WSML Deliverable

D20.3 v0.1
OWL Flight

WSML Working Draft – August 23, 2004

Authors:
Jos de Bruijn
Axel Polleres
Rubén Lara
Dieter Fensel

Editors:
Jos de Bruijn

Reviewers:
Ian Horrocks

This version:
`http://www.wsmo.org/2004/d20/d20.3/v0.1/20040823/`

Latest version:
`http://www.wsmo.org/2004/d20/d20.3/v0.1/`

Abstract

We introduce OWL Flight, an extension of OWL$^-$ (a subset of OWL, which can be translated to Datalog) with different kinds of constraints and a form of the local closed-world assumption, as well as support for datatypes based on the OWL-E datatypes extension for OWL.
The resulting language can be evaluated on a Datalog engine with support for integrity constraints, inequality (in the rule body) and default negation.
OWL Flight will form the basis for the WSML-Flight concrete syntax and will also form the basis for rule extensions, which can be straightforwardly added because OWL Flight is defined in terms of a rule language.

Contents

1 Introduction

We extend OW$^-$ with support for datatypes, based on the datatype group approach of OWL-E [Pan and Horrocks, 2004]. OWL-E is already an extension of OWL. Therefore, OWL$^-$ with datatype support is no longer a strict subset of OWL. However, we think the extension is justified, because practical applications require a more elaborate datatype support (e.g. datatype predicates) than OWL has to offer.

It turns out that the datatype support brings the language outside of plain Datalog. More specifically, the language needs the notion of *integrity constraints* to implement the language in Datalog.

In the WSML deliverable D20.1 [de Bruijn et al., 2004b], we have identified limitations of the ontology language OWL and have introduced a subset of OWL, called OWL$^-$, which can be evaluated using current datalog implementations.

OWL$^-$ already overcomes some of the drawbacks of OWL, but in some cases the expressivity had to be reduced significantly in order to get the desired effects (mainly the translation to Datalog). For example, we had to leave out most cardinality restrictions, because they introduced computational complexity. Furthermore, many limitations of OWL still exist in OWL$^-$, such as the lack of property value and cardinality constraints and the lack of negation.

In this deliverable we aim to overcome these limitations by developing an extension of OWL$^-$, called OWL Flight. OWL Flight supports the previously mentioned datatypes, as well as cardinality constraints and value constraints. OWL Flight is based on the Logic Programming formalism Datalog (with F-Logic [Kifer et al., 1995] syntactic sugar), extended with support for integrity constraints, inequality (in the body of the rule) and default negation.

1.1 A note on terminology

Restrictions, assertions and constraints First, we explain the difference between restrictions, assertions and constraints. These terms all refer to some aspect of a property, such as the cardinality or the value.

In OWL, it is possible to specify both cardinality and value *restrictions* on properties. Such a restriction *restricts* the cardinality of the property to a certain number or restricts the value of a property filler to be of a certain type. In OWL, such restrictions are interpreted as *assertions*, which means that the cardinality is asserted to be of a certain number, i.e. instances are created or equality is derived during reasoning to make the knowledge fit the assertions, and the value is asserted to be of a certain type, i.e. the reasoner concludes that the type of a property value fulfills the assertion.

Another way of interpreting restrictions, as is usually done in databases, is to interpreted them as constraints, which means that if the cardinality is not *known* to be of the specified number or the value of the property is not *known* to be of a certain type, the constraint is violated and the knowledge is inconsistent with respect to the constraints.

In other words, in the case of assertive restrictions, models are created of the conjunction of the theory and the restrictions. In the case of constraining restrictions, models are created of the theory alone, after which it is checked whether the models satisfy the constraints.

Features extending Datalog We refer in this document to features which extend Datalog in order to explain the complexity and expressivity of adding certain features in the language. By describing these features explicitly, we also indicate what kind of implementations could be used for reasoning with an ontology language incorporating these issues.

We refer to Datalog extended with certain features as 'Datalog(*feature-list*)'. For example, Datalog(IC,≠) denotes Datalog extended with support for integrity constraints and inequality (in the body). We distinguish the following Datalog extensions, which are mentioned throughout the chapter:

- **IC**, which denotes integrity constraints. An integrity constraint is a rule without a head. If all elements of the body are true, the constraint is violated and an inconsistency is derived.
- ≠, which denotes inequality (in the body of a rule). Inequality is often implemented as a built-in predicate in the Logic Programming engine, but can also be axiomatized.
- **not**, which denotes default negation. A literal *not p* is satisfied if p is not known to be true. Default negation implements a form of the closed-world assumption and can be implemented using, for example, stratified negation, well-founded negation or the stable model semantics.
- ***f***, which denotes the use of function symbols. The unrestricted use of function symbols makes the language undecidable, wrt. query answering. However, there exist several known restrictions on the use of function symbols, which make the language decidable.
- ¬, which denotes classical negation. This form of negation is not very common in logical programming, although there is work, which shows how to add classical negation to logic programming (e.g. [Gelfond and Lifschitz, 1991]). Classical negation allows to explicitly state that facts are not true and a classically negated atom is only true if its negation can be explicitly derived.

