



WSML Deliverable
D16.2 v0.2
**WSML REASONING
IMPLEMENTATION**

WSML Working Draft – October 3, 2005

Authors:

Jos de Bruijn
Cristina Feier
Uwe Keller
Rubén Lara
Axel Polleres
Livia Predoiu^a

^aIn alphabetic order

Editors:

Livia Predoiu
Axel Polleres

Reviewers:

Holger Lausen

This version:

<http://www.wsmo.org/TR/d16/d16.2/v0.2/20050902/>

Latest version:

<http://www.wsmo.org/TR/d16/d16.2/v0.2/>

Previous version:

<http://www.wsmo.org/TR/d16/d16.2/v0.2/20050714/>



Abstract

This deliverable will provide reasoner implementations for the variants of the WSML family of languages.

In its current version, this deliverable lists the requirements posed by the different WSML variants on the reasoner implementation and provides a survey of existing reasoners. The reasoners surveyed in this deliverable range from logic programming engines to description logic reasoners and first-order theorem provers. We describe a preliminary implementation of WSML-Rule on top of OntoBroker and a preliminary implementation of WSML-DL on top of FaCT++. Furthermore a survey of a general WSML reasoning infrastructure is provided which implements WSML-Core and WSML-Flight reasoners based on Mandrax, DLV, KAON2 and MINS.



Contents

1	Introduction	5
2	WSML Variants	6
2.1	WSML-Core	6
2.2	WSML-Flight	7
2.3	WSML-Rule	7
2.4	WSML-DL	7
2.5	WSML-Full	8
3	Survey of Reasoners	9
3.1	Logic Programming	9
3.1.1	SWI-Prolog	9
3.1.2	XSB	10
3.1.3	<i>FLORA-2</i>	11
3.1.4	TRIPLE	12
3.1.5	OntoBroker	13
3.1.5.1	SILRI	14
3.1.6	DLV	14
3.1.7	S MODELS and GNT	16
3.1.8	KAON2	16
3.1.9	Summary	17
3.2	First-order theorem provers	18
3.2.1	Vampire	18
3.2.2	SNARK	21
3.2.3	Summary	23
3.3	Description Logic reasoners	23
3.3.1	FaCT++	23
3.3.2	RACER	24
3.3.3	Pellet	25
3.3.4	Summary	25
3.4	Other Nonmonotonic Reasoning Techniques	25
3.4.1	Default logic	26
3.4.1.1	Default Logic and Logic Programming	27
3.4.1.2	DeReS	29
3.4.1.3	Antoniou's approach for defining and computing extensions of default theories	31
4	Implementation	33
4.1	The WSML reasoner infrastructure	33
4.2	WSML-Core and WSML-Flight reasoners	36
4.3	WSML-Rule and WSML-DL reasoners	37
4.3.1	Architecture	38
4.3.2	WSML-Rule reasoners	39
4.3.3	WSML-Rule to OntoBroker F-Logic and FLORA-2 F-Logic translation	40
4.3.3.1	Limitations	42
4.3.4	WSML-DL reasoner	43
4.3.4.1	WSML-DL to DIG translation	43
4.3.4.2	Limitations	45
4.4	Outlook	46
4.4.1	WSML reasoner architecture	46



4.4.2	Ontobroker	47
4.4.3	WSML-DL reasoners	47
5	Conclusions and Future Work	48
5.1	Related Developments	48
	Bibliography	49



1 Introduction

WSML [de Bruijn et al., 2005] is a family of representation languages for the Semantic Web, taking Description Logics [Baader et al., 2003], Logic Programming [Lloyd, 1987] and First-Order Logic [Fitting, 1996] as Semantic basis, with influences from F-Logic [Kifer et al., 1995] and frame-based representation systems.

In this context, it is the aim of this deliverable to provide a reasoner implementation for the variants of WSML. In the current version of the document, we first survey different reasoner implementations of different logical formalisms and evaluate the usability of the implementations for the different WSML variants. Then we describe a WSML reasoning infrastructure, the first implementation of a WSML-Rule reasoner using OntoBroker and the first implementation of WSML-DL reasoner using FaCT++.

This document is further structured as follows. In Chapter 2 we briefly discuss the different WSML variants and their requirements on a reasoning engine. We present the survey on reasoning implementations in Chapter 3. We present our implementations of WSML-Core, WSML-Flight, WSML-Rule and WSML-DL on top existing reasoners in chapter 4. Finally, we present conclusions and summarize future work and related developments in Chapter 5.



2 WSML Variants

In this chapter we provide a short description of each of the WSML variants, as well as requirements each variant has on the reasoner.

2.1 WSML-Core

WSML-Core is based on the well-known DHL (Description Horn Logic) fragment [Grosz et al., 2003; de Bruijn et al., 2004a], which is that subset of the Description Logic logic $\mathcal{SHIQ}(\mathbf{D})$ (which is close to the language underlying OWL [Horrocks and Patel-Schneider, 2003]) which falls inside the Horn logic fragment of First-Order Logic without equality and without existential quantification.

Two different types of reasoning can be done with WSML-Core, namely: (1) subsumption reasoning and (2) query answering. Subsumption reasoning is equivalent to checking entailment of non-ground formulae and can thus be reduced to checking satisfiability using a First-Order style or a Description Logic-style calculus. Query answering is equivalent to checking entailment of ground facts. Thus query answering can be reduced to satisfiability checking. However, using a First-Order or Description Logic calculus for query answering is not very efficient [de Bruijn et al., 2004a; Hustadt et al., 2004a]. Fortunately, there are well-known techniques for query answering in the area of logic programming and deductive databases [Ullman, 1988].

For subsumption reasoning, the following are the requirements on a reasoner for WSML-Core:

- Subsumption reasoning for WSML-Core can be done through unsatisfiability with Tableaux reasoning, theorem proving or any other technique for checking satisfiability of First-Order theories or query containment¹.
- In order to handle datatypes in WSML-Core, a datatype oracle is required which has a sound and complete procedure for deciding satisfiability of conjunctions of datatype predicates. Such requirements are described in [Pan and Horrocks, 2004].

For query answering, the following are the requirements on a reasoner for WSML-Core:

- A Datalog engine which can handle integrity constraints (for checking datatypes).
- Built-in predicates for handling strings and integers. For integers, basic arithmetic functions (+, -, /, *) should be provided, as well as basic comparison operators (=, \neq , >, <). For strings, at least the (in)equality predicates should be built-in.

¹While query containment for logic programs is in general undecidable, some for restricted forms of logic programs containment can be decided. The simplest form, checking containment of conjunctive queries is well-known to be NP-complete [Chandra and Merlin, 1977]. In [Calvanese et al., 1998; Calvanese et al., 2003] several more expressive query classes and methods to decide query containment for these are discussed. We are however currently not aware of any out-of-the-box implementations for such complex checks.



- The symbols *true* and *false*. The former represents universal truth; the latter represents falsehood. If *false* is derived from the program, the program is inconsistent. These symbols can be eliminated through simple preprocessing steps (cf. [de Bruijn et al., 2004b]).

2.2 WSML-Flight

WSML-Flight extends WSML-Core with the full expressive power of safe Datalog rules, default negation, the use of integrity constraints (note that constraints are already in WSML-Core; however, they are limited to datatype predicates), (in)equality for abstract individuals, and meta-modeling.

The semantics of WSML-Flight is grounded in Logic Programming. Since there exist no efficient implementation of query containment and since this problem is undecidable in general, the only reasoning task we envision for WSML-Flight is query answering (i.e. entailment of ground facts). Notice that subsumption reasoning can always be done for the WSML-Core subset of an ontology.

The additional requirements for a WSML-Flight reasoner over the requirements for a WSML-Core reasoner are:

- Full Datalog support
- Support for (stratified) default negation
- A built-in equality predicate (in the body of the rule)

Equality could also be axiomatized in the program. However, this would seriously degrade performance of query answering. Therefore we state this as a formal requirement.

The other features added in WSML-Flight compared with WSML-Core can be eliminated in a preprocessing step. However, it would be favorable to have also the following features in the reasoner:

- *Meta-modeling* (treating classes as instances, etc.) can be translated to plain Datalog (cf. [de Bruijn et al., 2004a; Hustadt et al., 2004a; Fensel et al., 2000]). However, if meta-modeling were built into the reasoner, less effort is required in the preprocessing step. Also, query answers might have to be rewritten in order to deal with meta modeling (e.g. [Fensel et al., 2000]).

2.3 WSML-Rule

WSML-Rule extends WSML-Flight with function symbols and additionally allows unsafe rules. We furthermore expect an extension of WSML-Rule with unstratified negation under the Well-Founded Semantics [Gelder et al., 1988].

2.4 WSML-DL

WSML-DL is still under development. However, it is envisioned as an alternate syntax for OWL DL and thus *SHOIN(D)*. Thus, both query answering



and subsumption can be done with any of the currently popular DL reasoners, such as FaCT++², RACER³, and Pellet⁴.

Both WSML-Core and WSML-DL are based on the $\mathcal{SHIQ}(\mathbf{D})$ Description Logic. However, WSML-Core corresponds with a restricted subset of $\mathcal{SHIQ}(\mathbf{D})$ which falls in the Horn fragment. Thus, WSML-DL adds the following features to WSML-Core: disjunction, (classical) negation and existential quantification. In terms of complexity, we know that the upper bound for the combined complexity for WSML-Core (not taking into account the datatypes) is in ExpTime [Dantsin et al., 2001], because WSML-Core is a subset of Datalog. Because WSML-Core is strictly less expressive than Datalog, it might be possible to find a lower upper bound. However, we are not aware of any research in this area.

2.5 WSML-Full

The semantics of WSML-Full has not yet been defined. However, it is envisioned to unify First-Order Logic with nonmonotonic negation. We are currently looking into different ways of combining these. A possible direction is to use a nonmonotonic formalism such as circumscription [McCarthy, 1986].

Table 2.1 gives an overview of the features included in each of the language variants.

Feature	Core	DL	Flight	Rule	Full
Classical Negation (neg)	-	X	-	-	X
Existential Quantification	-	X	-	-	X
Disjunction	-	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 2.1: WSML Variants and Feature Matrix

In the remainder of this deliverable we will use these requirements on the reasoner for the different WSML variants to evaluate the suitability of different reasoners for use with WSML.

²<http://owl.man.ac.uk/factplusplus/>

³<http://www.racer-systems.com/>

⁴<http://www.mindswap.org/2003/pellet/index.shtml>



3 Survey of Reasoners

This chapter presents the survey on existing reasoning implementations. Each reasoner implementation is described along the reasoning requirements for each of the WSML variants, which were outlined in chapter 2.

We describe reasoners from the areas of Logic Programming, First-order theorem proving, Description Logic reasoner and general nonmonotonic reasoning implementations.

3.1 Logic Programming

In this section we describe a number of Logic Programming implementations. We are interested which of the WSML variants these implementations are able to deal with and to what extent, i.e. what kind of inferences they support. As one may expect, the reasoners from this category deal very well with query answering for the WSML variants that are included in Logic Programming, namely WSML Core and WSML Flight, but they are not meant to be used as subsumption reasoners. Still, as shown in [Grosz et al., 2003], subsumption reasoning can be reduced to query answering for the subset of DL that intersects LP, named DLP(Description Logic Programs) on which WSML Core is based. A mapping function is used for translating from DL to LP, afterwards different types of DL queries being reduced to LP queries. Thus, LP reasoners can be used for performing subsumption with WSML Core.

3.1.1 SWI-Prolog

Prolog is a logical programming language based on first-order predicate calculus, restricted to allow only Horn clauses. The execution of a Prolog program is an application of theorem proving by first-order resolution. SWI-Prolog¹ is a free Prolog environment [Wielemaker, 2003], licensed under the Lesser GNU Public License.

As an add-on, SWI-Prolog offers support for storing and querying the RDF triple model. It has a library named SWI-Prolog/XPCE Semantic Web Library that offers packages for reading, querying and storing Semantic Web documents as well as XPCE libraries that provide visualization and editing for such documents. These packages are based on the RDF parser from the SGML library of SWI-Prolog.

WSML-Core Since WSML-Core falls inside the Horn-Logic Fragment without equality and without existential quantification, SWI-Prolog has no problems regarding query answering in this language.

Subsumption reasoning within WSMO can also be handled by SWI-Prolog in the manner described in the introduction to this section.

Another possibility for DL reasoning using SWI-Prolog might be using its Semantic Web Library. This is feasible only if an RDF syntax is defined for WSML. For the moment the library offers limited reasoning support for RDFS, but that can be extended. The current WSML RDF Syntax is partly layered on

¹<http://www.swi-prolog.org>



top of RDFS. That means that for reasoning with WSML RDF triples, an axiomatization should be provided for the DL primitives of WSML. Reasoning with RDF triples has the advantage that queries like *all-classes* and *all-properties* can be solved, which is not possible using the translation from DL mentioned above. Other advantages of using this library are that we have the parser for free and that it handles namespaces.

Prolog can handle very well integrity constraints, but have limited datatype support. SWI-Prolog has a comprehensive set of built-in predicates.

In conclusion SWI-Prolog can be used as a reasoner for WSML-Core.

WSML-Flight SWI-Prolog has built-in predicates for default negation and equality, so it covers the minimal requirements for a WSML-Flight reasoner.

WSML-Rule SWI-Prolog supports both function symbols and unsafe rules. However, it does not support meta-modeling, as is allowed in WSML-Rule. SWI-Prolog is seen as a candidate reasoner for WSML-Rule.

WSML-DL SWI-Prolog cannot handle the features of DL that are not in the intersection of DL with Horn Logic Programs, thus, is not a proper reasoner for WSML-DL.

WSML-Full SWI-Prolog is not able to handle features added by WSML-Full compared with WSML-Flight.

Among the advantages of SWI-Prolog are that it is portable to many platforms, including almost all Unix/Linux platforms, Windows (95/98/ME and NT/2000/XP), MacOS X (using X11 for graphics) and many more. Binary distributions for most popular platforms (Windows, Linux and MacOS X) are regularly released. Also, the full source packages can be accessed through the CVS. One of its advantages over other LP reasoning engines is that it offers fast and flexible libraries for parsing SGML (HTML) and XML, RDF, store and query the RDF triple model. As a disadvantage we can list the fact that it does not deal with non-stratified negation (compared with XSB Prolog, for example). Another thing is that it is not a DL reasoner, so naturally, it deals only with the DL subset that intersects Logic Programming.

3.1.2 XSB

XSB² is a Logic Programming system with features such as well-founded semantics, packages for evaluating F-Logic (see the FLORA-2 section) or a HiLog implementation.

The authors of XSB regard it as beyond Prolog because of the availability of SLG resolution [Chen and Warren, 1996] and the introduction of HiLog terms. SLG resolution enables the resolution of recursive queries that SLD resolution cannot deal with, and it also enables the use of well-founded semantics for non-stratified negation.

XSB also offers interfaces to other software systems, such as C, Java, Perl, ODBC, SModels, and Oracle. These interfaces also allow easy integration of new Built-in predicates.

²<http://xsb.sourceforge.org/>



WSML-Core XSB can be used without problems for query answering in WSML-Core. However, integrity constraints (both value and cardinality constraints) are not directly supported but can be externally simulated by queries.

Regarding subsumption reasoning, XSB is not intended for this reasoning task. However, it can be handled in the manner described at the beginning of this section.

XSB allows the use of integers, strings and floating point numbers, and a set of built-in datatype predicates.

Therefore, XSB is a candidate for query answering in WSML-Core, while it is limited regarding subsumption reasoning.

WSML-Flight XSB implements equality and stratified negation. The support for HiLog also enables meta-modelling. Therefore, XSB can be used as a WSML-Flight reasoner.

