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, Karlsruhe, Germany
{grimm,nagypal}@fzi.de
[2] Digital Enterprise Research Institute (DERI), University of Innsbruck, Austria
{uwe.keller,holger.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. 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 [17]. 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 project **R**W^2 (FFG 809250), where it is applied to Semantic Web Services discovery and to support domain modelling in various use case scenarios of eBanking, eGovernment and telecommunications. Thus, the features implemented have a close link to the needs in these use cases driven by industrial partners.

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) [18] and is based on well-known logic-based knowledge representation (KR) formalisms (i.e. Description Logics [1] and Logic Programming [17]), 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 [5] and the Description Logic $\mathcal{SHIQ}(\mathbf{D})$ [13]. 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) [10] 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. We illustrate the description of WSML ontologies with an example in Listing 1.

Listing 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 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[2]. 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 [3] and OWL [9]. 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 ontologies.

Instances of Concepts and Relations. Concepts and relations may have a arbitrary number of instances associated with it. Instances explicitly specified in an ontology are those which are shared as

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

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

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 exampe, 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: it 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 bandwith 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 [15] 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 C_1 **subConceptOf** C_2 (denoting a subconcept relationship) whereby I, C, C_i 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 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 bi-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.

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**.

Axioms BroadbandBundle_sufficient_condition and 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 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.

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 ϕ, 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 $\iota \in ground(\phi)$ of ϕ 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 $retrieve_O(Q) = \{\vec{t} : \vec{t} = (t_1, \ldots, t_n), t_i \in Term(O), O \models Q(\vec{t})\}$. Rule-based WSML is based on the wellfounded-model semantics [11]. 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.

3 Mapping WSML to Datalog

The semantics of rule-based WSML is defined via a mapping to Datalog [5] with (in)equality and integrity constraints, as described in [8]. 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.

conceptual syntax	*logical expression(s)*
$\tau_{axioms}($**concept** C_1 **subConceptOf** C_2 $)$	C_1 **subConceptOf** C_2.
$\tau_{axioms}($**concept** C_1 A **ofType** $(0,1)$ T $)$	C_1[A **ofType** T]. !- ?x **memberOf** C_1 **and** ?x[A **hasValue** ?y, A **hasValue** ?z] **and** ?y != ?z.
$\tau_{axioms}($**concept** C A_1 **inverseOf** A_2 **impliesType** T $)$	C_1[A **impliesType** T]. ?x **memberOf** C **and** ?v **memberOf** T **implies** ?x[A_1 **hasValue** ?v] **equivalent** ?v[A_2 **hasValue** ?x].
$\tau_{axioms}($**relation** R_1/n **subRelationOf** R_2 $)$	$R_1(\vec{x})$ **implies** $R_2(\vec{x})$. where $\vec{x}$ = (x_1,...,x_n)
$\tau_{axioms}($**instance** I **memberOf** C A **hasValue** V $)$	I **memberOf** C. I[A **hasValue** V].

Table 2. Examples for axiomatizing conceptual ontology modeling elements.

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 original ontology.

Axiomatization. In a first step, the transformation τ_{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 constraints, are converted into appropriate axioms specified by logical expressions. Table 2 shows the details of some of the conversions performed by τ_{axioms}, based on [8]. During the transformation, for each expression e in the WSML Ontology $O \in \mathcal{O}$ that matches a pattern on the left-hand side of Table 2, the formulae $\tau_{axioms}(e)$ are created and added to the resulting theory $\tau_{axioms}(O)$.