Another extension of Datalog we do not mention explicitly in the remainder of this document is an extension with the symbols *true* and $false$. These symbols are similar to the top ($\top$) and bottom ($\bot$) concepts, respectively, in Description Logics. The use of these symbols requires a minimal additional check:

- The body of a rule with $false$ in the head can be seen as an integrity constraint
- A rule with $false$ in the body or *true* in the head can be eliminated before the computation of the model, because a rule with $false$ in the body may never fire and when a rule with *true* in the body fires, nothing is added to the model.
- Any occurrence of *true* in the body of a rule can be removed.

This document is further structured as follows. We first introduce the notion of constraints, as well as the *unique name assumption* and the local-closed world assumption, in Chapter 2. Then, we describe how the datatype groups approach can be applied to OWL$^-$ in chapter 3. We introduce the abstract syntax and semantics (through a mapping to F-Logic) of OWL Flight in chapter 4. Finally, we present conclusions and future work in chapter 5.

2 Constraints

In this section we present the different types of constraints of OWL Flight, all related to certain aspects of properties, such as cardinality and range. These constraints are related to the notion of *integrity constraints*. An integrity constraint is a rule without a head and is violated if all the literals in the body are true under a certain interpretation. Integrity constraints are directly supported in the Stable Model Semantics (SMS) [Gelfond and Lifschitz, 1988]. When not using SMS, integrity constraints can be implemented by using a special predicate as the head of the rule of which the body corresponds with the integrity constraint. If the extension of the predicate is non-empty after the computation, the constraint is violated.

We first introduce the notions of *unique name assumption* (UNA) and (local) closed-world assumption (LCWA). These notions are required for the value and cardinality constraints, introduced in the rest of this chapter.

2.1 Unique Name Assumption

The Unique Name Assumption can be easily introduced in a Description Logic knowledge base by including an inequality axiom $x \neq y$ for each pair of distinct individuals x and y [Baader et al., 2003, pp. 64, 84]. Similarly, we can express this in OWL using the `DifferentIndividuals(`o_1 `...` o_n`)` statement, enumerating all distinct individuals $o_1 \ldots o_n$. In the case of the UNA, maximal cardinality restrictions in Description Logics behave like constraints, rather than assertions. If an equality between two individuals is derived from a cardinality restriction, this immediately leads to an inconsistency, because the inequality between the pair of individuals has already been asserted.

When translating such a Description Logic knowledge base with the unique name assumption into a logic program, it is often not necessary to translate the inequality assertions, because logic programming engines and deductive databases typically adhere to the Unique Name Assumption. With the UNA syntactically different terms are assumed to be unequal. This means that unification only requires syntactic matching of terms, as opposed to checking satisfiability, which greatly speeds up the proof procedure.

Under the UNA, the semantics of OWL Flight diverge from the original semantics of OWL presented in [Patel-Schneider et al., 2004]. For the OWL Full$^-$ subset of OWL Flight (and also OWL), this is no problem, since all features in OWL, which relied on the absence of the UNA were left out on purpose. In fact, for OWL$^-$ it does not matter whether you adhere to the UNA or not, because there is no (in)equality in the language.

The Unique Name Assumption in OWL Flight OWL Flight adheres to the Unique Name Assumption in the sense that we do not allow deriving equality of terms and we implicitly assume that all individuals in the knowledge base are different (in terms of Description Logics, we assume an assertion $o_i \neq o_j$ for each distinct pair of individual names o_i and o_j). We assume an oracle beyond the ontology language, which resolves any redundancy in the naming of entities on the Semantic Web and normalizes different terms that denote the same resource.

We expect that many applications will not even need such an oracle to

normalize terms, since each application typically operates on a very limited domain. When knowledge outside of the local knowledge base is required, a mediator (cf. [Roman et al., 2004]) is typically required to mediate between the differences in representation and to unify terms.