WSML-Rule The features added in WSML-Rule, namely function symbols and the use of unsafe rules, are supported by XSB. Furthermore, XSB supports the possible extension of the Well-Founded Semantics. Therefore, XSB is seen as a candidate reasoner for WSML-Rule.

WSML-DL XSB cannot handle disjunction on the head and classical negation. In addition, XSB is not intended for subsumption reasoning and, therefore, it is not a suitable WSML-DL reasoner.

WSML-Full It is not clear whether XSB will be able to handle features of WSML-full that are not in WSML-rule.

XSB is available for UNIX and Windows systems, and it is openly distributed under LGPL license. Its major disadvantage is its limited support for subsumption reasoning. In summary, XSB is seen as a suitable reasoner for WSML-Core (although subsumption reasoning is not expected to be efficient), WSML-Flight and WSML-Rule, but not for WSML-DL and WSML-Full.

3.1.3 FLORA-2

*FLORA-2*³ is a reasoner for a dialect of F-Logic [Kifer et al., 1995], including meta-modeling in the style of HiLog [Chen et al., 1993] and transaction logic [Bonner and Kifer, 1998]. *FLORA-2* translates a unified language which includes F-Logic, Transaction Logic and HiLog into tabled Prolog code, relying on XSB Prolog⁴ as the underlying inference engine. In fact, *FLORA-2* can be seen as syntactic sugar on top of XSB. More details about *FLORA-2* can be found in the *FLORA-2* manual⁵.

WSML-Core *FLORA-2* is not intended as a DL reasoner and, therefore, it is not expected to provide efficient subsumption reasoning. Although some attempts have been started to use *FLORA-2* as an OWL reasoner⁶, these seem to be stopped and no substantial results are available.

On the contrary, *FLORA-2* is well-suited for query answering in WSML-Core; it implements a superset of Datalog, and it provides built-in predicates

³<http://flora.sourceforge.net/>

⁴<http://xsb.sourceforge.net/>

⁵<http://flora.sourceforge.net/docs/releasemanual.pdf>

⁶<http://users.ebiquity.org/hchen4/fowl/>



for strings and integers, among others. Regarding integrity constraints, they are not directly supported by *FLORA-2*, as in pure Prolog based systems such as XSB. They can be handled by defining rules with the integrity condition on the body and a special symbol e.g. *false* on the head. For checking the integrity of a knowledge base \mathcal{KB} , it has to be checked that such special symbol is not entailed by \mathcal{KB} i.e. $\mathcal{KB} \not\models \text{false}$. However, notice that the semantics of integrity constraints is defined outside the reasoner. Therefore, *FLORA-2* fulfills the requirements for query answering in WSML-Core if integrity constraints are handled in the way discussed above.

WSML-Flight *FLORA-2* implements equality and stratified negation. Therefore, it can be used as a WSML-Flight reasoner. Meta-modelling is also built into the reasoner.

WSML-Rule The features added in WSML-Rule, namely function symbols and the use of unsafe rules, are supported by *FLORA-2*. Furthermore, *FLORA-2* supports the possible extension of the Well-Founded Semantics. Therefore, *FLORA-2* is seen as a candidate reasoner for WSML-Rule.

WSML-DL *FLORA-2* cannot handle disjunction on the head and classical negation, and as for WSML-Core, it is not well-suited for subsumption reasoning. The F-OWL attempt to provide OWL reasoning based on *FLORA-2* has mainly dealt with OWL-Lite, not with OWL-DL. Therefore, we do not see *FLORA-2* as a good candidate for a WSML-DL reasoner.

WSML-Full *FLORA-2* provides a good support for the extensions added on top of WSML-Core in the direction of Logic Programming. However, as explained before, it does not support some extensions in the direction of Description Logics.

FLORA-2 can handle query answering for a subset of WSML-Full⁷, while doing subsumption reasoning with *FLORA-2* is not expected to be efficient.

In addition, *FLORA-2* it uses minimal model semantics, not first-order semantics.

The main advantages of *FLORA-2* are that it is distributed under LGPL license, it is constantly been improved i.e. it is "alive", and it is available for both Windows and POSIX operating systems.

In addition, it has support for HiLog and transaction logic, and a Java API is already available⁸.

FLORA-2 is fully based on XSB Prolog and relies on the tabling of XSB. We are not aware of any ports planned to other Prolog systems.

The major disadvantage is that *FLORA-2* is not intended as a DL reasoner and, therefore, it is hard to be efficiently used for subsumption reasoning in WSML-Core and WSML-DL.

3.1.4 TRIPLE

TRIPLE⁹ is the name of both: a language for the Semantic Web and the corresponding reasoning system. The language TRIPLE is a layered and modular language for the Semantic Web (especially for the querying and transformation

⁷Our estimation is that this subset will coincide with WSML-Rule.

⁸<http://www.ontotext.com/downloads/>

⁹<http://triple.semanticweb.org/>



of RDF models) which bases on Horn Logic extended by F-Logic features and RDF constructs. Because these extensions are only syntactical, the language can be translated to Prolog and processed by a Prolog system. Correspondingly, the TRIPLE inferencing engine bases on the XSB Prolog system which has been described in section 3.1.2.

WSML-Core TRIPLE can be used for query answering in WSML-Core. For the purpose of subsumption reasoning, TRIPLE is suitable to only a limited extent. Subsumption reasoning can be handled by TRIPLE or XSB, respectively, only by the approach of Grosz et al. which is mentioned at the beginning of this section. Triple allows the usage of the same data type predicates like XSB (cf. section 3.1.2).

Therefore, TRIPLE can be used for query answering, while it is limited regarding subsumption reasoning.

WSML-Flight TRIPLE implements equality and stratified negation. Meta-modelling is also possible with TRIPLE. Therefore it can be used as a WSML Flight reasoner.

WSML-Rule TRIPLE supports function symbols, unsafe rules and well-founded semantics. Therefore WSML-Rule could be implemented on top of TRIPLE.

WSML-DL TRIPLE cannot handle disjunction in the head and is even limited for reasoning with the proper subset WSML-Core. Therefore, we don't see TRIPLE as a candidate for a WSML-DL reasoner.

WSML-Full TRIPLE cannot handle WSML-Full, as it cannot handle its proper subset WSML-DL.

The main advantage of TRIPLE is that it is distributed under BSD License. However, the newest version of TRIPLE is available only for Unix/Linux systems. The limitation to Linux might become inconvenient, when we want to combine several reasoners for the different WSML variants.

3.1.5 OntoBroker

Ontobroker¹⁰ is a reasoner for reasoning over ontologies that are formalized in F-Logic or in Datalog/Prolog. In the Ontobroker data sheet¹¹ it is described as a main memory deductive, object oriented database system.

WSML-Core Ontobroker is no DL reasoning engine. It has to be checked if Ontobroker also implements disjunctive Datalog. But it is very unlikely that it does. Therefore, we can assume that the subsumption checking can not be performed with Ontobroker.

Ontobroker implements a Datalog reasoning engine and can handle both integrity constraints and a huge amount of built-in predicates. Hence, it fulfills all the requirements for performing the query answering task within ontologies formulated in WSML-Core.

¹⁰The Ontobroker system is developed and distributed by the company Ontoprise (cf. http://www.ontoprise.de/home_en).

¹¹http://www.ontoprise.de/documents/datasheet_ontobroker.pdf



WSML-Flight Because Ontobroker implements both the equality symbol and stratified negation, all requirements for reasoning with ontologies in WSML-Flight are fulfilled.

Furthermore, Ontobroker has native support for meta-modeling.

WSML-Rule OntoBroker supports function symbols, as well as Well-Founded Semantics, as well as unsafe rules. Therefore, WSML-Rule could be implemented on top of OntoBroker.

WSML-DL Since Ontobroker is not a DL reasoner, it cannot deal with the most part of WSML-DL.

WSML-Full Ontobroker is not able to handle features added by WSML-Full compared with WSML-Flight. It is even not able to handle WSML-DL.

A major drawback of Ontobroker is that it is being developed within a company and isn't available freely for anyone to use.

3.1.5.1 SILRI

A sidenote is in place on one of Ontobroker's predecessors. SILRI¹² is a less efficient predecessor version of Ontobroker with less features, which however is planned to be made public by Ontoprise. We are currently starting efforts towards the development of the rule engine tailored for WSML, called MINS4 which essentially is a reimplementations of the SILRI rule system. More details on this effort are to be found in Section 4.4.1 later in this document.

3.1.6 DLV

The DLV system¹³ is being developed for several years as joint work of the University of Calabria and Vienna University of Technology.

DLV and the SMOBELS system discussed in the subsequent section distinguish from the above-mentioned Prolog-Based systems in that these systems implement fully declarative logic programming in a purer sense than PROLOG-like systems. These systems compute models rather than answering queries in a top-down fashion.

The DLV system is an efficient engine for computing answer sets (i.e. stable models for logic programs with negation under the extension with classical negation and disjunction [Gelfond and Lifschitz, 1991]. DLV uses as core input language safe Datalog programs (cf. [Ullman, 1989]) with disjunction in rule heads and default negation in rule bodies. Programs are safe if every variable occurring in head literals or default negated body literals also occurs in at least one non-default-negated body literal. Note that in DLV head and body literals may be classically negated. A logic Program Π is *safe* if all of its rules are safe.

Note that the safety restriction is only syntactical but does not really affect the expressive power of the language in any way.

DLV computes stable models, i.e. answer set semantics but not well-founded semantics.

DLV provides some preliminary APIs such as a wrapper for using DLV from Java¹⁴ and ongoing work on an ODBC interface connecting to relational databases.

¹²<http://ontobroker.semanticweb.org/silri/>

¹³[URL:http://www.dlvsystem.com](http://www.dlvsystem.com)

¹⁴<http://160.97.47.246:8080/wrapper/>



Built-In Predicates and Aggregates With respect to the built-ins prescribed in WSML, built-in support in DLV is currently limited as follows.

DLV allows in its current release for a limited set of the built-in predicates “ $A < B$ ”, “ $A \leq B$ ”, “ $A > B$ ”, “ $A \geq B$ ”, “ $A! = B$ ” with the obvious meaning of less-than, less-or-equal, greater-than, greater-or-equal and inequality for strings and numbers which can be used in the positive bodies of DLV rules and constraints.

DLV currently does not support full arithmetics but supports some built-in predicates, which can be used to “emulate” range restricted integer arithmetics: The arithmetic built-ins “ $A = B + C$ ” and “ $A = B * C$ ” which stand for integer addition and multiplication, and the predicate “ $\#int(X)$ ” which enumerates all integers (up to a user-defined limit).

The restrictions on arithmetic built-ins will be lifted in one of the next releases where integration of a rich library of built-ins and also an API for users to add further functional built-ins are planned.

Furthermore, DLV also supports a front-end to interface the DL-Reasoner racer, as described in [Eiter et al., 2004].¹⁵ Support to interface with other reasoners will probably be extended and generalized in future versions. ODBC support is also short to be supported in the near future. Syntactical extension towards higher-order syntax are under currently discussion.

WSML-Core Since WSML-Core (without nominals) falls in the Datalog subset of Horn-Logic, DLV can serve for several reasoning tasks. As the other Logic programming based engines, DLV is more suitable for query answering than for subsumption reasoning. However, to the expressiveness of Logic Programming under the stable model semantics, restricted forms of subsumption, namely conjunctive query containment which is well known to be NP-complete [Chandra and Merlin, 1977], could be checked, after Logspace transformations. As for datatype support the current basic support for built-ins and arithmetics seems not yet sufficient. APIs for extending the set of available built-ins is currently not provided.

Integrity constraints are naturally supported inside the stable model semantics. This distinguishes engines for logic programs under the stable model from other semantics for Logic Programming where Integrity constraints can only be emulated by additional queries.

Due to the safety restriction, the *true*-keyword in WSML would need to be emulated by a unary relation *HU* which contains the whole Herbrand Universe, for rules only having true in the body.

WSML-Flight Stratified Negation, more precisely stratified default negation is naturally supported in the language of DLV. Moreover, DLV allows the use of classical negation, which can be viewed as syntactic sugar in answer set semantics compared with stable model semantics. DLV supports inequality in the body. However, meta-modeling is not directly supported in DLV.

Thus, DLV fulfills all requirements for reasoning with WSML-Flight.

WSML-Rule DLV does not support function symbols or unsafe rules. Furthermore, DLV supports the stable model semantics which defers from the well-founded semantics which WSML-Rule is based upon for non-stratified programs. Thus, the restrictions we would have to apply to WSML-Rule to make use of the DLV reasoner make the language equivalent to WSML-Flight. Therefore, we do not consider DLV as a reasoner for WSML-Rule as is.

¹⁵<http://www.kr.tuwien.ac.at/staff/roman/semweb1p/>



WSML-DL DLV is on the one hand not tailored for reasoning about description logics. On the other hand the translations of DL to disjunctive Datalog outlined in [Hustadt et al., 2004b] can be used instantly. Note that the full expressiveness of the language supported by DLV is not needed in this translation, since it only uses positive disjunctive Datalog without negation in the body.

WSML-Full DLV is not able to handle features added by WSML-Full compared with WSML-Flight. It is even not able to handle WSML-DL (except via interfacing to an external DL reasoner as pointed out above).

DLV is not being developed within an open-source license, and therefore not completely open for extensions. However, since it is developed in an academic environment, collaborations for further academic development are possible. Future planned extensions of the system seem very promising for adoption towards reasoning in the context of WSML.

3.1.7 SMODELS and GNT

SMODELS [Niemelä, 1999; Simons et al., 2002]¹⁶ allows for the computation of stable models and well-founded models for normal logic programs, that is for Datalog programs with non-disjunctive heads. However, there is an extended prototype version for the evaluation of disjunctive logic programs as well, called GNT [Janhunen et al., 2000]¹⁷.

Syntactically, SMODELS imposes an even stronger restriction than rule safety in DLV by demanding that any variable in a rule is bounded to a so-called domain predicate in the rule body which is, intuitively, a predicate which is defined only positively (cf. [Niemelä, 1999] for details). Again, this restriction does not affect the expressive power of the language itself, but in some cases the weaker safety restriction of DLV allows for more concise problem encodings.

Arithmetics and Built-in Predicates Similar to DLV, SMODELS allows for a restricted form of integer arithmetics and lexicographic comparison predicates.

WSML As for the implementability for the various WSML variants, similar considerations apply as for DLV mentioned above. Differences show up in the fact that besides computation of stable models SMODELS also supports computation of well-founded models.

SMODELS is developed under the GNU Public Licence and provides more API support than DLV, also from third parties. For instance, also XSB mentioned above provides a direct interface to smodels called XASP enabling the use of stable model semantics from XSB.

3.1.8 KAON2

KOAN2¹⁸ provides a hybrid reasoner which is able to reason with a large subset of OWL DL, corresponding to the Description Logic *SHIQ(D)* [Horrocks et al., 2000], and Disjunctive Datalog with stratified negation, along with basic built-in predicates to deal with integers and strings. The theoretical work underlying KAON2 can be found in [Hustadt et al., 2004a].

¹⁶<http://www.tcs.hut.fi/Software/smodels/>

¹⁷<http://www.tcs.hut.fi/Software/gnt/>

¹⁸<http://sourceforge.net/projects/kaon2/>



WSML-Core KAON2 can perform subsumption reasoning with the subset of WSML-Core which falls inside the *SHIQ* Description Logic.

Since KAON2 implements a Datalog engine and is able to handle integrity constraints, as well as rudimentary built-in predicates, it fulfills all the requirements for performing the query answering task with WSML-Core.

WSML-Flight Because KAON2 implements the equality symbol and stratified negation, all requirements for reasoning with WSML-Flight are fulfilled. KAON2 has been extended with additional meta-level rules in order to be able to handle meta-modeling. Consequently, KAON2 is a candidate reasoner for WSML-Flight.

