special Logic Programming implication symbol **:-** and the integrity constraint symbol **!-**. A logic programming rule consists of a *head* and a *body*, separated by the **:-** symbol. An integrity constraint consists of the symbol **!-** followed by a rule body. Negation-as-failure **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: **and**, **implies**, **impliedBy**, and **equivalent**. The following connectives are allowed in the body of a rule (or constraint): **and**, **or**, and **naf**. The following are examples of LP rules and database constraints:

```
//every person has a father
?x[father hasValue f(?y)] :- ?x memberOf Person.
//Man and Woman are disjoint
!- ?x memberOf Man and ?x memberOf Woman.
//in case a person is not involved in a marriage, the person is a bachelor
?x memberOf Bachelor :- ?x memberOf Person and naf
Marriage(?x,?y,?z).
```

2.3 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 1 presents a number of language features and indicates in which variant the feature can occur.

Feature	Core	DL	Flight	Rule	Full
Classical Negation (**neg**)	-	X	-	-	X
Existential Quantification	-	X	-	-	X
(Head) Disjunction	-	X	-	-	X
n-ary relations	-	-	X	X	X
Meta Modeling	-	-	X	X	X
Default Negation (**naf**)	-	-	X	X	X
LP implication	-	-	X	X	X
Integrity Constraints	-	-	X	X	X
Function Symbols	-	-	-	X	X
Unsafe Rules	-	-	-	X	X

Table 1. WSML Variants and Feature Matrix

- *WSML-Core* allows only first-order formulae which can be translated to the DLP subset of $\mathcal{SHIQ}(\mathbf{D})$ [**?**]. 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 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 $\mathcal{SHIQ}(\mathbf{D})$. 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 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.

The separation between conceptual and logical modeling allows for an easy adoption by non-experts, since the conceptual syntax does not require expert knowledge in logical modeling, whereas complex logical expressions require more familiarity and training with the language. Thus, WSML allows the modeling of different aspects related to Web services on a conceptual level, while still offering the full expressive power of the logic underlying the chosen WSML variant. Part of the conceptual syntax for ontologies has an equivalent in the logical syntax. This correspondence is used to define the semantics of the conceptual syntax. Notice that, since only parts of the conceptual syntax are mapped to the logical syntax, only a part of the conceptual syntax has a semantics in the logical language for ontologies. For example, non-functional properties are not translated (hence, the name 'non-functional'). The translation between the conceptual and logical syntax is sketched in Table 2.

Conceptual	**Logical**
concept A subConcepOf B	A **subConceptOf** B.
concept A B **ofType** (0 1) C	A[B **ofType** C]. !− ?x **memberOf** A and ?x[B **hasValue** ?y, B **hasValue** ?z] and ?y != ?z.
concept A B **ofType** C	A[B **ofType** C].
relation A/n **subRelationOf** B	A(x_1,...,x_n) **implies** B(x_1,...,x_n)
instance A **memberOf** B C **hasValue** D	A **memberOf** B. A[C **hasValue** D].

Table 2. Translating conceptual to logical syntax

3 Realizing WSML Reasoning through a Mapping to Datalog

– briefly sketch the idea of reasoning via rule based inferencing

The semantics of rule-based WSML is defined via a mapping to datalog with (in)equality and integrity constraints. **sgr: probably use a special denotation like *datalog*$^{!=,IC}$, which then has to be introduced in Section 2** To make use of existing rule engines, the reasoning framework performs various transformations to convert an original ontology in WSML syntax into datalog rules. To maintain the semantics of more complex WSML language constructs that cannot directly be expressed in datalog, a fixed set of rules form the meta-level axioms that realise part of the WSML semantics during reasoning. Based on the transformed ontology, the WSML reasoning tasks of knowledge base satisfiability and instance retrieval are realised by datalog querying via calls to an underlying datalog inference engine that is fed with the rules produced during transformation together with the meta-level axioms.

3.1 Transforming WSML to Datalog

– describe different transformation steps

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

Axiomatization – In a first step, the transformation τ_{axioms} is applied as a mapping $\mathcal{O} \mapsto 2^{\mathcal{LE}}$ from the set of all valid rule-based WSML ontologies to the powerset of all logical expressions that conform to rule-based WSML. τ_{axioms} converts all conceptual syntax elements, such as concept and attribute definitions or cardinality and type constraints, into appropriate logical expressions according to [1](Table 8.1). **sgr: give the complete conversion table instead of the following example ??**

To give an example, the WSML fragment

```
concept C subConceptOf D
    r ofType (0 2) T
instance a memberOf C
    r hasValue b,c
```

is translated by τ_{axioms} to the following logical expressions.

```
C subConceptOf D. C[r ofType T].
!− ?x memberOf C and ?x[r hasValue?y1, r hasValue ?y2] and ?y1 != ?y2.
a memberOf C. a hasValue b,c.
```