2.2 Local Closed-World Assumption

The Closed-World Assumption (CWA) is typically applied in the semantics of logic programming (e.g. Prolog) and database applications. When applying the CWA you assume to have complete knowledge of the world, which implies that every fact that cannot be proven to be true is assumed to be false. This assumption can be very useful in order to infer new information from the absence of information. However, when knowledge is incomplete, this assumption might be inappropriate. This could be the case in the Semantic Web, because of its openness and distributiveness. A piece of information not known to an application might still be somewhere on the Semantic Web. This is why most current language for the Semantic Web, including OWL, adhere to the Open World Assumption (OWA). Under the OWA, it is only possible to infer negative information by explicitly stating it or by inferring it using some inference rules. In other words, under the OWA, a ground atomic sentence ϕ only holds when it is a direct consequence of the logical theory formed by the knowledge base KB, which consists of an ontology and a set on instances:

$$KB \models \phi \tag{2.1}$$

Under the CWA, a ground atomic sentence $\neg\phi$ holds when ϕ is *not* a logical consequence:

$$(KB \not\models \phi) \models \neg\phi \tag{2.2}$$

In the Semantic Web setting the OWA might seem the way to go[1], because there can always be more information somewhere on the Web which you don't know about. However, in many cases it is necessary to do some form of closed-world reasoning, because otherwise some problems would become untractable or even unsolvable. We illustrate this with an e-commerce example.

Say, you want to search for the cheapest price for which you can buy a certain product a. After the search, you want the following condition to hold, where *price* is a binary predicate symbol and o is the price returned after the search:

$$\neg\exists y : price(a, y) \wedge price(a, o) \wedge y < o \tag{2.3}$$

Under the OWA, this condition cannot be verified, because it is impossible to guarantee to have included the price from every possible vendor in the search.

Notice also that the minimal cardinality and value constraints, introduced later in this chapter, require closed-world reasoning to check the constraints and thus implicitly apply the local closed-world assumption on the properties for which the constraints are specified

Applying Local Closed-World (LCW) reasoning in the planning domain was investigated in [Etzioni et al., 1997]. [Heflin and Muñoz-Avila, 2002] applies LCW-reasoning to two Semantic Web languages, namely SHOE and DAML+OIL. The latter was the basis for the current Semantic Web ontology language recommendation OWL.

[1] Notice that we have not been able to come up with a real practical example where the OWA makes sense and the CWA does not.

[Etzioni et al., 1997] introduces the operator LCW, which allows to state that an agent has *local closed world information* (i.e. complete information) relative to a logical formula Φ. LCW can be defined as follows:

$$\texttt{LCW}(\Phi) \equiv (KB \models \Phi\theta) \vee (KB \models \neg\Phi\theta) \text{ for all ground substitutions } \theta \quad (2.4)$$

Applying the operator LCW to our example in the following way: LCW(*price*), the stated condition can be verified to hold with respect to the local knowledge base[2]. It is now up to the application using the knowledge base to make sure enough relevant knowledge about *price* is present in the knowledge base.

The Local-Closed-World Assumption in OWL Flight The local closed-world assumption can be used in queries or definitions to specify complete knowledge for a certain predicate, which could be a class or a property.

We can now benefit from local closed-world reasoning by allowing negation of classes and properties, which have been defined under the local closed-world assumption. This negation can now be treated as default negation, where it is up to the implementation to decide which type of negation to use (e.g. stratified negation, well-founded negation or negation under stable model semantics).

Note that, as described in the following, we allow default negation in cardinality constraints and value constraints, where we implicitly assume complete knowledge for the properties in order to check the constraints.

2.3 Types of constraints

We distinguish three types of constraints, namely minimal cardinality constraints, maximal cardinality constraints and value (range) constraints.

Minimal cardinality constraints The specification of a minimal cardinality constraint of arity 1 requires Datalog(IC,not). We illustrate the minimal cardinality constraint of property R at class C, which is expressed as the following integrity constraint:

$$\leftarrow not\ R(?x, ?y) \wedge C(?x)$$

A minimal cardinality constraint of arity higher than 1 additionally requires the use of inequality in the language and thus requires Datalog(IC,not,$\neq$). We illustrate a minimal cardinality of 2 for the property R at class C (ok is a newly introduced predicate, which does not occur anywhere else in the database):

$$ok(?x) \leftarrow R(?x, ?y) \wedge R(?x, ?z) \wedge ?y \neq ?z \wedge C(?x)$$
$$\leftarrow C(?x) \wedge not\ ok(x)$$

The first rule can be easily extended for a minimal cardinality of arity higher than 2. Notice however, that when the extending the rule for minimal cardinality of higher arity, the size of the rule will grow quadratically. Most Logic Programming systems (e.g. DLV[3]), which support minimal cardinality, use

[2] Note that we do not mean here necessarily all knowledge local to the agent, but rather all knowledge accessible to the agent at the point in time of evaluation of the query.