WSML-Rule KAON2 does not support function symbols or unsafe rules. Furthermore, it does not support the Well-Founded Semantics. Thus, the restrictions we would have to apply to WSML-Rule to make use of the DLV reasoner make the language equivalent to WSML-Flight. Therefore, we do not consider KAON2 as a reasoner for WSML-Rule.

WSML-DL Since KAON2 is a DL reasoner for the *SHIQ(D)* DL, it can deal all of WSML-DL.

WSML-Full KAON2 is not able to handle features added by WSML-Full compared with WSML-DL and WSML-Flight. However, it does allow for an interesting combination of Description Logics and rules, since Description Logic ontologies are reduced to disjunctive logic programs by KAON2 in order to allow for efficient query answering.

A major drawback of KAON2 is that it has not been fully implemented yet. The implementation of KAON2 is ongoing in the context of the DIP project¹⁹. A first implementation is promised for December 2004. This implementation would include all features except for built-in datatype predicates. However, the software is far from stable and it remains to be seen what the quality is of the software and what the efficiency of the reasoner remains to be proven in practice.

3.1.9 Summary

The surveyed Logic Programming implementations all agree on Datalog with stratified negation. This makes implementation of WSML-Flight on any of the surveyed reasoners possible.

The reasoners SWI-Prolog, XSB, FLORA-2, and OntoBroker all support function symbols and are thus suitable for implementation of WSML-Rule. The envisioned extension of unstratified negation under the Well-Founded Semantics is supported by XSB, FLORA-2 and OntoBroker.

Frame-based modeling is supported by FLORA-2 and OntoBroker. Native support for frame-based modeling make the implementation of a WSML-Rule reasoner easier. Thus, FLORA-2 and OntoBroker are the preferred candidates for the implementation of WSML-Rule.

¹⁹<http://dip.semanticweb.org/>



3.2 First-order theorem provers

First-order Logics are quite expressive logics which can be applied for reasoning in a wide range of domains. They provide a flexible general-purpose framework for representing domain knowledge and inferring new knowledge from a given knowledge base. Their theoretical properties are well-understood and the construction of deduction systems for First-order Logics has been a subject of investigation for several decades now [Davis, 2001]. Of course, the flexibility and expressiveness of this family of logics comes at a cost: Reasoning with these languages in general is undecidable²⁰ That means, when using these logics one loses in general a certain nice guarantee, namely to get an answer for *any* concrete input within a finite period of time when performing reasoning.

For this family of logics numerous calculi for checking logical entailment have been proposed and implemented. We have selected two of them that are discussed in the following sections. The reason underlying our choice is that both systems seem to be able to deal with larger knowledge bases during reasoning which is a key property in our applications as well as one of the major problems for reasoners for these logics. Both systems at least have been applied in applications which are similar to the ones that we aim at.

3.2.1 Vampire

The Vampire system has been developed by Alexandre Riazanov and Andrei Voronkov at the University of Manchester, England. Several versions of the system have been tested and evaluated in the annual contest for automated theorem provers called CASC²¹ and during the last years, the system won in the categories for First-order Logics with equality (FOF and MIX). It can be considered as one of the most powerful systems for fully-automated theorem proving in First-order Logics. In the following we refer to the most recent version of the system, namely Vampire 7.0 which participated in CASC in 2004.

One of the most interesting and distinguishing characteristics of this system is that the system designer spent significant effort in the development and design of powerful techniques for *implementing* theorem proving systems i.e. the overall technical infrastructure needed during proof search, such as indexing techniques for terms and formulae, runtime-compilation of code, careful memory-management, resource-oriented (or resource-aware) algorithms, etc. In this respect, there is currently no comparable system available.

Technical Overview. Vampire²² [Riazanov and Voronkov, 2002] is an automatic theorem prover for first-order classical logic. Its kernel implements the calculi of ordered binary resolution and superposition for handling equality. The splitting rule and negative equality splitting are simulated by the introduction of new predicate definitions and dynamic folding of such definitions. A number of standard redundancy criteria and simplification techniques are used for pruning the search space: subsumption, tautology deletion (optionally modulo commutativity), subsumption resolution, rewriting by ordered unit equalities, basicness

²⁰Depending on the concrete reasoning task the problem can be recursively enumerable or even not recursively enumerable. For instance, checking logical entailment of a formula from a knowledge base is semi-decidable since there are complete calculi that formalize logical entailment for First-order Logic. On the other hand, the set of satisfiable First-order Logic formulae is not recursively enumerable and thus checking satisfiability of arbitrary formulae in these logics is in general out of scope for algorithmic treatment.

²¹<http://www.cs.miami.edu/~tptp/CASC/>

²²<http://www.cs.man.ac.uk/~riazanov/Vampire>



restrictions and irreducibility of substitution terms. The reduction orderings used are the standard Knuth-Bendix ordering and a special non-recursive version of the Knuth-Bendix ordering. A number of efficient indexing techniques is used to implement all major operations on sets of terms and clauses. Runtime algorithm specialization is used to accelerate some costly operations, e.g., checks of ordering constraints. Although the kernel of the system works only with clausal normal forms, the preprocessor component accepts a problem in the full first-order logic syntax, clausifies it and performs a number of useful transformations before passing the result to the kernel. When a theorem is proven, the system produces a verifiable proof, which validates both the clausification phase and the refutation of the CNF. The current release features a built-in proof checker for the clausifying phase, which will be extended to check complete proofs. Vampire 7.0 is implemented in C++.

Strategies. The Vampire kernel provides a fairly large number of features for strategy selection. The most important ones are:

- Choice of the main saturation procedure : (i) OTTER loop, with or without the Limited Resource Strategy, (ii) DISCOUNT loop. A variety of optional simplifications.
- Parameterized reduction orderings.
- A number of built-in literal selection functions and different modes of comparing literals.
- Age-weight ratio that specifies how strongly lighter clauses are preferred for inference selection.
- Set-of-support strategy.

The automatic mode of Vampire 7.0 is derived from extensive experimental data obtained on problems from TPTP v2.6.0 problem library²³. Input problems are classified taking into account simple syntactic properties, such as being Horn or non-Horn, presence of equality, etc. Additionally, the presence of some important kinds of axioms, such as set theory axioms, associativity and commutativity is taken into account. Every class of problems is assigned a fixed schedule consisting of a number of kernel strategies called one by one with different time limits.

WSML-Core Since WSML-Core without datatype predicates can be considered as a subset of the *SHOIN* Description Logic, which can be translated into First-order Logic [Hustadt et al., 2004a]. Thus, Vampire (as any First-order Logic theorem prover) can be used for reasoning with WSML-Core without datatype predicates. In fact, Vampire has already been evaluated in this respect and compared with the Description Logic inference system FACT²⁴ [Tsarkov and Horrocks, 2003a] and its re-implementation FaCT++ [Tsarkov et al., 2004] (see also Section 3.3.1). The results of the evaluation show that Vampire is in general slower²⁵ than the FACT(++) system, which is not surprising since FaCT++ is specifically tailored towards a significantly simpler logic. On the other hand, Vampire actually proved to be able to solve many questions with acceptable response time.

²³<http://www.cs.miami.edu/~tptp/>

²⁴<http://www.cs.man.ac.uk/~horrocks/FaCT/>

²⁵Sometimes significantly slower!



A standard way for dealing with concrete datatypes in theorem proving is to use an explicit axiomatization of datatype: One gives a set of formulae that constrain the interpretation of the symbols used to denote datatype operators to their intended semantics. Nonetheless, this way of dealing with datatypes is not a very efficient one and thus should only be considered as a work-around. Since some examples for Vampire Problem files indicate, the system seems to support some built-in datatypes starting from Version 7.0, though we do not know yet, what datatypes are supported and to what extent they are supported. Currently, there is no documentation available. Nonetheless, it is not clear whether datatype predicates or oracles will be supported in the way that is needed for WSML-Core.

Although Vampire can be used to check logical entailment of arbitrary formulae and thus can basically be used for the required reasoning tasks for WSML-Core, there are no special techniques for a query answering task in the style of deductive databases. Thus, it is unlikely that Vampire can perform comparably well (wrt. deductive databases) on this task. In the worst case, one has to apply a naive enumeration and check algorithm for exhaustive query answering which obviously is not a good solution.

The symbols *true* and *false* are supported by WSML-Core as well. *What about Vampire?*

WSML-Flight Vampire supports equality and the *standard* first-order negation. No other form of negation is implemented in the Vampire system, since it deals with classical Predicate Logic with equality. It does not provide specific (built-in) support for Meta-Modelling since this is not part of First-order Logic, though it can deal with Meta-Modelling as long as it can be encoded in First-order Logic.

WSML-Rule Function symbols are supported by Vampire though it does not implement variants of the negation symbol other than the standard First-order negation symbol. In particular, Vampire (as every First-order logic theorem prover) sticks to the open-world semantics. Thus, it can not be used for reasoning with descriptions in WSML-Rule.

WSML-DL Here, the same arguments as for WSML-Core apply: Vampire can basically be used for both envisioned reasoning tasks, nonetheless for query answering we expect the system to not perform well in comparison to special-purpose systems like deductive databases.

WSML-Full Since Vampire is a deduction system for classical Predicate Logic, it does not provide any support for non-monotonic reasoning. Thus, it can not be applied for reasoning with WSML-Full in general. However, it can be applied for the subset of WSML-Full which does not use non-monotonic features. In this subset, Vampire is likely to outperform most other systems when checking logical entailment. The same drawbacks as before apply for query answering, although for languages of this expressiveness we are not aware of any system which is specifically tailored towards the query answering task.

Further Remarks on Vampire. Vampire is under continuous development and evaluation. There are no indications that this will change in the near future. At present, there is no Handbook or detailed documentation for users available.

Vampire supports the TPTP format²⁶ as a standard-like input format for

²⁶<http://www.cs.miami.edu/~tptp/TPTP/TR/TPTPTR.shtml>



problem specifications. When vampire is started on such an input file, it tries to construct a proof for the conjecture included in the problem specification. The output of the prover indicates whether a proof has been found (actually the proof can be output as well), no proof could be constructed either within a given time limit or if such a proof is not possible at all²⁷.

It seems that Vampire currently does not really support some sort of knowledge base mode where one feeds a knowledge base into the prover and just invokes the proof search for various queries. Instead, one has to feed everything which is relevant for proving into the prover with every query. Clearly, this might be a drawback in many application in the Semantic Web since the domain knowledge is needed for all reasoning tasks whereby the domain knowledge is comparably stable wrt. the given conjectures to be checked (or queries to be answered). From private communication with the authors we know that such a feature could be supported by Vampire in the future if there are applications that need that feature.

3.2.2 SNARK

SNARK, SRI's New Automated Reasoning Kit, is a theorem prover intended for applications in artificial intelligence and software engineering. SNARK is geared toward dealing with large sets of assertions and thus it is of special interest for us; it can be specialized with strategic controls that tune its performance; and it has facilities for integrating special-purpose reasoning procedures with general-purpose inference.

SNARK has been used as the reasoning component of SRI's High Performance Knowledge Base (HPKB) system²⁸ [Pease et al., 2000], which deduces answers to questions based on large repositories of information. It constitutes the deductive core of the NASA Amphion system²⁹, which composes software from components to meet users' specifications, e. g., to perform computations in planetary astronomy. SNARK has also been connected to Kestrel's SPECWARE environment³⁰ for software development.

Technical Overview. SNARK is a resolution-and-paramodulation theorem prover for first-order logic with equality. SNARK has provisions for both clausal and nonclausal reasoning, and it has an optional sort mechanism. It incorporates associative/commutative unification and a built-in decision procedure for reasoning about temporal points and intervals. Furthermore, it allows to use special-purpose external procedures (so-called Procedural Attachments) during the inference process and supports question answering applications to some extent. As input format for the prover, SNARK supports KIF+C (KIF plus Classes), the facility that enables SNARK to understand assertions and queries phrased in KIF [Genesereth, 1991; Genesereth and Fikes, 1992], the Knowledge Interchange Format, with some extensions from OKBC (Open Knowledge-Base Connectivity) [Chaudhri et al., 1998], an object-oriented framework for representing knowledge. This dialect was used in the HPKB project and other knowledge representation efforts. It has no special facilities for proof by mathematical induction. It has some capabilities for abductive reasoning, which have been used in natural-language applications. SNARK is implemented in an easily portable subset of COMMON LISP.

²⁷In this case, one is really lucky since this the problematic case for FOL theorem provers, since non-termination of the proof search can happen here!

²⁸<http://www.ai.sri.com/project/HPKB>

²⁹<http://ase.arc.nasa.gov/docs/amphion.html>

³⁰<http://www.kestrel.edu/HTML/prototypes/specware.html>



SNARK is a refutation system; in other words, rather than trying to show directly that some assertions imply a desired conclusion, it attempts to show that the assertions and the negation of the conclusion imply a contradiction. It is an agenda-based system; that is, in seeking a refutation, it will put the assertions and the negation of the conclusion on an agenda. An agenda is a list of formulas. When a formula reaches the top of the agenda, SNARK will perform selected inferences involving that formula and the previously processed formulas. The consequences of those inferences are added to the agenda. This process continues until the propositional symbol `false` is derived; this means that a contradiction has been deduced and the refutation is complete. The user has considerable control over the position at which a newly derived formula is placed on the agenda; this is one way in which a knowledgeable user can tailor SNARK's search strategy to a particular application.

WSML-Core SNARK can be used for reasoning with WSML-Core without datatype predicates. Via procedural attachments it seems to be possible to add special-purpose decision procedures for concrete datatypes and datatype predicates. Thus, extending SNARK to support full WSML-Core should be possible. SNARK was intended to support query answering applications as well and thus seems to support the derivation from multiple concrete answers to queries of a knowledge base. At present, we do not have further details on this feature and thus it is unclear whether SNARK is significantly better than other FOL provers in this respect. However, at least there is a guarantee for minimal support for the query answering reasoning task. An interesting feature of SNARK is that it provides support for sorts. Since it uses an extension of KIF by classes as a possible input format it as well provides some rudimentary support for ontology reasoning. Whether reasoning in the presence of sorts in problem specifications is more efficient than with provers that do not support sorts still has to be evaluated.

WSML-Flight SNARK supports equality but no non-monotonic form of negation. As in the case of Vampire it does not provide specific (built-in) support for Meta-Modelling since this is not part of First-order Logic, though it can deal with Meta-Modelling as long as it can be encoded in First-order Logic.

WSML-Rule The system supports function symbols though it does not implement non-monotonic variants of the negation symbol. Thus, it can not be used for reasoning with descriptions in WSML-Rule.

WSML-DL Here the same arguments as for WSML-Core apply: SNARK can basically be used for reasoning with WSML-DL descriptions. Actually, there is a First-order axiomatization of OWL (including OWL-Full) in the input format of SNARK available. The authors evaluated the axiomatization as well as reasoning with OWL using the SNARK deduction system [Waldinger, 2004].

WSML-Full Since SNARK is a deduction system for classical Predicate Logic, it does not provide any support for non-monotonic reasoning. Thus, it can not be applied for reasoning with WSML-Full in general. However, it can be applied for the subset of WSML-Full which does not use non-monotonic features.



3.2.3 Summary

First-order logic theorem provers are flexible and very general tools. Often they are too general for specific application and thus perhaps unnecessarily heavy-weight. One general difficulty when applying deduction systems for First-order Logic in some application is that most often one has the opportunity to set a lot of parameters to affect the proof search and thus the efficiency of theorem proving in a specific application at hand. This freedom at the same time can be a burden if the system does not provide a strong automatic mode where the parameters are determined automatically by syntactically analyzing the input data. Customizing a automated theorem prover for First-order Predicate Logic is a non-trivial task which is needed in order to get reasonable performance within specific applications. The configuration process requires a deep understanding of how a system works and what formulae one typically has to deal with in the specific application.