Normalization – The transformation τ_{norm} is applied to normalize WSML logical expressions as a mapping $2^{\mathcal{LE}} \mapsto 2^{\mathcal{LE}}$. This normalization step reduces the complexity of WSML logical expressions according to [1](Section 8.2) to make them better fit the simple syntactic form of literals in datalog rules. This reduction includes conversion to negation and disjunctive normal forms as well as decomposition of complex WSML molecules. The various normalization steps are shown in Table 3.1. **sgr: include this table or rather skip it??** After τ_{norm} has been applied, the resulting WSML logical expressions have the form of logic programming rules with no deep nesting of logical connectives.

original expression	*normalized expression*
$\tau_{norm}(X$ **and** $(Y$ **or** $Z).)$	$\tau_{norm}(X)$ **and** $\tau_{norm}(Y)$ **or** $\tau_{norm}(X)$ **and** $\tau_{norm}(Z).$
...	
$\tau_{norm}(X$ **implies** $Y.)$	$\tau_{norm}(Y) :- \tau_{norm}(X).$
$\tau_{norm}(X$ **impliedBy** $Y.)$	$\tau_{norm}(X) :- \tau_{norm}(Y).$
...	

Table 3. Normalization of WSML logical expressions.

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

Table 4. Lloyd-Topor transformations.

Lloyd-Topor Transformation – The transformation τ_{lt} is applied as a mapping $2^{\mathcal{LE}} \mapsto 2^{\mathcal{LE}}$ to flatten the complex WSML logical expressions, producing simple rules according to the Lloyd-Topor transformations [2], as shown in Table 3.1. **sgr: is the specification of the lloyd-topor transformations correct? (esp. the middle one with nested LP-rule and the lack of parenthesis)** 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 $\tau_{datalog}$ is applied as a mapping $2^{\mathcal{LE}} \mapsto \mathcal{P}$ from all valid logical expressions in rule-based WSML to the set of all datalog programs, yielding generic datalog rules that represent the content of the original WSML ontology. In this generic datalog program, 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 a further subsection. Table 3.1 shows the mapping from WSML logical expressions to datalog including the meta-level predicates p_{sco}, p_{mo}, p_{hval}, p_{itype} and p_{otype} that represent their respective WSMl language constructs as can be seen from the mapping.

Ultimately, the basic[4] transformation pipeline for converting a rule-based WSML ontology into a datalog program is the following, constituted by the single transformation steps introduced before.

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

As a mapping $\mathcal{O} \mapsto \mathcal{P}$, this chaining 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.

[4] In Section 4 the transformation pipeline is modified to support debugging features.

3.2 Realising WSML Semantics through Meta Axioms

– describe how a fixed set of rules implements (part of) the WSML semantics during reasoning
– – each WSMl entity is mapped to a datalog constant
– – special meta-level predicates stand for specific WSMl constructs with a certain semantics; they are applied to datalog constants (give example in picture)
– – a direct mapping would not facilitate metamodelling as a feature of WSML
– – meta-level axioms assure that the proper semantics of the wSMl constructs is maintained
– – the meta-level axioms form rules for the meta-level predicates (, which appear in these rules)
– – explain the intuition behind the various 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 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 predicates that appear in Table 3.1, 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_{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_{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_{mo}, in which case it is interpreted as both an instance and a concept.**sgr: does this become clear or is there more that needs to be said?**

A fixed set P_{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 3.2 shows the rules that make up the meta-level axioms in P_{meta}. Axiom (1) realizes transitivity for the WSML **subConceptOf** construct, while axiom (2) ensures that an instance of a subconcept

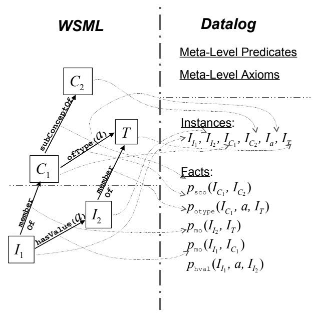
WSML
Datalog
Meta-Level Predicates
Meta-Level Axioms
C_2
T
C_1
I_2
I_1
subConceptOf
ofType(a)
memberOf
hasValue(a)
Instances:
$I_{I_1}, I_{I_2}, I_{C_1}, I_{C_2}, I_a, I_T$
Facts:
$p_{\mathtt{sco}}(I_{C_1}, I_{C_2})$
$p_{\mathtt{otype}}(I_{C_1}, a, I_T)$
$p_{\mathtt{mo}}(I_{I_2}, I_T)$
$p_{\mathtt{mo}}(I_{I_1},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

Fig. 1. Meta-level predicates and axioms for realising the WSML semantics.

is also an instance of its superconcepts. Axiom (3) realizes the semantics for the **implisType** 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 Mapping WSML Reasoning Tasks to Datalog Querying

– describe how to realise WSML satisfiability and entailment through datalog querying
– – characterize the KB (datalog program) on which reasoning is performed with the different facts and rules
– – show how the WSML reasoning tasks are mapped to datalog queries (KB sat., entailment and conjunctive query answering)

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 build up, consisting of the meta-level axioms togeter with the transformed ontology. The different WSML reasoning tasks are then realized by performing datalog queries on P_O, as follows.