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 τ_{norm} is applied as a mapping $2^{\mathcal{LE}} \rightarrow 2^{\mathcal{LE}}$ to normalize WSML logical expressions. This normalization step reduces the complexity of WSML logical expressions according to [8, 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 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 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*
$\tau_{\mathit{norm}}(\{E_1, \ldots, E_n\})$	$\{\tau_{\mathit{norm}}(E_1), \ldots, \tau_{\mathit{norm}}(E_n)\}$
$\tau_{\mathit{norm}}(E_x \textbf{ and } E_y.)$	$\tau_{\mathit{norm}}(E_x) \textbf{ and } \tau_{\mathit{norm}}(E_y)$
$\tau_{\mathit{norm}}(E_x \textbf{ or } E_y.)$	$\tau_{\mathit{norm}}(E_x) \textbf{ or } \tau_{\mathit{norm}}(E_y)$
$\tau_{\mathit{norm}}(E_x \textbf{ and } (E_y \textbf{ or } E_z).)$	$\tau_{\mathit{norm}}(\tau_{\mathit{norm}}(E_x) \textbf{ and } \tau_{\mathit{norm}}(E_y) \textbf{ or } \tau_{\mathit{norm}}(E_x) \textbf{ and } \tau_{\mathit{norm}}(E_z).)$
$\tau_{\mathit{norm}}((E_x \textbf{ or } E_y) \textbf{ and } E_z).)$	$\tau_{\mathit{norm}}(\tau_{\mathit{norm}}(E_x) \textbf{ and } \tau_{\mathit{norm}}(E_z) \textbf{ or } \tau_{\mathit{norm}}(E_y) \textbf{ and } \tau_{\mathit{norm}}(E_z).)$
$\tau_{\mathit{norm}}(\textbf{ naf } (E_x \textbf{ and } E_y).)$	$\textbf{naf } \tau_{\mathit{norm}}(E_x) \textbf{ or } \textbf{ naf } \tau_{\mathit{norm}}(E_y).$
$\tau_{\mathit{norm}}(\textbf{ naf } (E_x \textbf{ or } E_y).)$	$\textbf{naf } \tau_{\mathit{norm}}(E_x) \textbf{ and } \textbf{ naf } \tau_{\mathit{norm}}(E_y).$
$\tau_{\mathit{norm}}(\textbf{ naf } (\textbf{ naf } E_x).)$	$\tau_{\mathit{norm}}(E_x)$
$\tau_{\mathit{norm}}(E_x \textbf{ implies } E_y.)$	$\tau_{\mathit{norm}}(E_y) :- \tau_{\mathit{norm}}(E_x).$
$\tau_{\mathit{norm}}(E_x \textbf{ impliedBy } E_y.)$	$\tau_{\mathit{norm}}(E_x) :- \tau_{\mathit{norm}}(E_y).$
$\tau_{\mathit{norm}}(X[Y_1, \ldots, Y_n].)$	$X[Y_1] \textbf{ and } \ldots \textbf{ and } X[Y_n].$

Table 3. Normalization of WSML logical expressions.

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

Table 4. Lloyd-Topor transformations.

Lloyd-Topor Transformation. The transformation τ_{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 [16], as shown in Table 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 [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 τ_{datalog} 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. Table 5 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. 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.

WSML	*Generic Datalog*
$\tau_{\mathit{datalog}}(\{E_1, \ldots, E_n\})$	$\{\tau_{\mathit{datalog}}(E_1), \ldots, \tau_{\mathit{datalog}}(E_n)\}$
$\tau_{\mathit{datalog}}(\ !-\ B.)$	$\Box\ :- \tau_{\mathit{datalog}}(B)$
$\tau_{\mathit{datalog}}(H.)$	$\tau_{\mathit{datalog}}(H)\ .$
$\tau_{\mathit{datalog}}(H\ :-\ B.)$	$\tau_{\mathit{datalog}}(H)\ :- \tau_{\mathit{datalog}}(B)$
$\tau_{\mathit{datalog}}(E_x \textbf{ and } E_y.)$	$\tau_{\mathit{datalog}}(E_x) \wedge \tau_{\mathit{datalog}}(E_y)$
$\tau_{\mathit{datalog}}(\textbf{naf } E.)$	$\sim \tau_{\mathit{datalog}}(E)$
$\tau_{\mathit{datalog}}(C_x \textbf{ subConceptOf } C_y.)$	$p_{\mathsf{sco}}(C_x, C_y)$
$\tau_{\mathit{datalog}}(I \textbf{ memberOf } C.)$	$p_{\mathsf{mo}}(I, C)$
$\tau_{\mathit{datalog}}(I[a \textbf{ hasValue } V].)$	$p_{\mathsf{hval}}(I, a, V)$
$\tau_{\mathit{datalog}}(C[a \textbf{ impliesType } T].)$	$p_{\mathsf{itype}}(C, a, T)$
$\tau_{\mathit{datalog}}(C[a \textbf{ ofType } T].)$	$p_{\mathsf{otype}}(C, a, T)$
$\tau_{\mathit{datalog}}(\textbf{r}(X_1, \ldots, X_n).)$	$r(X_1, \ldots, X_n)$
$\tau_{\mathit{datalog}}(X \textbf{ = } Y.)$	$X = Y$
$\tau_{\mathit{datalog}}(X \textbf{ != } Y.)$	$X \neq Y$

Table 5. Transformation WSML logical expressions to Datalog.