3.3 Description Logic reasoners

We describe three state-of-the-art Description Logic reasoners, namely: FaCT++, RACER, and Pellet. Note that KAON2, earlier described in Section 3.1.8, is also a Description Logic reasoner. In fact, it is a Hybrid reasoner, able to reason with both Description Logic ontologies and Disjunctive Datalog programs.

3.3.1 FaCT++

FaCT++³¹ is a DL reasoner for the $SHIQ(\mathcal{D})$ Description Logic. It is a re-implementation of the DL reasoner FaCT [Horrocks, 1998] which also adds some additional features. More details can be found in [Tsarkov and Horrocks, 2003b].

WSML-Core FaCT++ can handle $SHIQ(\mathcal{D})$. It can handle string and integer datatypes, and it provides symbols for the top and bottom concepts, which are equivalent to the *true* and *false* symbols. FaCT++ can provide subsumption reasoning for all subset of WSML-Core.

FaCT++ is not a good candidate for query answering in WSML-Core, as A-Box reasoning in FaCT++ is not efficient and it provides limited support for individuals.

WSML-Flight FaCT++ does not support neither default negation nor meta-modelling. Although it does provide an equality predicate for concepts and individuals, it does not fulfill all the requirements for a WSML-Flight reasoner.

WSML-Rule FaCT++ does not support function symbols and, as said before, it does not support default negation. Therefore, it is not suitable as a WSML-Rule reasoner.

WSML-DL As for WSML-Core, FaCT++ can handle $SHIQ(\mathcal{D})$, and therefore it can complete handle WSML DL.

³¹<http://owl.man.ac.uk/factplusplus/>



WSML-Full As explained before, FaCT++ does not completely support WSML-Core and WSML-DL, and it does not support WSML-Flight and WSML-Rule. Therefore, it is not a good candidate as a reasoner for WSML-Full.

The main advantages of FaCT++ is that it is expected to include support for OWL DL, it is available for Windows, Linux and MAC operating systems, and provides efficient subsumption reasoning for WSML-Core and WSML-DL.

FaCT++ is distributed under GPL license. Therefore, it is open source, which is an advantage. However, every piece of software depending on it must also be open source, which can be an important disadvantage if it is intended to be used for commercial applications.

3.3.2 RACER

RACER³² is a DL reasoner for the *SHIQ* Description Logic. It also provides limited support for concrete domains.

WSML-Core As said before, RACER can handle *SHIQ*. RACER can handle datatypes, namely natural, integer, real and complex numbers, and strings, and it provides symbols for the top and bottom concepts, which are equivalent to the *true* and *false* symbols.

Therefore, RACER can provide subsumption reasoning for WSML-Core.

Regarding query answering support, RACER supports restrictions for abstract domains and constraints for concrete domains i.e. constraints for abstract domains are not handled by RACER. It provides built-in predicates for the datatypes afore-mentioned. RACER provides support for individuals and adheres to the Unique Name Assumption (UNA). Although ABox query processing, which can be seen as query answering, has been improved in the last version of RACER, its efficiency when compared to Logic Programming reasoners is expected to be worse. Therefore, RACER can be considered as a reasoner for query answering in WSML-Core, but it is not expected to improve the results of LP reasoners.

WSML-Flight RACER supports neither default negation nor meta-modelling. Although it does provide an equality predicate for concepts and individuals, it does not fulfill all the requirements for a WSML-Flight reasoner.

WSML-Rule RACER supports neither function symbols nor default negation. Therefore, it is not suitable as a WSML-Rule reasoner.

WSML-DL As said before, RACER can handle *SHIQ*. Therefore, RACER can be used as a reasoner for WSML-DL with sound and complete reasoning.

WSML-Full RACER does not support non-monotonic features that are expected to be added in WSML-Full. In addition, and as seen before, it cannot support WSML-Flight and WSML-Rule.

The main advantage of RACER is that it is seen as a state-of-the-art DL reasoner and, therefore, it is currently expected to provide the most efficient support for WSML-DL.

The major disadvantage is that RACER is only provided free of charge for universities and research labs, and that the source code is not publicly available.

³²<http://www.sts.tu-harburg.de/~r.f.moeller/racer/>



3.3.3 Pellet

Pellet³³ is a sound and complete DL reasoner for the $SHIN(\mathcal{D})$ and $SHON(\mathcal{D})$ Description Logics, and sound and incomplete for $SHOIN(\mathcal{D})$.

WSML-Core Pellet can handle XML Schema basic datatypes. In addition, it can test the satisfiability of conjunctions of XML Schema constructed datatypes. It also provides symbols for the top and bottom concepts, which are equivalent to the *true* and *false* symbols.

Therefore, RACER can provide subsumption reasoning for WSML-Core.

Regarding query answering support, constraints for abstract domains are not handled by Pellet. It provides support for individuals but it does not adhere to the Unique Name Assumption (UNA). Although optimization algorithms are proposed, the efficiency of query answering in Pellet is expected to be worse than the efficiency of LP reasoners.

WSML-Flight Pellet supports neither default negation nor meta-modelling. Although it does provide an equality predicate for concepts and individuals, it does not fulfill all the requirements for a WSML-Flight reasoner.

WSML-Rule Pellet supports neither function symbols nor default negation. Therefore, it is not suitable as a WSML-Rule reasoner.

WSML-DL Pellet can be used as a sound and complete reasoner for WSML-DL.

WSML-Full Pellet does not support non-monotonic features that are expected to be added in WSML-Full. As seen before, it cannot support WSML-Flight and WSML-Rule.

When compared to the other DL reasoners, and based on the examples tested in the context of [Keller et al., 2005], Pellet seems to provide a better support for nominals than RACER. It also shows a good coverage of the OWL test cases³⁴.

However, in terms of optimizations, it is not clear whether Pellet is more optimized than FaCT++ and RACER.

3.3.4 Summary

The surveyed Description Logic reasoners all agree on the $SHIQ(\mathbf{D})$ Description Logic, which underlies WSML-DL. Thus, all three reasoners could be used for WSML-DL. At this point in time, we cannot make a distinction between the mentioned reasoners.

3.4 Other Nonmonotonic Reasoning Techniques

Apart from non-monotonic semantics for Logic programs like the stable and well-founded semantics there are several other logic formalisms providing non-monotonic reasoning facilities.

³³<http://www.mindswap.org/2003/pellet/index.shtml>

³⁴<http://www.w3.org/TR/owl-test/>



3.4.1 Default logic

Default logic is one of the most well-known nonmonotonic logics introduced by Reiter[Reiter, 1980]. Defaults are used to derive new information under the assumption of "normality" or "typicality" of a situation. Knowledge is represented in the form of a default theory which is a pair (D, W) where W is a set of closed well-formed formulae(wff) in L , and D is a set of defaults of the form:

$$\frac{\alpha : \beta_1, \dots, \beta_n}{\gamma}$$

where α, β_i, γ are formulae in L .

An abbreviated syntax for such a default is: $\alpha : \beta_1, \dots, \beta_n / \gamma$. The meaning of a default δ is that γ can be believed when α is believed and for all $i = \overline{1, n}$, there is no reason to believe $\neg\beta_i$. The formula α is called the *prerequisite* (notation: $pre(\delta)$), $\beta_i, i = \overline{1, n}$, are the *justifications* (notation: $just(\delta)$) and γ is the *consequent* of the default (notation: $cons(\delta)$).

The acceptable belief sets a reasoner may derive based on a default theory are called *extensions* of that theory. They formally describe a semantics for default theories. Two definitions for the notion of extension were originally given by Reiter :

Definition 3.1. Let (D, W) be a default theory, S a set of formulae. Let $\Gamma(S)$ be the smallest set such that:

1. $W \subseteq \Gamma(S)$,
2. $Cn(\Gamma(S)) = \Gamma(S)$,
3. if $\alpha : \beta_1, \dots, \beta_n / \gamma \in D, \gamma \in \Gamma(S), \neg\beta_i \notin \Gamma(S) (1 \leq i \leq n)$, then $\gamma \in \Gamma(S)$.

E is an extension of (D, W) iff $E = \Gamma(E)$, that is if E is a fix point of Γ .

Definition 3.2. Let $E \subseteq L$ be a set of closed wffs, and let (D, W) be a closed default theory. Define

1. $E_0 = W$, and
2. For $i \geq 0$ $E_{i+1} = E_i^* \cup \{\gamma \mid \alpha : \beta_1, \dots, \beta_n / \gamma \in D \text{ where } \alpha \in E_i \text{ and } \neg\beta_1, \dots, \neg\beta_n \notin E_i\}$.

E is an extension for Δ iff for some ordering $E = \bigcup_{i=0}^{\infty} E_i$.

The most frequent reasoning tasks that are associated with the default logic are checking whether a default theory has an extension and whether a set of clauses T is contained in some or in all the extensions of the theory. A formula γ is called a sceptical (credulous) consequence of a default theory (D, W) iff γ is contained in all extensions (at least one extension) of the theory (D, W) .

As a remark about the definitions above, the first one is based on a fix-point equation, while the term to be defined (the extension) appears in the body of the second definition. Due to these features, the definitions are not constructive.

Default logic is related with the class of logical programs with negation as failure, a one-to-one correspondence being established between stable models of logic programs and the extensions of their translation as a default theory. This link is extended between different subsets of default theories and extended logic programs, respectively disjunctive logic programs. A detailed view of these connections is given in section 3.4.1.1. This is a very important feature of default logic that is exploited by many systems that implement it. On the other hand,



we can exploit this feature for reasoning with the WSML variants that intersect with (Extended) Logic Programming, i.e WSML Core, WSML Flight and maybe WSML Rule.

For overcoming the unconstructive definitions of extensions, in [Antoniou, 1997] is given an alternative definition that makes straightforward the design of an algorithm for computing extensions. A partial implementation of this algorithm is given in Prolog. This approach is discussed in the section 3.4.1.3.

Many approaches for computing extensions use the following property: every extension of a default theory (D, W) can be regarded as the logical closure of W and the consequents of some defaults from D , which are the *generating defaults* for that extension.

Theorem 3.1. *Each extension of a default theory (D, W) has a set of generating defaults $GD \subseteq D$, i.e. if E is an extension of (D, W) , then $E = Cn(W \cup \{cons(d) | d \in GD\})$ where $GD = \{d \in D | pre(d) \in E\}$, and for all $\psi \in just(d), \psi \in E$.*

The converse of the above property is also true for theories that have only prerequisite-free defaults. This property is used by the system for automatic default reasoning DeReS that offers the possibility to reason with both normal logic programs with negation as failure and with default theories. The system is presented in the section 3.4.1.2.

3.4.1.1 Default Logic and Logic Programming

One interesting property about default logic is that it provides a semantics for normal logic programs (without disjunction) with negation as failure. This is also valid the other way around, one of the semantics attached to the default logic (to the subset of default logic that can be translated to logic programming) being the stable model semantics [Gelfond and Lifschitz, 1988]. In [Cholewiński et al., 1999] is described an encoding of logic programs as default theories, a one-to-one correspondence between stable models of a program and extensions of its default interpretation being established.

The Default Logic interpretation of a ground program clause Cl

$$A \leftarrow B_1, \dots, B_n, notC_1, \dots, notC_m$$

is given by the default:

$$df(Cl) = \frac{B_1 \wedge \dots \wedge B_n : \neg C_1, \dots, \neg C_m}{A}$$

The default logic interpretation of the logic program P is the default theory (D, W) with $W = \emptyset$ and $D = \{df(Cl) | Cl \in ground(P)\}$. It is shown that:

Theorem 3.2. *A subset M of the Herbrand base is a stable model of the logic program P iff $Cn(M)$ is an extension of $df(P)$.*

Based on this property, systems that implement default logic can be used for reasoning with logic programs with negation as failure (since usually all these systems implement the subset of default theories that have correspondents in normal logic programs). Thus, such systems are suitable as reasoners for WSML-Core and WSML-Flight. If these systems do not accept logic programs as an entry, a preprocessing step is needed for translating from logic programs to default theories.

Query answering with such a system amounts to checking whether the corresponding default theory has a single extension and whether the ground formula



that has to be proved is included in that unique extension. Subsumption reasoning for WSML-Core can be reduced to query answering as discussed in the introduction to the section 3.1. So this also is feasible with any default logic reasoner.

Since the semantics attached to normal logic programs in this case is the stable model semantics, non-stratified logic programs can be dealt with default logic reasoners. This is one of the requirements of WSML-Rule. Regarding the possibility to deal with other potential features of WSML-Rule like classical negation or function symbols, this depends on the expressiveness of allowed default theories. This is also the case for WSML-DL and WSML-Full. When the default logic reasoners allow FOL formulas as facts and as prerequisites, justifications and consequences of default rules, both of these languages are covered.

Also, systems that implement the stable model semantics can be used for reasoning with the subset of default logic that intersects with normal logic programs with negation as failure. Let's note that this subset is composed from default theories that allow only atomic facts and default rules with atomic prerequisites and conclusions and negative literals as justifications. Thus DLV and SMOBELS, already described in previous sections, can be employed for reasoning with these kinds of defaults.

The property is generalized in [Antoniou, 1997] for *extended logic programs*, i.e. programs that admit also the classical negation. Thus the default interpretation of an extended clause

$$Cl = L \leftarrow L_1, \dots, L_n, \text{not}L^1, \dots, \text{not}L^m,$$

is the default:

$$df(Cl) = \frac{L_1 \wedge \dots \wedge L_n : \sim L^1, \dots, \sim L^m}{L}.$$

If L is an atomic formula A , $\sim L$ is the formula $\neg A$, whereas if L has the form $\neg A$, $\sim L$ is the formula A . Similar with the case of logic programs, the interpretation of an extended logic program is given by the default theory $(\emptyset, \{df(Cl) | Cl \in \text{ground}(P)\})$ and:

Theorem 3.3. *A subset M of the set of ground literals in the fixed(chosen) language is a stable answer set for an extended logic program P iff $Cn(M)$ is an extension of $df(P)$.*

Let's note that, based on this property, the subset of default theories that have attached a semantics in terms of stable answer sets is composed from default theories that allow literals as facts and default rules with literals as prerequisites, justifications and conclusions. Systems that implement the stable model semantics for extended logical programs, like DLV can be used for reasoning with such defaults.

A larger class of logical programs, disjunctive logic programs with stable model semantics [Przymusiński, 1991], turn out to have a translation into default theories. In [Sakama and Inoue, 1993], the following translation is proposed for a disjunctive program P into a default theory D_P :

- to each rule $A_1 \vee \dots \vee A_k \leftarrow B_1, \dots, B_m, \text{not}C_1, \dots, \text{not}C_n$ in P the following default corresponds in D_P :

$$\frac{: \neg C_1, \dots, \neg C_n}{B_1 \wedge \dots \wedge B_m \rightarrow A_1 \vee \dots \vee A_k}$$



- to each atom A appearing in P , the following default corresponds in D_P :

$$\frac{: \neg A}{\neg A}$$

The atomic part of the extensions of D_P provide the stable models of P , i.e.:

Theorem 3.4. *A subset M of the set of ground atoms in the fixed(chosen) language is a stable model for a disjunctive logic program P iff it is the set of atoms in an extension of D_P .*

Thus, default logic systems that deal with default theories like those that are translations of disjunctive logic programs can be used for reasoning with such logic programs. The converse is not valid because the stable models of a disjunctive logic program are only subsets of the extensions of the corresponding default theory.

3.4.1.2 DeReS

DeReS[Cholewiński et al., 1999; Cholewiński et al., 1996] is a software package developed at the University of Kentucky implementing the Answer Set Programming paradigm[Lifschitz, 1999] and running under all major versions of Unix, including Linux. It enables automated reasoning with Default Logic and with Logic Programming with the stable model semantics.