[3] http://www.dbai.tuwien.ac.at/proj/dlv/

different methods beyond the logical language to detect violation of minimal cardinality, such as aggregate functions.

Note that by adopting this kind of minimal cardinality constraints, we diverge from the first-order style semantics of OWL. In our approach, the constraint is violated if we do not know about any property filler for the property R. Under the open world assumption of OWL, there is no violation, because you never know whether there might be a property filler somewhere on the Semantic Web.

We argue that the constraint approach does not pose a problem, because, intuitively, when the designer of the ontology models a minimal cardinality constraint, the designer expects that this constraint is *violated* if no property filler is known to exist, rather than a property filler being created during reasoning, which satisfies this constraint. In other words, in order to check minimal cardinality constraints, we apply a form of closed-world reasoning.

Note that we do not diverge from the OWL$^-$ semantics, since minimal cardinality is not in OWL$^-$. Notice that the `minCardinality(1)` restriction (on the left-hand side of the GCI) is in OWL$^-$. As we can see later on (in Chapter 4), this does not pose a problem, because minimal cardinality constraints are only allowed on the right-hand side.

Maximal cardinality constraints In order to specify maximal cardinality constraints (of any arity), the language requires integrity constraints and inequality, thus Datalog(IC,$\neq$) suffices. As an example we show how a maximal cardinality constraint of 1 for the property R at class C is expressed:

$$\leftarrow R(?x, ?y) \wedge R(?x, ?z) \wedge C(?x) \wedge ?y \neq ?z$$

This constraint can be trivially extended for maximal cardinality constraints of higher arity. However, just as for minimal cardinality constraints, this leads to a blow-up of the size of the rule. Therefore, existing systems (e.g. DLV) use aggregate functions beyond the logic programming language to deal with this type of constraints more efficiently.

Notice that we stay inside the first-order semantics of OWL, albeit that we assume inequality assertions between each pair of distinct individuals.

Value constraints OWL (and also OWL$^-$) has universal value restrictions, which are actually assertions, rather than constraints (see Section 1.1). For example, restricting the range of property R to class D at class C is written down (in first-order logic) in the following way:

$$D(?y) \leftarrow R(?x, ?y) \wedge C(?x)$$

Such an assertion would entail the fact that an instance is a member of D if it is in the range of R. However, we argue that it is more useful to *check* whether an instance in the range of R is actually a member of D. This can be done in Logic Programming using the following integrity constraint:

$$\leftarrow R(?x, ?y) \wedge C(?x) \wedge \text{not } D(?y)$$

As we can see, we need both integrity constraints and default negation, thus Datalog(IC,not).

As we can see from the use of default negation for these value constraints, these constraints bring us outside of the first-order style semantics of OWL. A form of non-monotonicity is introduced, because if we do not know that a certain

value for a property is a member of D, the constraint is violated. However, if we later learn that this particular value is a member of D, the constraint is no longer violated, thus new information can invalidate existing inferences.

A more important issue is how to integrate value constraints into OWL Flight. Because the OWL$^-$ semantics already guarantee that the value for the property is a member of D, simply adding the constraint to the knowledge base has no effect, since it will never be violated.

A solution to this problem is the introduction of a new type of value restrictions, alongside the existing value restrictions. We call the value restrictions coming from OWL Lite$^-$ *assertive* value restrictions. The value restrictions we introduced here are called *constraining* value restrictions. The designer of the ontology can choose the kind of value restrictions he/she wants to use. Notice that when both an assertive and a constraining value restriction (assumed they both specify the same range) are specified for a certain property at a certain class, this amounts to an assertive restriction, since the constraint is then never violated. Because it does not make sense the model the same restriction both assertive and constraining, the ontology engineering tool should disallow this.

This approach guarantees a clean semantic layering on top of OWL$^-$, since the semantics of the value restrictions introduced in an OWL$^-$ ontology do not change in OWL Flight. Thus, for an OWL$^-$ ontology, the same conclusions will be drawn by both an OWL$^-$ and an OWL Flight reasoner. However, if the OWL$^-$ ontology is extended with this type of value constraints, existing inferences might be invalidated because of the non-monotonicity of this feature.