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[3] transformation τ for converting a rule-based WSML ontology into a Datalog program based on the the single transformation steps introduced before by $\tau = \tau_{\mathit{datalog}} \circ \tau_{\mathit{lt}} \circ \tau_{\mathit{norm}} \circ \tau_{\mathit{axioms}}$.

As a mapping $\tau : \mathcal{O} \rightarrow \mathcal{P}$, this concatenation 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 predicates that appear in Table 5, and which are applied to the datalog constants that represent the WSML entities. The facts listed

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

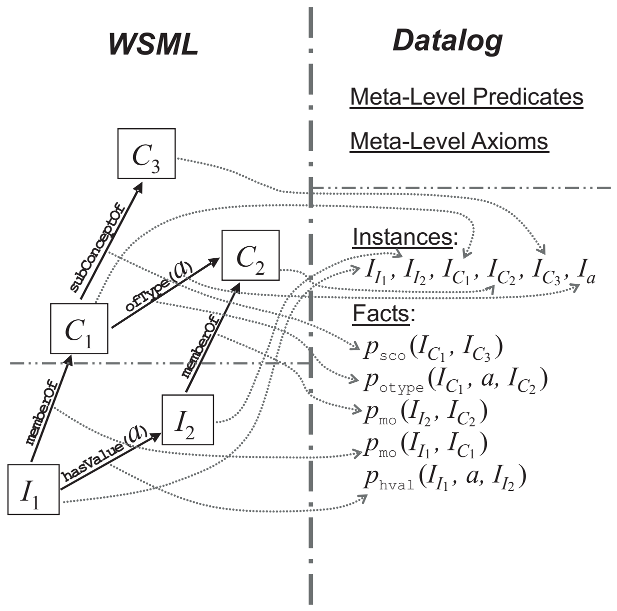
WSML
Datalog
Meta-Level Predicates
Meta-Level Axioms
C_3
C_2
C_1
I_2
I_1
subConceptOf
ofType(a)
memberOf
memberOf
hasValue(a)
Instances:
$I_{I_1}, I_{I_2}, I_{C_1}, I_{C_2}, I_{C_3}, I_a$
Facts:
$p_{\mathtt{sco}}(I_{C_1}, I_{C_3})$
$p_{\mathtt{otype}}(I_{C_1}, a, I_{C_2})$
$p_{\mathtt{mo}}(I_{I_2}, I_{C_2})$
$p_{\mathtt{mo}}(I_{I_1}, I_{C_1})$
$p_{\mathtt{hval}}(I_{I_1}, a, I_{I_2})$

Figure 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)	$\square\ :-\ p_{\mathsf{otype}}(C_1, a, C_2) \wedge p_{\mathsf{mo}}(I, C_1) \wedge p_{\mathsf{hval}}(I, a, V) \wedge \sim p_{\mathsf{mo}}(V, C_2)$

Table 6. Realising WSML semantics in Datalog.

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 at the same time.

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 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 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.