DeReS computes extensions for finite propositional default theories. Given a default theory, DeReS can determine whether one or more extensions exist and can compute one of the extensions or all of the extensions. There are no syntactic restrictions on input default theories and formulas.

The strategy of DeReS, called *generate and check*, is the following: it search through the space of defaults sets for the sets of *generating defaults* of an extension. For this, it generates and searches a full binary tree whose nodes are labeled by subsets of D . When the sets of defaults represented by a node in the search space is found to be generating an extension, DeReS prunes all descendants of this node in the search tree.

Due to the exhaustive character of the search, the algorithm described above always provides correct results, but it is not very efficient, especially when the theory contains a large number of default rules. A technique called relaxed stratification[Cholewiński, 1995a; Cholewiński, 1995b] of a default theory was devised for applying the principle *divide-and-conquer* when computing extensions. This type of stratification is a relaxation of the classical stratification of logical programs and it is quite a unique feature of DeReS compared with other systems for nonmonotonic reasoning. It can be applied for default theories that do not have justification-free defaults and in which formulas from W do not have common propositional variables with the consequences of the defaults. An original default theory is partitioned in small sub-theories called strata. The extensions of this original theory are computed by assembling the computed extensions of its strata.

As the authors of DeReS acknowledge, if relaxed stratification does not yield a partition of an input theory into small strata, the efficiency of the reasoner can be poor. In their vision, the next generation implementations of nonmonotonic systems must combine the techniques of relaxed stratification used by DeReS with the techniques used for computing stable model semantics for logic programs used by *s-models*.

DeReS has a prover module that is used like an oracle by all reasoning procedures. Three propositional provers were considered to be used by DeReS:



- **full prover** - sound and complete propositional theorem prover; this would enable DeReS to be used with arbitrary default theories, but it is very inefficient, most of the time spent for proving a formula being actually spent on checking the consistency of the theory;
- **local prover** - local provability propositional tableaux theorem prover; this kind of prover does not perform consistency checks, being sound but not complete.
- **df-lookup** - table lookup method for disjunction-free theories.

The last two provers were included in different variants of the system. The first one enables DeReS to reason with arbitrary propositional default theories, while the second one performs better, but can be used only for disjunction-free theories. The algorithms of DeReS were modified so that a full prover can be replaced with a local prover without affecting the correctness of DeReS.

Several versions of DeReS are available. They include:

- **LDERES** : it can process arbitrary default theories in the original DeReS syntax. It is also possible to use LDERES for programs in the logic programming syntax (SMODELS version). To this end one needs first to convert from SMODELS syntax to DeReS syntax using a program SM2DT. Relaxed stratification is used as the main pruning mechanism.
- **SDERES** : it computes stable models of a normal logic program. `sm2dt` is needed in order to convert from the SMODELS syntax to the DeReS syntax. The table lookup method is used as a prover for this version. Programs computing stratification are also available with SDERES. They improve the performance of the system.
- **STABLE** : it is a close derivative of SMODELS. The main difference is in the heuristic choice of the next atom on which to split. STABLE uses stratification to select an atom for splitting when propagation through constraints is no longer possible. It accepts SMODELS syntax.

All three programs require LPARSE (developed by the SMODELS group) to perform grounding. Among the first two systems, the first one is more complete. The third one, STABLE, is one attempt to combine the technique of relaxed stratification with SMODELS.

WSML Since all three DeReS variants implement the stable model semantics they all can be used for reasoning with WSML Core and WSML Flight. All three variants of the system accept as entries logical programs in the format expected by SMODELS, so they allow the use of the same syntactical constructs. The requirements for WSML Rule are met partially : non-stratified negation is allowed, but not other features envisioned to make part from WSML Rule, like classical negation and function symbols. Due to the fact that DeReS deals only with propositional default theories, it cannot be used as a reasoner for WSML DL and WSML Full. However, if default logic is envisioned to give the nonmonotonic flavor of WSML Full, DeReS deals with a subset of WSML Full that includes propositional logic, which is the class of propositional default theories.

One disadvantage of this system is that is no longer under development, although in the last release there were mentioned some features that still had to be implemented. Among the three variants, STABLE was last updated, but as its man pages indicate, usually it is slower than SMODELS in computing stable models. So, the assumption made by the initiators of the system in [Cholewiński



et al., 1999] regarding the better performance obtained by combining the techniques of SMOBELS with the techniques of relaxed stratification is questionable.

3.4.1.3 Antoniou's approach for defining and computing extensions of default theories

In [Antoniou, 1997] a different interpretation for the notion of extension of a logical theory is given in order to have an operational semantics.

First, let's note that in the same reference is explained and formalized the notion of *open default* (that has open formulae as its components) which is taken for granted in more of the papers dealing with default logic. Thus, an open default is interpreted as a default schema, that is, it represents a *set of defaults* (this set may be infinite). A *default schema* is formally described as:

$$\frac{\alpha^\sigma : \beta_1^\sigma, \dots, \beta_n^\sigma}{\gamma^\sigma}$$

and denotes the set of defaults obtained for all ground substitutions σ that assign values to all free variables occurring in the schema. That means, free variables are interpreted as being universally quantified over the whole default schema.

The interpretation of a default in this approach is the following:

If α is currently known, and if all β_i are consistent with *the current knowledge base*, then conclude γ . The current knowledge base is obtained from the facts and the consequents of some defaults that have been applied previously.

The advantage compared with the classical interpretation is that no theory is used beforehand (the final knowledge base) for evaluating facts, as in Reiter's definitions.

Extensions are defined as current knowledge bases satisfying some conditions. For formally defining them, several additional notions are introduced. Thus, for a given default theory $T = (W, D)$ let $\Pi = (\delta_0, \delta_1, \dots)$ be a finite or infinite sequence of defaults from D without multiple occurrences. Π is called a *process* of T if the defaults from Π can be applied in the given order.

Two sets of first-order formulae are associated with each P_i , namely $In(\Pi)$ and $Out(\Pi)$:

- $In(\Pi)$ is $Th(M)$, where $M = W \cup \{cons(\delta) \mid \delta \text{ occurs in } \Pi\}$. It represents the *current knowledge base* after the defaults in Π have been applied.
- $Out(\Pi) = \{\neg\psi \mid \psi \in just(\delta) \text{ for some } \delta \text{ occurring in } \Pi\}$. It collects the formulae that should not turn out to be true, even after subsequent application of other defaults.

A process Π of T is defined to be:

- *successful* iff $In(\Pi) \cap Out(\Pi) = \emptyset$, otherwise it is *failed*.
- *closed* iff every $\delta \in D$ that is applicable to $In(\Pi)$ already occurs in Π .

Definition 3.3. A set of formulae E is an extension of the default theory T iff there is some closed and successful process Π of T such that $E = In(\Pi)$.

Extensions of a default theory are determined by constructing a process tree, where all possible processes are arranged in a canonical manner and that enables to discriminate between different types of processes.



The underlying language for the supported default theories in this approach is predicate logic, i.e. FOL. A prototype implementation in Prolog is given for default theories with only one justification. Extending this implementation for general default theories does not raise problems, being left as an exercise. The program makes use of an external theorem prover based on sequent calculi, called *sequent*. The sequent calculi is a sound and complete calculi for propositional and predicate logic. It is also said that the algorithm is not very efficient but that it can be improved easily.

For a particular class of default theories, *normal default theories*, extensions are always guaranteed to exist and checking whether a formula is supported by a default theory can be done using a goal-driven approach, namely by constructing a *default proof*. This is much more reasonable than computing each extension of a default theory. A normal theory is one that comprises only *normal defaults*, where a normal default is defined as one that has only one justification identical with its consequent:

$$\frac{\alpha : \gamma}{\gamma}$$

Formally: a *default proof* of φ in a normal default theory $T = (W, D)$ is a finite sequence (D_0, D_1, \dots, D_k) of subsets of D such that:

- φ follows from $W \cup \text{cons}(D_0)$
- for all $i < k$, the prerequisites of defaults in D_i follow from $W \cup \text{cons}(D_{i+1})$
- $d_k = \emptyset$
- $W \cup \text{cons}(\bigcup_i D_i)$ is consistent.

The implementation of such a default proof is not given, but it is not difficult to implement it in Prolog. Unfortunately, normal defaults have quite a limited expressiveness for many common-sense reasoning scenarios.

Another class of defaults that is guaranteed to have at least one extension is the class of semi-normal, plain, ordered default theories. A semi-normal default has the form:

$$\frac{\alpha : \beta \wedge \gamma}{\gamma}$$

Plain refers to the fact that the set of defaults is finite and that the default theory is propositional. For the definition of ordered default theories, the reader is referred to [Antoniou, 1997]. Any default that has only one justification can be translated in a semi-normal default such that the resulting theory has the same extensions as the original one.

WSML Besides the support for WSML Core and WSML Flight, the algorithm that deals with general default theories can be used for reasoning with all the other WSML variants. Thus, it offers the features required for WSML Rule like non-stratified negation, classical negation and function symbols, it covers FOL and implicitly its DL subset that underpins WSML DL and has the non-monotonic flavor required by WSML Full.



4 Implementation

In this chapter we will give a general introduction to a reasoning infrastructure for WSML and detail the implementation efforts for WSML-Core, WSML-Flight, WSML-Rule and WSML-DL on top of DLV (WSML-Flight), KAON2 (WSML-Flight), MINS (WSML-Rule), Ontobroker (WSML-Rule), *FLORA-2* (WSML-Rule) and FaCT++ (WSML-DL), respectively. Note that all reasoners for one of the rule-based variants of WSML can handle also every rule-based language which is contained within that variant. This means that all reasoners for WSML-Flight can deal with WSML-Core and all reasoners for WSML-Rule can also deal with WSML-Flight and thus also with WSML-Core.

The remainder of this chapter is structured as follows. In section 4.1, we introduce a general reasoner infrastructure and describe the current state of implementation. In section 4.2, we describe the implementation of WSML-Core and WSML-Flight and their integration into the WSML reasoner infrastructure. In section 4.3, we present the implementation of WSML-Rule on top of the Ontobroker and the *FLORA-2* reasoner, respectively, as well a reasoner for WSML-DL based on FaCT++. These reasoners have a similar architecture which is described in section 4.3.1. Finally, in section 4.4, we present an outlook to future developments of the reasoners.

4.1 The WSML reasoner infrastructure

In Computer Science, *Reasoning* is commonly understood as the process of inferring "new" (i.e. not explicitly stated) information about some domain of discourse from a given (formal) model of that domain that contains a set of explicit descriptions of properties of the domain, i.e. a given knowledge base. Taking a broader perspective on reasoning, one can understand this term in a much more generic way as the computation based on explicit and formalized knowledge or computation based on logics. Reasoning then allows to exploit information represented in the formal model of the domain of discourse.

In Artificial Intelligence and its applications various forms of reasoning have been investigated for a long time, e.g. for *instance checking*, *logical entailment*, *consequence finding*, *abductive reasoning*, *logic programming* etc. Respective algorithms have been investigated for various logics. Taking this general perspective, a traditional database task, namely finding answers to given queries with respect to some database (instance retrieval), can be seen as a reasoning task as well. This task has been extensively studied in the deductive database community. Therefore it is not surprising that recent approaches to information integration are based on logics as data models and instance retrieval.

It is necessary to distinguish between the various forms of reasoning because they serve significantly different purposes, are implemented by means of a specific algorithm, that takes a well-defined set of input information and is expected to deliver results of a certain kind. The different forms of reasoning (i.e. computation with explicit formal models based on logics) differ significantly in all these aspects. This means that an algorithm to perform some sort of reasoning (in the broader sense) is designed to fulfill a single, well-defined reasoning task. Algorithms for different reasoning task are based on different techniques and thus differ significantly. In general, they will even not share the same infrastructure in terms of the datastructures on which the algorithm operate. In



the following, we will call the different forms of *reasoning tasks*.

The reasoning tasks that we consider as being relevant in the context of WSML are the following:

Logic-based reasoning tasks. Reasoning tasks that are not specific to ontologies but are usually considered with different kinds of knowledge bases. As the most relevant examples for such reasoning tasks, we consider at present:

- Instance Retrieval (IR)
- Logical Entailment (LE)
- Consequence Finding (CF)

Ontology-specific Reasoning tasks (ORT). Reasoning tasks that specifically refer to ontologies and their modelling primitives. Typical examples from the Description Logic Community are:

- Concept Satisfiability: check whether a concept is satisfiable w.r.t. a T-Box
- Concept Subsumption: check whether a subsumption relationship holds between two concepts
- Relation Subsumption: check whether two relations are subsumed by each other
- Classification tasks: e.g. compute a hierarchy of named concepts in the knowledge base (knowledge base classification), identify most specific concepts

Reasoning tasks specific to other top-level elements in WSMO. Reasoning tasks that specifically refer to web services, goals and mediators and the respective additional modelling primitives in WSML. Examples would be:

- Goal-Capability matching
- Choreography matching

We believe that many reasoning tasks of the third-mentioned class are in fact reducible to tasks of the first and second class. At least they will be based on reasoning tasks of the first and second kind to a large extent. For the moment, we will not consider the third class at all.

Please note that there are a lot more reasoning tasks, such as abductive reasoning for the first class of reasoning tasks mentioned above, that might be investigated in the long run, in case that sensible application could be identified.

When it comes to the design and implementation of algorithms for a specific reasoning tasks, another important aspect has to be considered, that potentially results in very different techniques, datastructures and systems: the concrete logic (or formal language) in which input to the reasoning task (such as the knowledge base representing the domain of discourse) is represented. Here, we have to consider the different variants of WSML, namely: WSML-Core, WSML-Flight, WSML-Rule, WSML-DL, WSML-Full and perhaps the monotonic subset of WSML-Full (WSML-FOL).

From an implementation perspective, the two above mentioned dimensions are (largely) orthogonal to each other and span the space of the various different reasoners (or more precisely reasoning algorithms) that need to be integrated into a common WSML Reasoner Framework. Figure 4.1 illustrates the space to be considered. It deliberately leaves out some parts of the space that are not relevant right now or can not be achieved in a foreseeable period of time.

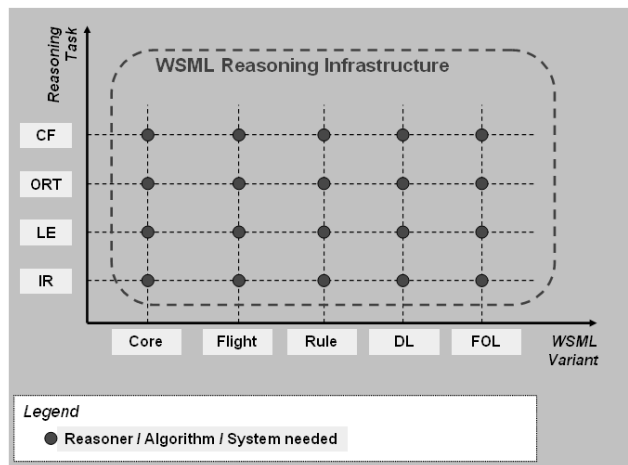


Figure 4.1: The Space of Reasoners to be considered in the WSML Reasoning Framework.

Recently, we started with the implementation of a prototype of the WSML Reasoner Framework described here. During this work, we focused on one specific reasoning task: instance retrieval. We developed a flexible architecture that allows for easy integration of external reasoning components for instance retrieval by means of wrapper classes.

Integrated external reasoning components: So far we integrated the following systems:

- DLV: A disjunctive Datalog engine.
- Mandarax: A naive Java-implementation of a resolution-based Prolog engine.
- Kaon2: An ontology reasoner (for almost all of OWL-DL) based on a disjunctive Datalog.
- MINS: The DERI rule reasoner (which currently is nothing more than a lean version of the SILRI system).