Existential Value Constraints OWL makes the distinction between existential and universal value restrictions. In the former, when we discussed value restrictions in OWL, we actually meant the *universal* value restrictions. The term *existential* value restriction is in our opinion misleading, because it is in fact a (qualified) number restriction (assertion) of one. The restriction in fact asserts that there is at least one property filler with a specific class as its range. Qualified number restrictions, allowed in popular Description Logics such as $\mathcal{SHIQ}$, are not allowed in OWL, even though they do not add complexity over unqualified number restrictions, which are allowed in OWL.

In order to stay compatible with the OWL abstract syntax, we do allow to state existential value *constraints*, which are simply interpreted as minimal cardinality constraints of one. Notice that the semantics of minimal cardinality constraints is different from the semantics of restriction in OWL.

3 Adding Datatype Support to OWL$^-$

OWL$^-$ [de Bruijn et al., 2004b] does not have support for datatypes. However, it was already identified as a possible extension for the language. In fact, we believe that any practical application will need support for datatypes.

There currently exists a gap between the way data types are handled in OWL and the treatment of concrete domains in Description Logics. The latter only allows one concrete domain (e.g. *integer*), whereas the former allows many different data types (e.g. *string*, *date*, *integer*), albeit with many limitations. [Pan and Horrocks, 2004] shows a way to bridge the gap between the two, while extending datatype support in OWL, in order to allow the full expressiveness of concrete domains in Description Logics, while using different data types and retaining decidability.

The major limitations of the datatype support in OWL are the lack of support for datatype predicates and the lack of user-defined data types. We believe that both facilities are very important for practical applications. Therefore, in this chapter, we extend OWL$^-$ with a richer datatype support than OWL. We include support for datatype predicates and user-defined datatypes, following the datatype group approach of Pan and Horrocks [Pan and Horrocks, 2003; Pan and Horrocks, 2004]. Unfortunately, OWL$^-$ with the datatype group extension is no longer a strict subset of OWL. However, in order to make the language more useable, this extension is required in our opinion and it provides the first step in the direction of OWL Flight.

Because we use the datatype extension described in [Pan and Horrocks, 2004], we stay compliant with the way of handling datatypes in Description Logics. In fact, [Pan and Horrocks, 2004] describes the exact relationship between datatype (sub-)groups and corresponding concrete domains, as they have been studied in the Description Logic literature (e.g. [Baader and Hanschke, 1991; Horrocks and Sattler, 2001]).

We first describe the abstract syntax of the datatype properties and the datatype expressions, as an extension of the OWL$^-$ abstract syntax. Then, we describe the semantics of the datatype support in OWL$^-$. Because OWL$^-$ has been carefully constructed to fall within the intersection of Description Logics and Datalog, it is important to see how datatypes can be handled in Logic Programming.

Many datalog implementations have support for concrete values built in, although there is typically no strong support for data typing. Most implementations, however, do provide a large number of built-in predicates[1]. One could view these built-in predicates as an oracle, since they are outside of the logical framework of the logic programming formalism. Many datalog engines therefore already provide a limited datatype oracle. Constraint Logic Programming (CLP) [Jaffar and Maher, 1994; Jaffar et al., 1998; Marriott and Stuckey, 1998] provides a way to work with datatypes in logic programming. In fact, CLP can be used to program the required functionality, such as datatype predicates, for working with datatypes. A constraint solver can be used as datatype oracle and could even be used together with a Description Logic reasoner. The Constraint Handling Rules (CHR) [Frühwirth, 1998] approach generalizes CLP and allows for, for example, temporal and spatial reasoning.

[1] e.g. http://www.ontoprise.de/documents/tutorial_flogic.pdf, Section 6

3.1 Datatype Abstract Syntax

Table 3.1 summarizes which feature in the OWL-E abstract syntax we support in the datatype extension of OWL$^-$. In the table, T_i stand for a unary datatype predicate, which usually corresponds to a built-in datatype. U_i stands for a datatype property. P stands for a `DatatypeExpression` ID. Note that the set of URIs for datatype expressions is disjoint from the sets of URIs for classes, properties, individuals and (datatype) predicates. *deCom* stands for a datatype expression component, which is either a datatype predicate (ID), `domain()`, `and()` or `or()`.

Notice that we do not allow the `or()` and `oneOf()` constructors for datatype expressions, because they cannot be expressed in Datalog.

Notice that we diverge from the syntax used for restrictions on datatype properties in [Pan and Horrocks, 2004]. We do not use the keywords `someTuplesSatisfy` and `allTuplesSatisfy`, but rather the OWL keywords `someValuesFrom` and `allValuesFrom`. We also do not allow qualified number restrictions on datatype properties, which is allowed in OWL-E, but not in OWL.