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.

$$O \text{ is satisfiable } \Leftrightarrow\ (P_O,\ ?-\ \square) = \emptyset$$

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.

$$O \models \phi\ \Leftrightarrow\ (P_O,\ ?-\ \tau'(\phi')) \neq \emptyset$$

From the WSML ground fact $\phi \in \mathcal{LE}$ we derive a non-ground formula $\phi' \in \mathcal{LE}$ by replacing the left-most occurrence of a constant by the variable x. ϕ' 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 ϕ 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.

$$retrieve_O(Q)\ =\ (P_O,\ ?-\ \tau'(Q(\vec{x})))$$

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 $Q(?x) =$
?x **memberOf** BroadbandBundle
posed to the ontology in Listing 1 yields the set $\{(MyBundle)\}$ 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 in implementing datatype reasoning is related to Axiom (4) in Table 3.2, 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_{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:

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

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 Datalog program P_O is extended with this new set of 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[4]. For example, the definition of the SharePriceFeed_requires_bandwidth axiom from the WSML ontology in Section 2 uses a shortcut of the WSML **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 τ_{dpred} as a mapping $\mathcal{P} \rightarrow \mathcal{P}$, which translates all predefined WSML datatype predicates in the generic Datalog program to tool-specific built-in predicates. The transformation pipeline τ is augmented by this additional step and is redefined as follows.

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

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.

[4]A full list of WSML datatypes can be found in the WSML specification [8].

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 contain the needed additional information in their heads.

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 querying the ontology separately.)

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 pipeline

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

where the additional transformation step τ_{debug} is applied after the WSML conceptual syntax has been resolved, replacing constraints on the level of WSML logical expressions. Table 7 shows the detailed replacements performed by τ_{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 τ_{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 τ_{debug}, as shown in Table 7.[5]

To maintain the constraining semantics of the replaced constraints, an additional set of meta-level axioms $P_{debug} \in \mathcal{P}$ is included for reasoning. The rules in P_{debug} derive the empty clause for any occurrence of a constraint violation, as shown in Table 8.

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 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

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

Table 7. Replacing constraints by rules.

Debugging Meta-Level Axioms		
(1)	$\Box$	$:- p_{\mathsf{v\_otype}}(a, T, I, V)$
(2)	$\Box$	$:- p_{\mathsf{v\_mincard}}(a, I)$
(3)	$\Box$	$:- p_{\mathsf{v\_maxcard}}(a, I)$
(4)	$\Box$	$:- p_{\mathsf{v\_user}}(Ax_{ID})$

Table 8. Meta-level axioms for debugging.

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[6]. The presented framework has been implemented in Java and can be downloaded at `http://dev1.deri.at/wsml2reasoner` for ready-usage. An online demo is available at `http://tools.deri.org/wsml/rule-reasoner`.

5.1 Architecture and Internal Layering

Figure 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 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 τ_{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 tool-specific Datalog implementation and built-in predicates used by the external inference engine.

[6]In fact, the adaptation of the framework to the MINS rule engine took less then a day.

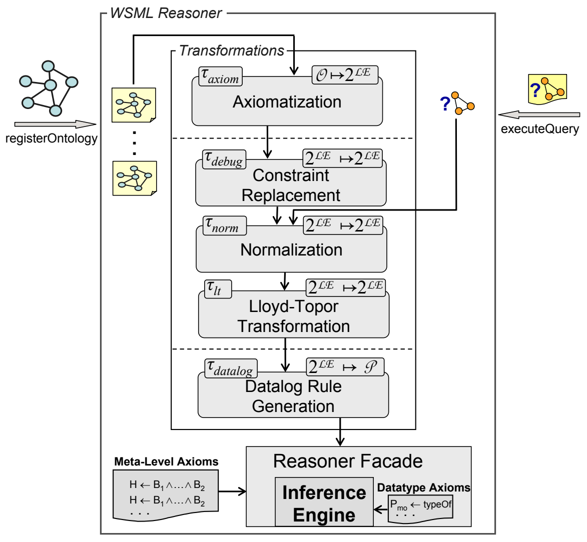
WSML Reasoner
Transformations
registerOntology
executeQuery
τ_{axiom}
$\mathcal{O} \mapsto 2^{\mathcal{LE}}$
Axiomatization
τ_{debug}
$2^{\mathcal{LE}} \mapsto 2^{\mathcal{LE}}$
Constraint Replacement
τ_{norm}
$2^{\mathcal{LE}} \mapsto 2^{\mathcal{LE}}$
Normalization
τ_{lt}
$2^{\mathcal{LE}} \mapsto 2^{\mathcal{LE}}$
Lloyd-Topor Transformation
$\tau_{datalog}$
$2^{\mathcal{LE}} \mapsto \mathcal{P}$
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

Figure 2. Internal framework architecture.