Candidates for additional integration are:

- XSB: A powerful implementation of a Prolog engine.
- *FLORA-2*: A deductive database system based on F-Logic.

As our work shows, integration of new systems does not take longer than around 2 - 4 hours, which means that within a day one can come up with a reasoner for query answering in WSML¹ which supports all features of language² and is based on a specific external tool.

The current limitations of the reasoner infrastructure are: At present, we can not deal with all WSML variants and not with all features of the supported variants. We can support WSML-Core and WSML-Flight. Extension to WSML-Rule will take a few days.

Figure 4.3 overviews the current state with respect to the reasoner space shown in Figure 4.1.

¹At least the WSML Rule fragment

²Depending on what part of WSML the external tool actually can deal with itself.

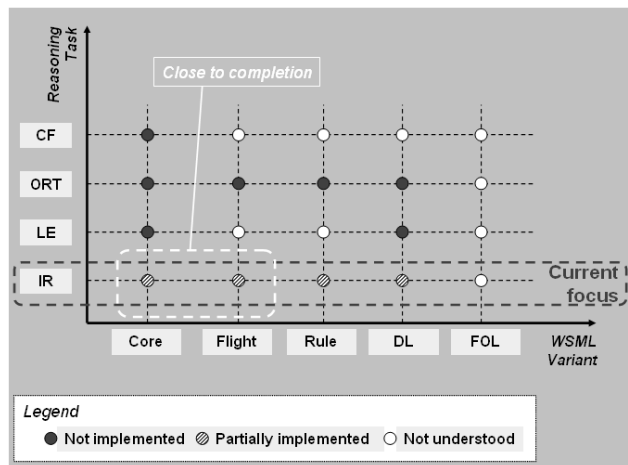


Figure 4.2: Current status of the implementation of the WSML Reasoning Framework.

4.2 WSML-Core and WSML-Flight reasoners

The WMSL-Core and WSML-Flight reasoners have been integrated into the general WSML reasoner architecture. So far only the reasoning task *Instance Retrieval* has been considered.

Instance retrieval is a well-understood reasoning task for the rule-based WSML Variants, namely, WSML-Core, WSMLFlight and WSML-Rule. More precisely, WSML-Core and WSML-Flight can be mapped to Datalog, that means the function-free Horn-subset of First-order logic and to Datalog[¬], that means Datalog extended by a default negation. WSML-Rule can be mapped to Prolog, which means Datalog[¬] extended by function symbols³.

For Datalog, Datalog[¬] and Prolog, there are already various available implementations that can be used to perform query answering on rule bases. Thus, we decided to start with these languages and come up with an architecture that allows us to integrate different tools easily and exploit several person-years of development efforts that has been invested in other groups to come up with powerful systems for this specific task. In particular, we used

- DLV: A disjunctive Datalog engine.
- Mandarax: A naive Java-implementation of a resolution-based Prolog engine.
- Kaon2: An ontology reasoner (for almost all of OWL-DL) based on a disjunctive Datalog.
- MINS: The DERI rule reasoner (which currently is nothing more than a lean version of the SILRI system).

We realized the architecture for the query answering reasoner for WSML-Core and WSML-Flight as shown in Figure 4.3: We defined a general API for creation and interaction with a WSML reasoner of any kind. WSML Ontologies are transformed to a syntactically simpler form and finally to a pure logical representation in terms of Datalog rules. The specific semantics of object-oriented modelling primitives in WSML is captured by auxilliary rules, at is has been

³We consider function symbols to have a positive arity.



done in the Ontobroker system. An API for wrapping external datalog engines has been defined and instantiated for a few different systems. A specific auxiliary datastructure that can be reused for different wrappers is a symbol map which takes care of translating the WSML specific naming convention, in particular IRIs, into symbol names that are actually valid for a specific tool.

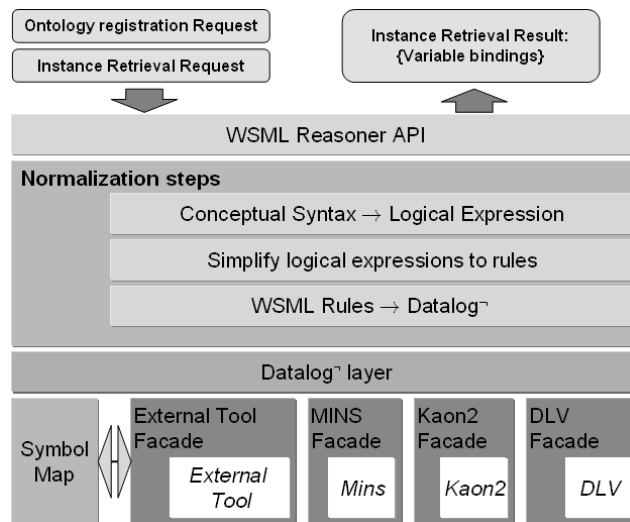


Figure 4.3: Architecture of the Instance Retrieval Reasoners for WSML-Core and WSML-Flight

4.3 WSML-Rule and WSML-DL reasoners

Currently, two implementations of a WSML-Rule reasoner are available: one based on the Ontobroker reasoner and the other based on *FLORA-2* reasoner. Furthermore, an implementation of a WSML-DL reasoner is available based on FaCT++ and its DIG interface. All three reasoners are not integrated in the WSML reasoner infrastructure.

Note that some of the reasoners used for WSML-Core and WSML-Flight 4.2 that are integrated in the WSML reasoner infrastructure can also be used for WSML-Rule. However, the reasoners that are able to deal with WSML-Rule and have been integrated into the reasoner infrastructure, have not been supplemented by a WSML-Rule wrapper.

For the WSML-Rule reasoners, we have implemented a translation from WSML-Rule to OntoBroker F-Logic, respective *FLORA-2* F-Logic, as well as a connection to the OntoBroker reasoner, respective *FLORA-2* reasoner to answer conjunctive queries.

For the WSML-DL reasoner, we have implemented a translation from WSML-DL to the XML-Syntax of Description Logics which is used in the DIG interface.

The OntoBroker Web interface for the reasoner is available at:
<http://dev1.deri.at:8080/wsml/reasoner.html>

The *FLORA-2* Web interface for the reasoner is available at:
<http://dev1.deri.at:8080/wsml/flReasoner.html>

For WSML-DL, a web interface will be provided soon.

The Javadoc for the Ontobroker Reasoner API is available at:



<http://www.wsmo.org/TR/d16/d162/v0.2/api/index.html>

The Javadoc for the *FLORA-2* Reasoner API is available at:

<http://www.wsmo.org/TR/d16/d162/v0.2/apiFloraReasoner/index.html>

For WSML-DL, a Javadoc will be provided soon.

In the remainder of this Chapter, we first describe the general reasoner architecture in Section 4.3.1. Then, we describe in more detail the translation of WSML to OntoBroker F-Logic and WSML to *FLORA-2* in Section 4.3.3. Finally, we describe limitations of the reasoner in Section 4.3.4.2.

4.3.1 Architecture

The general reasoner architecture, depicted in Figure 4.4, consists of the following components:

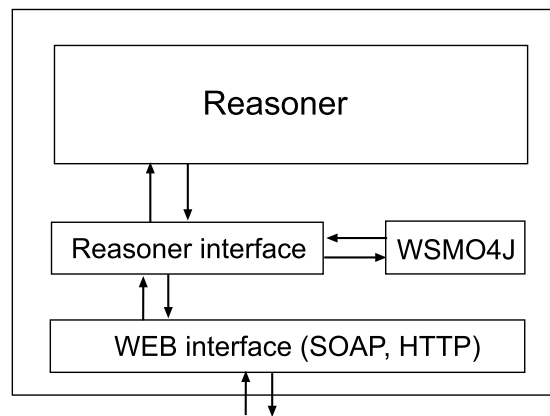


Figure 4.4: Reasoner Architecture

Web Interface The user can access the reasoner via the Web Interface, where the user can input a WSML ontology and a WSML conjunctive query (see Figure 4.5 for OntoBroker and Figure 4.6 for *FLORA-2*). The Web Service interface is exposed through a WSDL description, which can be accessed at:

<http://dev1.deri.at:8080/wsm1/services/reasoner?wsdl> for OntoBroker, and

<http://dev1.deri.at:8080/wsm1/services/flReasoner?wsdl> for *FLORA-2*. As already mentioned in the last section, the Web interface, there is no web interface available for the WSML-DL reasoner yet.

Reasoner interface The reasoner interface implements the translation from WSML-Rule to F-Logic (OntoBroker and *FLORA-2* F-Logic) and executes queries on the reasoner. For the WSML-DL reasoner this interface is provided as an API and implements a translation from WSML-DL to the XML syntax of Description Logics on which the DIG interface bases.

Rule based Reasoner/Description Logics based Reasoner OntoBroker, *FLORA-2* and FaCT++ provide the reasoning services, exposed through the reasoners interfaces (OntoBroker reasoner interface; *FLORA-2* reasoner interface; DIG reasoner interface).



WSMO4J WSMO4J is used by the reasoner interface for the purpose of parsing and in-memory representation of WSML documents. All implementations use version 0.4.0 of WSMO4J.

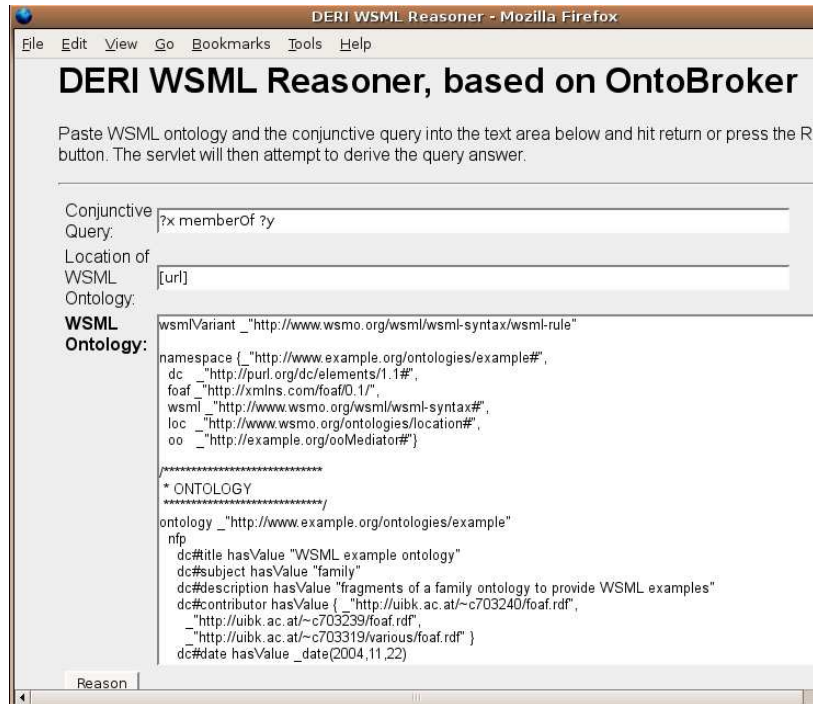


Figure 4.5: WSML/OntoBroker Reasoner Web Interface

The reasoners offer the following functionality:

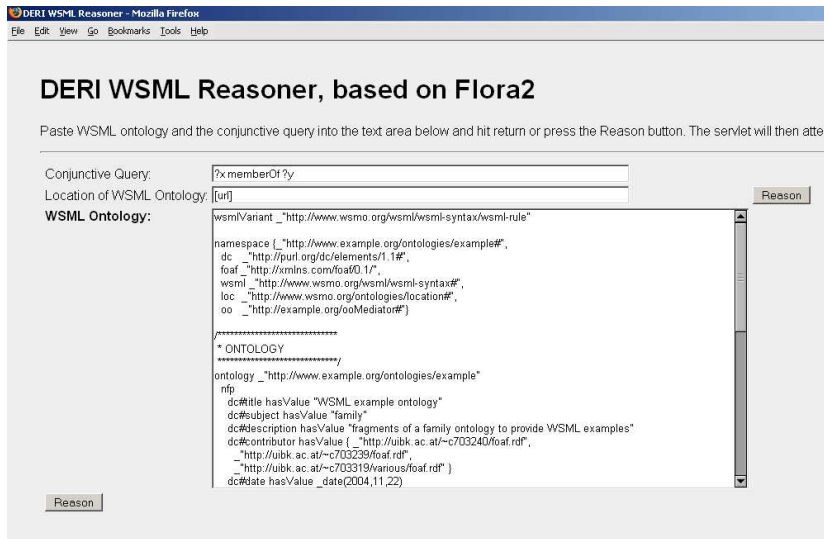
Loading an Ontology Load a WSML Ontology into OntoBroker/*FLORA-2*/FaCT++

Answer query The WSML-Rule reasoners answer a given **conjunctive** WSML query wrt. the Ontology. The WSML-DL reasoners answers queries that query

- for the names of all concepts in the ontology
- for the names of the concept ancestors or direct parents of a given concept
- for the name of the concept descendants or direct children of a given concept
- whether a concept is satisfiable w.r.t. a ontology.
- whether a subsumption relationship holds between two concepts w.r.t. the ontology

4.3.2 WSML-Rule reasoners

In this section, we describe the translation of WSML-Rule to Ontobroker F-Logic and Flora-2 Flogic, respectively. Furthermore, we identify the limitations of the current versions of the reasoners.

Figure 4.6: WSML/*FLORA*-2 Reasoner Web Interface

4.3.3 WSML-Rule to OntoBroker F-Logic and FLORA-2 F-Logic translation

In this Section we outline the translation of WSML-Rule logical expression syntax to OntoBroker F-Logic syntax, respectively to *FLORA*-2 F-Logic syntax. We assume translation WSML-Rule conceptual syntax has been translated to WSML-Rule logical expression syntax, as in Section 8.1 of [de Bruijn et al., 2005].

We first normalize WSML-Rule logical expressions to obtain rules with a single atomic formula in the head and with elimination of molecule which use **impliesType** by applying the following normalization rules until no rule is applicable (with H_n, B logical expression):

- Replace rules of the form H_1 **and** ... **and** H_n :- B . with:
 - H_1 :- B .
 - ...
 - H_n :- B .
- Replace rules of the form H_1 **impliedBy** H_2 :- B . with:
 - H_2 **implies** H_1 :- B .
- Replace rules of the form H_1 **equivalent** H_2 :- B . with:
 - H_2 **implies** H_1 :- B .
 - H_1 **implies** H_2 :- B .
- Replace rules of the form H_1 **implies** H_2 :- B . with:
 - H_1 :- B **and** H_1 .
- Replace molecules of the form $A[B$ **impliesType** $C]$ with:
 - `_"http://www.wsmo.org/wsmo/wsmo-syntax#impliesType"(A,B,C)`



In case the molecule is part of a compound molecule, conjoin (using **and**) the newly obtained atom with the original molecule (without the **impliesType**).

- Split molecules of the form $A \text{ isa } \{B_1, \dots, B_n\}$, with $\text{isa} \in \{\text{memberOf}, \text{subConceptOf}\}$ into n different molecules, conjoined with **and**:

$$- A \text{ isa } B_1 \text{ and } \dots \text{ and } A \text{ isa } B_n$$

Table 4.1 sketches the WSML-Rule constructs with the corresponding OntoBroker F-Logic constructs.

Table 4.2 sketches the WSML-Rule constructs with the corresponding *FLORA-2* F-Logic constructs.