O is satisfiable	$(P_O, ?dummyQuery) \rightarrow true$
$O \models \phi_g$	$(P_O, ?\phi_g) \rightarrow true$
$\{\boldsymbol{X} : O \models Q(\boldsymbol{X})\}$	$\{\boldsymbol{X} : (P_O, ?Q(\boldsymbol{X}) \rightarrow true)\}$

(ϕ_g : ground fact ; $\boldsymbol{X}$: variable binding)

3.4 Realising Datatype Reasoning

– describe how reasoning with datatypes is realised

4 Debugging Support

– briefly motivate debugging features for the ontology engineering process

4.1 Debugging Features to Support Ontology Development

– describe the kind of debugging features the framework supports and what they allow for

4.2 Realisation of Debugging Features through Meta Axioms

– describe how these debugging features are realised via an additional fixed set of rules

5 Reasoning Framework Overview

5.1 Architecture and Internal Layering

– present a picture that shows the layering of the framework's internal components
– emphasise the high modularisation w.r.t. use of external inference engines and also internal extendibility

5.2 Interface and Integration with Existing Technology

– mention the connection to the WSMO4J project via the use of the API
– say something about the integrated/supported inference engines KAON2 and MINS

6 A Use Case Example for Reasoning with WSML Ontologies

– present a WSML example ontology that demonstrates the style of modelling in rule-based WSML, to give the reader an impression what he can do with the reasoning framework

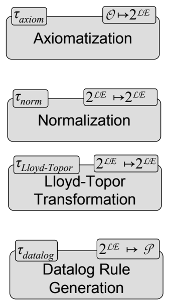
τ_{axiom}
$\mathcal{O} \mapsto 2^{\mathcal{LE}}$
Axiomatization
τ_{norm}
$2^{\mathcal{LE}} \mapsto 2^{\mathcal{LE}}$
Normalization
$\tau_{Lloyd\text{-}Topor}$
$2^{\mathcal{LE}} \mapsto 2^{\mathcal{LE}}$
Lloyd-Topor Transformation
$\tau_{datalog}$
$2^{\mathcal{LE}} \mapsto \mathcal{P}$
Datalog Rule Generation

Fig. 2. Layering of the internal architecture.

– – how do constraints work
– – how do derivation rules work
– – how does debugging support ontology development
– a candidate example is the WP8 telecom bundle scenario from the last DIP review

7 Outlook

– mention the planned use of KAON2's DL-capabilities for future extensions in the direction of WSML-DL
– . . .

References

1. Jos de Bruin. The Web service Modeling Language (WSML) Specification. Technical report, Digital Enterprise Research Institute (DERI), February 2005. http://www.wsmo.org/TR/d16/.
2. John Lloyd and Rodney Topor. Making Prolog More Expressive. *Journal of Logic Programming*, 3:225–240, 1984.

WSML	*Datalog*
$\tau_{datalog}(\{E_1,\ldots,E_n\})$	$\{\tau_{datalog}(E_1),\ldots,\tau_{datalog}(E_n)\}$
$\tau_{datalog}($!- $B.)$	$\leftarrow \tau_{datalog}(B)$
$\tau_{datalog}(H$:- $B.)$	$\tau_{datalog}(H) \leftarrow \tau_{datalog}(B)$
$\tau_{datalog}(X$ **and** $Y.)$	$\tau_{datalog}(X) \wedge \tau_{datalog}(Y) \leftarrow$
$\tau_{datalog}(C$ **subConceptOf** $D.)$	$p_{\mathsf{sco}}(C,D) \leftarrow$
$\tau_{datalog}(I$ **memberOf** $C.)$	$p_{\mathsf{mo}}(I,C) \leftarrow$
$\tau_{datalog}(I[a$ **hasValue** $V].)$	$p_{\mathsf{hval}}(I,a,V) \leftarrow$
$\tau_{datalog}(C[a$ **impliesType** $T].)$	$p_{\mathsf{itype}}(C,a,T) \leftarrow$
$\tau_{datalog}(C[a$ **ofType** $T].)$	$p_{\mathsf{otype}}(C,a,T) \leftarrow$
$\tau_{datalog}(\mathbf{r}(X_1,\ldots,X_n).)$	$r(X_1,\ldots,X_n) \leftarrow$
$\tau_{datalog}(X = Y.)$	$X = Y \leftarrow$
$\tau_{datalog}(X$ **!=** $Y.)$	$X \neq Y \leftarrow$

Table 5. Transformation from logical expressions in rule-based WSML to datalog including meta-level predicates.

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

Table 6. Meta-level axioms for WSML semantics in datalog.