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[7] 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 KAON2 inference engine[8] [14]. As we have seen in Section 3.4, datatype reasoning poses the biggest challenge for the Datalog implementation. KAON2 provides a

[7]`http://wsmo4j.sourceforge.net`

[8]KAON2 is available for download from `http://kaon2.semanticweb.org`

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].

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 as well as our implementation 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 [4] (implementing disjunctive datalog under the stable model semantics) can be used for reasoning the same purpose. Furthermore, we plan to integrate the KRHyper system [19], 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 (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. An additional translation to widely used description logic (DL) system APIs (e.g. DIG [2]) to support efficient reasoning with WSML-DL based on state-of-the-art DL systems like Racer [12], Pellet[11] or Fact++[12] is on the way.

[9]`http://dev1.deri.at/mins/`

[10]A demo of this feature is available at: `http://tools.deri.org/wsml/validator`

[11]`http://www.mindswap.org/2003/pellet/`

[12]`http://owl.man.ac.uk/factplusplus/`

Acknowledgements. This work is supported by the European Commision under the the DIP project (FP6-507483), and by the Austrian Federal Ministry for Transport, Innovation, and Technology under the project **R**W^2 (FFG 809250).

References

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

[2] Sean Bechhofer, Ralf Möller, and Peter Crowther. The DIG Description Logic Interface. In *Proceedings of the 2003 International Workshop on Description Logics (DL2003), Rome, Italy September 5-7*, 2003.

[3] Dan Brickley and Ramanathan V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. Recommendation 10 February 2004, W3C, 2004. Available from http://www.w3.org/TR/rdf-schema/.

[4] Simona Citrigno, Thomas Eiter, Wolfgang Faber, Georg Gottlob, Christoph Koch, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The dlv System: Model Generator and Advanced Frontends (system description). In *Workshop Logische Programmierung*, 1997.

[5] Michael Dahr. *Deductive Databases: Theory and Applications*. International Thomson Publishing, December 1996. Good introductory book on deductive databases.

[6] Jos de Bruijn, Holger Lausen, Axel Polleres, and Dieter Fensel. The Web Service Modeling Language WSML: An Overview. In *Proceedings of the 3rd European Semantic Web Conference (ESWC)*, 2006.

[7] Jos de Bruijn, Axel Polleres, Rubén Lara, and Dieter 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] 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/.

[9] Mike Dean and Guus Schreiber, editors. *OWL Web Ontology Language Reference*. 2004. W3C Recommendation 10 February 2004.

[10] M. Duerst and M. Suignard. Internationalized resource identifiers (IRIs). RFC 3987, IETF, 2005. http://www.ietf.org/rfc/rfc3987.txt.

[11] Allen Van Gelder, Kenneth Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

[12] Volker Haarslev and Ralf Möller. RACER System Description. *Lecture Notes in Computer Science*, 2083:701–??, 2001.

[13] Ian Horrocks, Peter F. Patel-Schneider, and Frank van Harmelen. From SHIQ and RDF to OWL: The making of a web ontology language. *Journal of Web Semantics*, 1(1):7–26, 2003.

[14] Ullrich Hustadt, Boris Motik, and Ulrike Sattler. Reducing SHIQ-Description Logic to Disjunctive Datalog Programs. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Ninth International Conference (KR2004)*, pages 152–162, 2004.

[15] Michael Kifer, Georg Lausen, and James Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *JACM*, 42(4):741–843, 1995.

[16] John Lloyd and Rodney Topor. Making Prolog More Expressive. *Journal of Logic Programming*, 3:225–240, 1984.

[17] John Wylie Lloyd. *Foundations of Logic Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1993.

[18] Dumitru Roman, Uwe Keller, Holger Lausen, Rubén Lara Jos de Bruijn, Michael Stollberg, Axel Polleres, Cristina Feier, Christoph Bussler, and Dieter Fensel. Web Service Modeling Ontology. *Applied Ontology*, 1(1):77–106, 2005.

[19] C. Wernhard. System Description: KRHyper. Technical report, Fachberichte Informatik 14–2003, Universitat Koblenz-Landau, Institut fur Informatik., 2003.