WSML	OntoBroker F-Logic	Remarks
and	AND	
or	OR	
? x (variable)	$\forall x$	
- "http://..." (IRI)	"http://..."	Identifiers are always written as full IRIs, instead of sQNames
$a[b \text{ ofType } c]$	$a[b ==>> c]$	
$a[b \text{ hasValue } c]$	$a[b ->> \{c\}]$	OntoBroker always requires braces { }
$p(a_1, \dots, a_n)$	$p(a_1, \dots, a_n)$	
$a \text{ subConceptOf } b$	$a::b$	
$a \text{ memberOf } b$	$a:b$	
$H :- B$	$H <- B$	
!- B	checkIC(S) <- B AND S is "constraint"	

Table 4.1: WSML-Rule constructs with their correspondence in OntoBroker

WSML	FLORA-2 F-Logic	Remarks
and	,	
or	;	
? x (variable)	$\forall x$	
- "http://..." (IRI)	url("http://...")	Identifiers are always written as full IRIs, instead of sQNames
$a[b \text{ ofType } c]$	$a[b ==>> c]$	
$a[b \text{ hasValue } c]$	$a[b ->> \{c\}]$	
$p(a_1, \dots, a_n)$	$p(a_1, \dots, a_n)$	
$a \text{ subConceptOf } b$	$a::b$	
$a \text{ memberOf } b$	$a:b$	
$H :- B$	$H :- B$	
!- B	checkIC(S) :- B AND S is "constraint"	

Table 4.2: WSML-Rule constructs with their correspondence in *FLORA-2*

Some remarks about the translation:

- All identifiers are written as full IRIs, rather than sQNames. This does not affect the result of the reasoner and it rules out possible conflict between identifiers, variables, and built-in symbols.



- Variables are prefixed with 'V_'. This is done because OntoBroker requires variables to start with an initial capital and using such a prefix keeps the original variable name intact. Same requirements holds for *FLORA-2* therefor we keep the same convention for naming variables.
- OntoBroker does not support integrity constraints. Therefore, we translate an integrity constraint to a rule with the predicate 'checkIC' in the head. We do checking of integrity constraints by querying for this predicate. In case the query returns any result, an integrity constraint is violated.
- OntoBroker requires the explicit quantification of all variables. The translation generated this quantification by prepending **FORALL** with a comma-separated list of all the variable to the rule.

For example, the following WSML logical expression (assuming `http://www.example.org/ontologies/example#` is the default namespace):

```
?x memberOf Human and ageOfHuman(?x,?age)
and ?age =< 14 implies ?x memberOf Child.
```

Translated to OntoBroker F-Logic results in the following rule:

```
FORALL V_age,V_x
V_x:"http://www.example.org/ontologies/example#Child" <-
V_x:"http://www.example.org/ontologies/example#Human" AND
"http://www.example.org/ontologies/example#ageOfHuman"(V_x,V_age)
AND lessorequal(V_age, 14).
```

And translated to *FLORA-2* results in the following rule:

```
V_x:'http://www.example.org/ontologies/example#Child' :-
V_x:'http://www.example.org/ontologies/example#Human' ,
'http://www.example.org/ontologies/example#ageOfHuman'(V_x,V_age)
, V_age <= 14.
```

Finally, for the OntoBroker implementation we add a number of F-Logic rules to enforce the expected behavior of several WSML language constructs:

```
// universal truth
"http://www.wsmo.org/wsml/wsml-syntax#true".

// universal falsehood
FORALL checkIC(S) <- "http://www.wsmo.org/wsml/wsml-syntax#false" AND
S is "universal falsehood".

// impliesType semantics
FORALL W,X,Y,Z W:Z <-
"http://www.wsmo.org/wsml/wsml-syntax#impliesType"(X,Y,Z) AND X[Y->W].;
```

4.3.3.1 Limitations

The following are the limitations of the current version of the reasoner:

Limited Web Access The OntoBroker does not (yet) run on the Web Server and thus the Web front-end for the reasoner has only limited functionality, i.e., it currently only shows a translation of the ontology and the query to OntoBroker F-Logic. See Figure 4.7 for an example. We expect to resolve this issue shortly.

Only logical expressions The reasoner currently only takes logical expressions into account, since for the current version of WSML (v0.2), WSMO4J only supports the logical expression syntax and not the conceptual syntax. This issue will be resolved as soon as WSMO4J will support WSMLv0.2.



```

DERI WSML Reasoning result - Mozilla Firefox
File Edit View Go Bookmarks Tools Help

OntoBroker F-Logic Ontology

Here follows the OntoBroker F-Logic version of the input WSML ontology:

FORALL V_obit,V_x V_x|"http://www.example.org/ontologies/example#isAlive" -> {_boolean("false")}}
FORALL V_obit,V_x V_x|"http://www.example.org/ontologies/example#isAlive" -> {_boolean("true")}}
FORALL V_age,V_x V_x:"http://www.example.org/ontologies/example#Child" <- V_x:"http://www.example.
"http://www.wsmo.org/wsml/wsml-syntax#true".
FORALL checkIC(S) <- "http://www.wsmo.org/wsml/wsml-syntax#false" AND S is "universal falsehood".
FORALL W,X,Y,Z W:Z <- "http://www.wsmo.org/wsml/wsml-syntax#impliesType"(X,Y,Z) AND X{Y->W}.

Query

Here follows the query:

FORALL V_x,V_y <- V_x:V_y.

```

Figure 4.7: Reasoner output

4.3.4 WSML-DL reasoner

In this section, we describe the translation of WSML-DL to the Description Logics Syntax of the DIG interface which bases on an XML Schema definition. Furthermore, we identify the limitations of the current version of the WSML-DL reasoner.

4.3.4.1 WSML-DL to DIG translation

In this Section we outline the translation of WSML-DL logical expression syntax to the XML syntax for Description Logics of the DIG interface which transmits the ontology and the query to the Description Logics reasoner FaCT++. We consider here currently only the WSML-DL logical expressions syntax which has been presented in section 4.1 of [de Bruijn et al., 2005].

In Table 4.3, the translation of WSML-DL concept descriptions is shown. Note that $C(?x)$ and $?x \text{ memberOf } C$ can be replaced by each other. The same holds for $R(?x, ?y)$ and $?x[R \text{ hasValue } ?y]$. Note also that the usage of C outside of the context of a `memberOf` keyword or without being part of a unary relation denotes arbitrary concept descriptions. The WSML keyword **impliedBy** is not mentioned explicitly here but treated in a symmetric way as the WSML keyword **implies**.

In the following table 4.4, the axiom translations are shown. Note again that C , C_1 and C_2 denote arbitrary concept descriptions when not used in combination with the `memberOf` keyword or when used as a unary predicate.

As an Example of the translation consider the following ontology:

```

Human subConceptOf Mammal.
Mammal subConceptOf Animal.
?x memberOf Dog implies ?x memberOf Mammal.
SmallDog(?x) implies Dog(?x).
Dog(?x) impliedBy BigDog(?x).

Human(?x) and Dog(?x) implies false.
?x memberOf SmallDogOwner implies Human(?x) and forall ?x (hasDog(?x, ?y) implies SmallDog
(?y)).
BigDogowner(?x) implies ((?x memberOf Human) and (forall ?y (?x[hasDog hasValue ?y] implies
(?y memberOf BigDog)))).

Human(anne).
clare memberOf Human.

```



WSML	XML syntax of DIG
$C(?X)$	<code><catom name="C"/></code>
$?x \text{ memberOf } C$	<code><catom name="C"/></code>
$R(?x, ?y)$	<code><ratom name="R"/></code>
$?x[R \text{ hasValue } ?y]$	<code><ratom name="R"/></code>
$F1 \text{ and } F2$	<code><and>F1 F2</and></code>
$F1 \text{ or } F2$	<code><or>F1 F2</or></code>
$\text{neg } F1$	<code><not>F1</not></code>
$\text{forall } ?y (R(?x, ?y) \text{ implies } C)$	<code><all>R C</all></code>
$\text{exists } ?y (R(?x, ?y) \text{ and } C)$	<code><some>R C</some></code>
$\text{exists } \{?x_1, \dots, ?x_n\}$ $(R(?y, ?x_1) \text{ and } \dots \text{ and } R(?y, ?x_n))$ $\text{and } C$	<code><atleast num="n">R C </atleast></code>
$\text{and } \text{neg}(?x_1 := ?x_2) \text{ and } \dots$ $\text{and } \text{neg}(?x_{n-1} := ?x_n)$	
$\text{forall } \{?x_1, \dots, ?x_{n+1}\}$ $(R(?y, ?x_1) \text{ and } \dots \text{ and } R(?y, ?x_{n+1}))$ $\text{and } C$	<code><atmost num="n">R C</atmost></code>
$\text{implies } (?x_1 := ?x_2) \text{ or } \dots$ $\text{or } (?x_n := ?x_{n+1})$	

Table 4.3: WSML-DL concept and role descriptions with their correspondence in the XML syntax of the DIG interface

WSML	XML syntax of DIG
$C(\text{inst}_1)$	<code><instanceof>inst₁ C</instanceof></code>
$\text{inst}_1 \text{ memberOf } C$	<code><instanceof>inst₁ C</instanceof></code>
$R(\text{inst}_1, \text{inst}_2)$	<code><related>inst₁ R inst₂</related></code>
$\text{inst}_1[R \text{ hasValue } \text{inst}_2]$	<code><related>inst₁ R inst₂</related></code>
$R_1(?x, ?y) \text{ implies } R_2(?x, ?y)$	<code><impliesr>R₁ R₂</impliesr></code>
$R_1(?x, ?y) \text{ implies } R_2(?y, ?x)$	<code><impliesr> R₁ <inverse>R₂</inverse> </impliesr></code>
$R(?x, ?y) \text{ implies } R(?y, ?x)$	<code><equalr></code> <code> R <inverse>R</inverse></code> <code></equalr></code>
$R(?x, ?z) \text{ and } R(?z, ?y) \text{ implies } R(?x, ?z)$	<code><transitive>R</transitive></code>
$R_1(?x, ?y) \text{ equivalent } R_2(?x, ?y)$	<code><equalr>R₁ R₂</equalr></code>
$?x \text{ memberOf } C_1 \text{ subConceptOf } C_2(?x)$	<code><impliesc>C₁ C₂</impliesc></code>
$C_1 \text{ implies } C_2$	<code><impliesc>C₁ C₂</impliesc></code>
$C_1 \text{ equivalent } C_2$	<code><equalr>C₁ C₂</equalr></code>

Table 4.4: WSML-DL T-Box and A-Box axioms with their correspondence in the XML syntax of the DIG interface



```
Human(ellen).
BigDog(paul).
SmallDog(Arthur).
hasDog(anne, paul).
clare [hasDog hasValue arthur].
```

After the translation into the XML representation for the DIG interface, the ontology has the following format:

```
<implies>Human Mammal</implies>
<implies>Mammal Animal</implies>
<implies>Dog Mammal</implies>
<implies>SmallDog Dog</implies>
<implies>BigDog Dog</implies>
<implies>
  <and>Human Dog</and> <bottom/>
</implies>
<implies>SmallDogOwner <all>hasDog SmallDog</all></implies>
<implies>BigDogOwner <all>hasDog BigDog</all></implies>

<instanceof>anne Human</instanceof>
<instanceof>clare Human</instanceof>
<instanceof>ellen Human</instanceof>
<instanceof>paul BigDog</instanceof>
<instanceof>arthur SmallDog</instanceof>
<related>anne hasDog paul</related>
<related>clare hasDog arthur</related>
```

Remarks concerning the WSML-DL translation:

- The WSML keywords **true** and **false** are translated to **<top/>** and **<bottom/>**, respectively.
- The equality keyword **==:** is currently not considered in the WSMO4J logical expressions API. Therefore, equality needs currently to be written as a relational expression.
- The current WSML-DL syntax bases on first order logic enriched by frame logic constructs. This is very inappropriate for formulating the description logic descriptions and axioms. The syntax is very verbose and thus error prone. An easier to write and easier to comprehend syntax would be more appropriate and would be a lookalike to typical description logics syntaxes.
- The queries are very simple at the moment and consist of either
 - no WSML-DL specification at all (i.e. for querying for all concepts in the ontology no specific WSML-DL query needs to be specified)
 - one concept description (i.e. for querying for all ancestors, direct parents, descendants and direct children of a concept or querying if a concept w.r.t. an ontology is satisfiable)
 - two concept descriptions (i.e. for querying whether a subsumption relationship holds between two concepts w.r.t. the ontology)

4.3.4.2 Limitations

The following are the limitations of the current version of the WSML-DL reasoner:

No A-Box reasoning support FaCT++ doesn't support A-box reasoning tasks.

Limited Datatype Support DIG as well as FaCT++ supports only strings and integers as built-in datatypes. Thus the extensive use of XML Schema datatypes as envisioned in WSML is not possible with the WSML-DL reasoner.



Only restricted logical expressions The reasoner currently takes only logical expressions into account. Furthermore, the logical expressions are restricted as no compound molecules are allowed and no molecules with the keywords `impliesType` or `ofType` are accepted by the reasoner. Furthermore, the usage of arbitrary compound concept descriptions is not possible yet in qualified number restrictions, existential quantifications and value restrictions. There, only atomic concept descriptions are currently allowed.

4.4 Outlook

In this section we present our aims for the near future. In subsection 4.4.1 we present our aims and plans for the WSML reasoner architecture.

4.4.1 WSML reasoner architecture

In the near future we will continue to focus on query answering mainly. Additionally, we will implement some of the ontology related reasoning tasks.

Instance Retrieval. We plan to complete the support for instance retrieval in WSML-Core and WSML-Flight such that all features of WSML are covered. That means datatypes and anonymous ids have to be dealt with. This can be done until end of September.

Ontology import will be dealt with as well in a simple form, since a serious approach will require an implementation of some sort of sophisticated (distributed) Ontology registry. At present, we believe that a simple prototype for such a distributed registry could be realized with moderate effort on top of the P-Grid system, developed at the ETH Lausanne, Switzerland.

Integration of XSB for Datalog has not primary priority and thus will be skipped for the next few weeks.

Basic constraint handling will be included until mid of September.

Until mid of October the extension to WSML-Rule should be completed and thus a reasoner for WSML-Rule supporting all WSML features should be available.

In parallel, we will start with the development of the DERI rule engine, called MINS4 which essentially is a reimplement of the SILRI rule system. This reimplement will take some time. It is unrealistic that it will be ready until the end of the year in a stable version. However, some first prototype system will be possible, depending on how much time the developers can spend on this project.

Ontology related Reasoning tasks. We will implement some of the ontology related reasoning tasks for WSML-Core, WSML-Flight, WSML-Rule and WSML-DL until the end of the year. For those tasks that can be simply implemented as query answering problems, this will only need a few days. Hence, they will be available already until

Logical entailment checking. Logical entailment checking for WSML-Core is not difficult to implement and will be realized until the end of November. For WSML-DL some prototype should be possible until the end of the year. For the other WSML Variants contain non-monotonic features (i.e. default negation) this will be more difficult and not be ready until the end of the year. However and implementation for the non-monotonic part of these languages should be possible until the end of the year as well.



Everything else. It is unrealistic that we will consider other reasoning tasks and languages until the end of the year. Hence, they are out of scope for now. For some of them, more research and clarification is needed.

4.4.2 Ontobroker

The following is possible functionality to be added to the OntoBroker reasoner:

Connection to databases Most data is stored in relational databases. Future versions of the reasoner might connect to existing databases to retrieve instance information. Care needs to be taken as to how the connection between the ontology and the WSML ontology is realized. One possible tool to realize this is the OntoMap tool from Ontoprise.

One limitation is that OntoBroker can currently only connect to the commercial database systems MS SQL Server and Oracle.

Concurrency Currently, the reasoner architecture does not allow multiple reasoner clients to connect to the same OntoBroker instance. In fact, when multiple clients connect to the same OntoBroker, clients might encounter unexpected behavior. Possible ways of enabling concurrency are:

- Using a separate module for each reasoner client connecting to the same OntoBroker.
- Using a new instance of OntoBroker for each client.