Notice that when observing the following restrictions, the ontology stays inside OWL:

- Not using datatype predicates of arity higher than 1 in property restrictions. Furthermore, the unary predicate ID must corresponds with a datatype ID.
- Not creating user-defined datatypes using the `DatatypeExpression` keyword.

3.2 Datatype Semantics

A datatype group is essentially a group of disjoint datatypes, where each (disjoint) datatype forms a subgroup. A datatype group $\mathcal{G}$ consists of a predicate map M_p, which maps predicate URI references to predicates, a set of predicate URI references $D_\mathcal{G}$ and a domain function dom. A subgroup intuitively consists of all predicates defining relations on one particular datatype (e.g. *integer*). A subgroup is identified with a predicate URI, which is interpreted as a unary predicate with the entire datatype as its extension (e.g. `xsd:integer` corresponds to the datatype *integer*). Finally, [Pan and Horrocks, 2004] introduces a number of so-called $\mathcal{G}$-Datatype expressions, with which the user can construct new data types and new predicates, derived from the existing datatypes and predicates

3.3 Transforming Datatype Expressions to Datalog(IC)

We describe the translation of OWL DL$^-$ datatype properties and expressions to Datalog(IC) (Datalog with integrity constraints). The translation for OWL Full$^-$ is similar and will be provided in a future version of this document. Notice that a datatypeexpression ID is just a syntactic shortcut for the expression and in the translation, the ID is substituted with the expression (denoted

OWL Abstract Syntax	**DL syntax**	**OWL$^-$**
Datatype Expressions (P)		
$deCom$ (datatype predicate)	$deCom$	+
`DatatypeExpression(`$P\ deCom$`)`		+
`domain(`$T_1\ \ldots\ T_n$`)`	$(T_1, \ldots, T_n)$	+
`and(`$deCom_1\ \ldots\ deCom_n$`)`	$deCom_1 \sqcap \ldots \sqcap deCom_n$	+
`or(`$deCom_1\ \ldots\ deCom_n$`)`	$deCom_1 \sqcup \ldots \sqcup deCom_n$	-
`oneOf(`$v_1\ \ldots\ v_n$`)`	$\{v_1, \ldots, v_n\}$	-
Datatype Properties (U)		
`DatatypeProperty(`U `super(`U_1`)...super(`U_n`)`	$U \sqsubseteq U_i$	+
`domain(`C_1`) ... domain(`C_n`)`	$\top \sqsubseteq \forall U^-.C_i$	+ $(C_i \neq \bot)$
`range(`T_1`) ... range(`T_n`)`	$\top \sqsubseteq \forall U.T_i$	+
`[Functional])`	$\top \sqsubseteq \leqslant 1R$	–
`SubPropertyOf(`$U_1\ U_2$`)`	$U_1 \sqsubseteq U_2$	+
`EquivalentProperties(`$U_1\ \ldots\ U_n$`)`	$U_1 \equiv \ldots \equiv U_n$	+
`Individual(value(`$U_1\ v_1$`) ... value(`$U_n\ v_n$`))`	$\langle o, v_i \rangle \in U_i$	+
Data Ranges (B)		
B (datatype name)	B	+
Descriptions (D)		
`restriction(`$U_1 \ldots U_n$ `someValuesFrom(`P`))`	$\exists U_1, \ldots, U_n.P$	lhs*
`restriction(`$U_1 \ldots U_n$ `allValuesFrom(`P`))`	$\forall U_1, \ldots, U_n.P$	rhs**
`restriction(`U `value(`v`))`	$\exists U.\{v\}$	++
`restriction(`U `minCardinality(1))`	$\geqslant 0U$	rhs**
`restriction(`U `minCardinality(`n`))`	$\geqslant 0U$	–
`restriction(`U `maxCardinality(`n`))`	$\leqslant 0U$	–
* May only be used on the left-hand side (as the first argument) of `SubClassOf` ** May only be used in partial class definitions and on the right-hand side (as the second argument) of `SubClassOf`		