The latter may require a lot of additional work, since it is not straightforward to start multiple instances of OntoBroker.

Reasoning with Goals and Web Services The operational semantics, as well as the applicable reasoning tasks, of Goals and Web Services has not yet been defined. When these have been defined, they might be implemented in the WSML reasoner.

4.4.3 WSML-DL reasoners

On the theoretical side, there needs to be done further research on the translation of first order logic to description logics. Also, the need for a DL-style-like WSML-DL Syntax needs to be evaluated.

Concerning the implementation of the first prototype of a WSML-DL reasoner based on FaCT++, we will focus on a smoother and run-time enhanced implementation. We will also switch to the recently released WSMO4J 0.5.0. Furthermore, we want to add support for all types of valid WSML-DL logical expressions.

On the long run, a WSML-DL reasoner based on Racer is also envisioned. Pellet being a OWL-DL reasoner is not as appropriate a WSML-DL reasoner because it doesn't support qualified number restrictions.



5 Conclusions and Future Work

In this deliverable, we have surveyed several logical reasoners and evaluated them in the context of possibilities of usage for the different WSML variants.

We have presented a WSML reasoner infrastructure and implementations of WSML-Core, WSML-Flight, WSML-Rule and WSML-DL reasoners as initial reasoner implementations. We envision further development of these reasoners.

5.1 Related Developments

The development of the WSML reasoners is closely related with the development of the WSMO discovery engine (deliverable D5.2 [Keller et al., 2005]), which will provide requirements on the WSML reasoners and which will eventually make use of the WSML reasoners.

Other related developments are:

- WSMO4J¹, which will provide a data model in Java for WSML and will also provide (de-)serializers for the different WSML syntaxes. WSMO4J can be extended to connect with the specific reasoners to be used for WSML.
- The WSML validator² currently provides validation services for the basic syntax defined in D2v1.0 [Roman et al., 2004]. We expect the validator to be extended to handle the different WSML variants under development in D16.1 [de Bruijn et al., 2005]. However, we expect that the WSML validator will eventually be subsumed by WSMO4J.
- WSMX provides the reference implementation for WSMO. WSMX makes use of pluggable reasoning services. The only component in WSMX of which we are aware that it uses a reasoner is the mediation component, which uses the *FLORA-2* [Yang et al., 2003] logic programming engine. We expect that in future versions, WSMX will make use of the reasoners to be provided in the context of this deliverable for tasks such as mediation, discovery and composition. Therefore, future versions of this deliverable will also take specific requirements on the reasoners coming from WSMX into account.

Acknowledgements

The work is funded by the European Commission under the projects DIP, Knowledge Web, InfraWebs, SEKT, SWWS, ASG and Esperanto; by Science Foundation Ireland under the DERI-Lion project; by the FIT-IT (Forschung, Innovation, Technologie - Informationstechnologie) under the projects RW² and TSC.

The editors would like to thank to all the members of the WSML working group for their advice and input into this document.

¹<http://wsmo4j.sourceforge.net/>

²<http://dev1.deri.at:8080/wsml/validator.html>



Bibliography

- [Antoniou, 1997] Antoniou, G. (1997). *Nonmonotonic Reasoning*. MIT Press.
- [Baader et al., 2003] Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., and Patel-Schneider, P. F., editors (2003). *The Description Logic Handbook*. Cambridge University Press.
- [Bonner and Kifer, 1998] Bonner, A. and Kifer, M. (1998). A logic for programming database transactions. In Chomicki, J. and Saake, G., editors, *Logics for Databases and Information Systems*, chapter 5, pages 117–166. Kluwer Academic Publishers.
- [Calvanese et al., 1998] Calvanese, D., Giancomo, G. D., and Lenzerini, M. (1998). On the decidability of query containment under constraints. In *Proc. of the 17th ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS'98)*, pages 149–158.
- [Calvanese et al., 2003] Calvanese, D., Giancomo, G. D., and Vardi, M. Y. (2003). Decidable containment of recursive queries. In *Proc. of the The 9th International Conference on Database Theory (ICDT 2003)*, pages 327–342.
- [Chandra and Merlin, 1977] Chandra, A. K. and Merlin, P. M. (1977). Optimal Implementation of Conjunctive Queries in Relational Data Bases. In *Conference Record of the Ninth Annual ACM Symposium on Theory of Computing*, pages 77–90.
- [Chaudhri et al., 1998] Chaudhri, V. K., Farquhar, A., Fikes, R., Karp, P. D., and Rice, J. P. (1998). OKBC: A programmatic foundation for knowledge base interoperability. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pages 600–607, Madison, Wisconsin, USA. MIT Press.
- [Chen et al., 1993] Chen, W., Kifer, M., and Warren, D. S. (1993). HILOG: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230.
- [Chen and Warren, 1996] Chen, W. and Warren, D. S. (1996). Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74.
- [Cholewiński, 1995a] Cholewiński, P. (1995a). Reasoning with stratified default theories. In *Proceedings of LPNMR'95*, number 928 in Lecture Notes in Computer Science, pages 273–286, Berlin. Springer-Verlag.
- [Cholewiński, 1995b] Cholewiński, P. (1995b). Stratified default theories. In *Proceedings of CSL'94*, number 933 in Lecture Notes in Computer Science, pages 456–470, Berlin. Springer-Verlag.
- [Cholewiński et al., 1999] Cholewiński, P., W.Marek, V., Mikitiuk, A., and Truszczyński, M. (1999). Computing with default logic. *Artificial Intelligence*, 112(1-2):105–146.
- [Cholewiński et al., 1996] Cholewiński, P., W.Marek, V., and Truszczyński, M. (1996). Default reasoning system deres. In *Proceedings of KR-96*. Morgan Kaufman.



- [Dantsin et al., 2001] Dantsin, E., Eiter, T., Gottlob, G., and Voronkov, A. (2001). Complexity and expressive power of logic programming. *ACM Computing Surveys (CSUR)*, 33(3):374–425.
- [Davis, 2001] Davis, M. (2001). The early history of automated deduction. In Robinson, A. and Voronkov, A., editors, *Handbook of Automated Reasoning*, volume I, chapter 1, pages 3–15. Elsevier Science.
- [de Bruijn et al., 2005] de Bruijn, J., Lausen, H., Krummenacher, R., Polleres, A., Predoiu, L., Kifer, M., and Fensel, D. (2005). The web service modeling language WSML. Technical report, WSML. WSML Final Draft D16.1v0.2. <http://www.wsmo.org/2004/d16/d16.1/v0.2/>.
- [de Bruijn et al., 2004a] de Bruijn, J., Polleres, A., Lara, R., and Fensel, D. (2004a). OWL⁻. Deliverable d20.1v0.2, WSML. Available from <http://www.wsmo.org/2004/d20/d20.1/v0.2/>.
- [de Bruijn et al., 2004b] de Bruijn, J., Polleres, A., Lara, R., and Fensel, D. (2004b). OWL flight. Deliverable d20.3v0.1, WSML. Available from <http://www.wsmo.org/2004/d20/d20.3/v0.1/>.
- [Eiter et al., 2004] Eiter, T., Lukasiewicz, T., Schindlauer, R., and Tompits, H. (2004). Combining answer set programming with description logics for the semantic web. In *Proc. of the International Conference of Knowledge Representation and Reasoning (KR04)*.
- [Fensel et al., 2000] Fensel, D., Angele, J., Decker, S., Erdmann, M., Schnurr, H.-P., Studer, R., and Witt, A. (2000). Lessons learned from applying AI to the web. *International Journal of Cooperative Information Systems*, 9(4):361–382.
- [Fitting, 1996] Fitting, M. (1996). *First-Order Logic and Automated Theorem Proving*. Springer-Verlag, second edition edition.
- [Gelder et al., 1988] Gelder, A. V., Ross, K., and Schlipf, J. S. (1988). Unfounded sets and well-founded semantics for general logic programs. In *Proceedings 7th ACM Symposium on Principles of Database Systems*, pages 221–230, Austin, Texas. ACM.
- [Gelfond and Lifschitz, 1988] Gelfond, M. and Lifschitz, V. (1988). The stable model semantics for logic programming. In Kowalski, R. A. and Bowen, K., editors, *Proceedings of the Fifth International Conference on Logic Programming*, pages 1070–1080, Cambridge, Massachusetts. The MIT Press.
- [Gelfond and Lifschitz, 1991] Gelfond, M. and Lifschitz, V. (1991). Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4):365–386.
- [Genesereth, 1991] Genesereth, M. (1991). Knowledge interchange format. In Allenet, J. et al., editors, *Proceedings of the Second International Conference on the Principles of Knowledge Representation and Reasoning (KR-91)*, pages 238–249. Morgan Kaufman. See also <http://logic.stanford.edu/kif/kif.html>.
- [Genesereth and Fikes, 1992] Genesereth, M. R. and Fikes, R. E. (1992). Knowledge interchange format, version 3.0 reference manual. Technical report logic-92-1, Computer Science Department, Stanford University, Stanford, US,.



- [Grosz et al., 2003] Grosz, B. N., Horrocks, I., Volz, R., and Decker, S. (2003). Description logic programs: Combining logic programs with description logic. In *Proc. Intl. Conf. on the World Wide Web (WWW-2003)*, Budapest, Hungary.
- [Horrocks, 1998] Horrocks, I. (1998). The fact system. In de Swart, H., editor, *Automated Reasoning with Analytic Tableaux and Related Methods: International Conference Tableaux98*, number 1397 in Lecture Notes in Artificial Intelligence, pages 307–312. Springer-Verlag.
- [Horrocks and Patel-Schneider, 2003] Horrocks, I. and Patel-Schneider, P. F. (2003). Reducing OWL entailment to description logic satisfiability. In *Proc. of the 2003 International Semantic Web Conference (ISWC 2003)*, Sanibel Island, Florida.
- [Horrocks et al., 2000] Horrocks, I., Sattler, U., and Tobies, S. (2000). Practical reasoning for very expressive description logics. *Logic Journal of the IGPL*, 8(3):239–264.
- [Hustadt et al., 2004a] Hustadt, U., Motik, B., and Sattler, U. (2004a). Reasoning for description logics around SHIQ in a resolution framework. Technical Report 3-8-04/04, FZI.
- [Hustadt et al., 2004b] Hustadt, U., Motik, B., and Sattler, U. (2004b). Reducing SHIQ⁻ description logic to disjunctive logic programs. FZI-Report 1-8-11/03, revised version, FZI.
- [Janhunen et al., 2000] Janhunen, T., Niemelä, I., Simons, P., and You, J.-H. (2000). Partiality and Disjunctions in Stable Model Semantics. In Cohn, A. G., Giunchiglia, F., and Selman, B., editors, *Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR 2000)*, April 12-15, Breckenridge, Colorado, USA, pages 411–419. Morgan Kaufmann Publishers, Inc.
- [Keller et al., 2005] Keller, U., Lara, R., Lausen, H., Polleres, A., Predoiu, L., and Toma, I. (2005). Wsmx discovery. Technical report, WSMX. WSMX Working Draft D10v0.2. Available from <http://www.wsmo.org/TR/d10/v0.2/>.
- [Kifer et al., 1995] Kifer, M., Lausen, G., and Wu, J. (1995). Logical foundations of object-oriented and frame-based languages. *JACM*, 42(4):741–843.
- [Lifschitz, 1999] Lifschitz, V. (1999). *Action languages, answer sets and planning*, pages 357–373. Springer Verlag.
- [Lloyd, 1987] Lloyd, J. W. (1987). *Foundations of Logic Programming (2nd edition)*. Springer-Verlag.
- [McCarthy, 1986] McCarthy, J. (1986). Applications of circumscription to formalizing common sense knowledge. *Artificial Intelligence*, 28:89–116. Reprinted in [McCarthy, 1990].
- [McCarthy, 1990] McCarthy, J. (1990). *Formalization of common sense, papers by John McCarthy edited by V. Lifschitz*. Ablex.
- [Niemelä, 1999] Niemelä, I. (1999). Logic Programming with Stable Model Semantics as Constraint Programming Paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3–4):241–273.



- [Pan and Horrocks, 2004] Pan, J. Z. and Horrocks, I. (2004). OWL-E: Extending OWL with expressive datatype expressions. IMG Technical Report IMG/2004/KR-SW-01/v1.0, Victoria University of Manchester. Available from <http://dl-web.man.ac.uk/Doc/IMGTR-OWL-E.pdf>.
- [Pease et al., 2000] Pease, A., Chaudhri, V., Lehman, F., and Farquhar, A. (2000). Practical knowledge representation and the darpa high performance knowledge base project. In *Seventh International Conference on Principles of Knowledge Representation and Reasoning*, Breckenridge, CO.
- [Przymusinski, 1991] Przymusinski, T. C. (1991). Stable semantics for disjunctive programs. *New Generation Computing*, 9:401–424.
- [Reiter, 1980] Reiter, R. (1980). A logic for default reasoning. *Artificial Intelligence*, 13:81 – 132.
- [Riazanov and Voronkov, 2002] Riazanov, A. and Voronkov, A. (2002). The Design and Implementation of Vampire. *AI Communications*, 15(2-3):91–110.
- [Roman et al., 2004] Roman, D., Lausen, H., and Keller, U. (2004). Web service modeling ontology standard (WSMO-standard). Working Draft D2v1.0, WSMO. Available from <http://www.wsmo.org/2004/d2/v1.0/>.
- [Sakama and Inoue, 1993] Sakama, C. and Inoue, K. (1993). Relating disjunctive logic programs to default theories. In *Proc. of the 2nd International Conference on Logic Programming and Non-Monotonic Reasoning*, pages 266–282. MIT Press.
- [Simons et al., 2002] Simons, P., Niemelä, I., and Sooinen, T. (2002). Extending and Implementing the Stable Model Semantics. *Artificial Intelligence*, 138:181–234.
- [Tsarkov and Horrocks, 2003a] Tsarkov, D. and Horrocks, I. (2003a). DL reasoner vs. first-order prover. In *Proc. of the 2003 Description Logic Workshop (DL 2003)*, volume 81 of *CEUR* (<http://ceur-ws.org/>), pages 152–159.
- [Tsarkov and Horrocks, 2003b] Tsarkov, D. and Horrocks, I. (2003b). Reasoner prototype. implementing new reasoner with datatypes support. Technical report, WonderWeb project.
- [Tsarkov et al., 2004] Tsarkov, D., Riazanov, A., Bechhofer, S., and Horrocks, I. (2004). Using vampire to reason with owl. In *Proc. of the 2004 International Semantic Web Conference (ISWC 2004)*. To appear.
- [Ullman, 1988] Ullman, J. D. (1988). *Principles of Database and Knowledge-Base Systems, Volume I*. Computer Science Press.
- [Ullman, 1989] Ullman, J. D. (1989). *Principles of Database and Knowledge-Base Systems, Volume II*. Computer Science Press.
- [Waldinger, 2004] Waldinger, R. (2004). Formulation and validation of the axiomatic semantics of owl. available at: <http://www.ai.sri.com/daml/owl/axiomatic.htm>.
- [Wielemaker, 2003] Wielemaker, J. (2003). An overview of the SWI-Prolog programming environment. In Mesnard, F. and Serebenik, A., editors, *Proceedings of the 13th International Workshop on Logic Programming Environments*, pages 1–16, Heverlee, Belgium. Katholieke Universiteit Leuven. CW 371.



[Yang et al., 2003] Yang, G., Kifer, M., and Zhao, C. (2003). FLORA-2: A rule-based knowledge representation and inference infrastructure for the semantic web. In *Proceedings of the Second International Conference on Ontologies, Databases and Applications of Semantics (ODBASE)*, Catania, Sicily, Italy.