Table 3.1: Datatype support in OWL$^-$

OWL DL$^-$ Abstract Syntax	**Datalog**
Datatype Properties (U)	
$\phi_{\mathcal{LP}}$`(DatatypeProperty(`U `super(`U_1`)...super(`U_n`))`	$U(x,y) \rightarrow \bigwedge U_i(x,y)$
$\phi_{\mathcal{LP}}$`( domain(`C_1`) ... domain(`C_n`))`	$U(x,y) \rightarrow \bigwedge \phi_{\mathcal{LP}}(C_i, x)$
$\phi_{\mathcal{LP}}$`( range(`T_1`) ... range(`T_n`)))`	$\leftarrow U(x,y) \wedge \bigwedge T_i(y)$
$\phi_{\mathcal{LP}}$`(SubPropertyOf(`$U_1\ U_2$`))`	$U_1(x,y) \rightarrow U_2(x,y)$
$\phi_{\mathcal{LP}}$`(EquivalentProperties(`$U_1\ \ldots\ U_n$`))`	$\begin{cases} U_i(x,y) \rightarrow U_j(x,y) \\ U_j(x,y) \rightarrow U_i(x,y) \end{cases}$
$\phi_{\mathcal{LP}}$`(Individual(`o `value(`$U_1\ v_1$`) ... value(`$U_n\ v_n$`)))`	$\bigwedge U_i(o, v_i)$
Datatype expressions	
$\phi_{\mathcal{LP}}(p, Y_1, \ldots, Y_k)$ (datatype predicate with arity k)	$p(Y_1, \ldots, Y_k)$
$\phi_{\mathcal{LP}}$`(DatatypeExpression(`$P\ deCom$`),`$Y_1, \ldots, Y_k$`)`	$P := \phi_{\mathcal{LP}}(deCom, Y_1, \ldots, Y_k)$
$\phi_{\mathcal{LP}}$`( domain(`$T_1\ \ldots\ T_k$`),`$Y_1, \ldots, Y_k$`)`	$\bigwedge T_i(Y_i)$
$\phi_{\mathcal{LP}}$`( and(`$deCom_1\ \ldots\ deCom_n, Y_1, \ldots, Y_k$`)`	$\bigwedge \phi_{\mathcal{LP}}(deCom_i, Y_1, \ldots, Y_k)$
Descriptions (D)	
$\phi_{\mathcal{LP}}$`(restriction(`$U_1 \ldots U_k$ `someValuesFrom(`P`)),`X`)`	$(\bigwedge U_i(X, y_i)) \wedge \phi_{\mathcal{LP}}(P, y_1, \ldots, y_k)$
$\phi_{\mathcal{LP}}$`(restriction(`$U_1 \ldots U_k$ `allValuesFrom(`P`)),`X`)`	$\leftarrow \phi_{\mathcal{LP}}(P, y_1, \ldots, y_k) \wedge \bigwedge U_i(X, y_i)$
$\phi_{\mathcal{LP}}$`(restriction(`U `value(`v`)),`X`)`	$U(X, v)$

Table 3.2: Translating OWL DL$^-$ datatype properties and expressions into Datalog

by the ':=' symbol). An integrity constraint is denoted by a left arrow (←) followed by a conjunction of positive literals.

There are several restrictions on the datatype expressions. Each component in an `and()` expression needs to have the same arity. Furthermore, the variables $y_1, \ldots, y_k$ occurring in the datatype expressions and the universal value restriction refer to the same variables and these variables should not be renamed during the translation. The arity of the restriction (the number of properties participating in the restriction) has to match the arity of the datatype expression. A simple datatype reference is interpreted as a unary predicate and thus has arity 1.

Notice that many Datalog implementations have built-in support for several datatype predicates. String and integer operations are quite common for Logic Programming engines, because it was already recognized to be an important issue for practical applications. Because these built-in predicates are not part of the logic programming language, they can be seen as beyond the language and thus part of a datatype oracle. Notice that the oracle is not necessarily an external application and for efficiency reasons it is usually preferable to implement the datatype predicates in the Datalog engine itself.

4 OWL Flight

In this chapter, we present OWL Flight, which extends OWL$^-$ with a number of features, such as constraints and local-closed world assumption.

4.1 Constructing OWL Flight

The underlying formalism for OWL Flight is Datalog($IC, \neq, not$), which is Datalog extended with integrity constraints, inequality and default negation. These features are required to express various kinds of constraints, see Chapter 2, and other interesting features. See Table 4.1 for an overview of OWL DL constructs, which are supported in OWL Flight. This list is slightly bigger than for OWL DL$^-$, because several features of OWL DL not expressible in pure Datalog are expressible in Datalog($\neq, IC, not$).

As can be seen from Table 4.1, the only real features added from OWL DL compared with OWL DL$^-$ is support for the top ($\top$) and the bottom ($\bot$) concepts, which account also for allowing the `DisjointClasses` construct.

Besides the features coming from OWL DL (Table 4.1), we add the following features in OWL Flight:

Minimal Cardinality Constraints

Maximal Cardinality Constraints

Property Value Constraints

Local Closed World Assumption

These constraints were described in more detail in Chapter 2.

The underlying formalism for OWL Flight is Datalog($\neq$,IC,not). However, we use F-Logic [Kifer et al., 1995] (we denote the combination as F-Logic(Datalog($\neq$,IC,not))) syntactic sugar in order to make the semantics more intuitive and in order to facilitate future extension of the language.

4.2 OWL Flight Abstract Syntax

We give here the OWL Flight Abstract Syntax (an extension of the OWL Full$^-$ syntax) in the style of [Patel-Schneider et al., 2004].

A concrete syntax for OWL Flight, based on WSMO/WSML will be provided in WSML D16.10: WSML-Flight.

A symbol between square brackets ([]) is not required to occur (may occur 0 or 1 time). A symbol between curly brackets ({}) may occur zero or more times. A symbol not enclosed in square or curly brackets is required and may only occur one time. The bar (|) stands for an exclusive choice. All keywords are represented in boldface.

An ontology in OWL Flight consists of a number of so-called directives, which can either be annotations, axioms or facts.

OWL Abstract Syntax	**DL syntax**	**OWL Flight**
Axioms		
`Class(`A `partial` C_1 `...` C_n`)`	$A \sqsubseteq C_i$	+
`Class(`A `complete` C_1 `...` C_n`)`	$A \equiv C_1 \sqcap \ldots \sqcap C_n$	+
`EnumeratedClass(`A o_1 `...` o_n`)`	$A \equiv \{o_1, \ldots o_n\}$	–
`SubClassOf(`C_1 C_2`)`	$C_1 \sqsubseteq C_2$	+
`EquivalentClasses(`C_1 `...` C_n`)`	$C_1 \equiv \ldots \equiv C_n$	+
`DisjointClasses(`C_1 `...` C_n`)`	$C_i \sqcap C_j \sqsubseteq \bot$	+
`ObjectProperty(`R `super(`R_1`)...super(`R_n`)`	$R \sqsubseteq R_i$	+
`domain(`C_1`)` `...` `domain(`C_n`)`	$\top \sqsubseteq \forall R^-.C_i$	+
`range(`C_1`)` `...` `range(`C_n`)`	$\top \sqsubseteq \forall R.C_i$	+
`[inverseOf(`R_0`)]`	$R \equiv ({}^-R_0)$	+
`[Symmetric]`	$R \equiv ({}^-R)$	+
`[Functional]`	$\top \sqsubseteq \leqslant 1R$	+
`[InverseFunctional]`	$\top \sqsubseteq \leqslant 1R^-$	+
`[Transitive])`	Trans(R)	+
`SubPropertyOf(`R_1 R_2`)`	$R_1 \sqsubseteq R_2$	+
`EquivalentProperties(`R_1 `...` R_n`)`	$R_1 \equiv \ldots \equiv R_n$	+
`Individual(`o `type(`C_1`)` `...` `type(`C_n`)`	$o \in C_i$	+
`value(`R_1 o_1`)` `...` `value(`R_n o_n`))`	$\langle o, o_i \rangle \in R_i$	+
`SameIndividual(`o_1 `...` o_n`)`	$o_1 = \ldots = o_n$	–
`DifferentIndividuals(`o_1 `...` o_n`)`	$o_i \neq o_j, i \neq j$	– ***
Descriptions (C)		
A (URI Reference)	A	+
`owl:Thing`	$\top$	+
`owl:Nothing`	$\bot$	+
`intersectionOf(`C_1 `...` C_n`)`	$C_1 \sqcap \ldots \sqcap C_n$	+
`unionOf(`C_1 `...` C_n`)`	$C_1 \sqcup \ldots \sqcup C_n$	lhs*
`complementOf(`C_0`)`	$\neg C_0$	–
`oneOf(`o_1 `...` o_n`)`	$\{o_1, \ldots o_n\}$	lhs*
`restriction(`R `someValuesFrom(`C`))`	$\exists R.D$	lhs*
`restriction(`R `allValuesFrom(`C`))`	$\forall R.D$	rhs**
`restriction(`R `value(`o`))`	$\exists R.o$	+
`restriction(`R `minCardinality(`n`))`	$\geqslant nR$	–
`restriction(`R `maxCardinality(`n`))`	$\leqslant nR$	rhs**
* May only be used on the left-hand side (as the first argument) of `SubClassOf` ** May only be used in partial class definitions and on the right-hand side (as the second argument) of `SubClassOf` *** Notice that already all individuals in an OWL Flight knowledge base are different; therefore, an explicit assertion would be superfluous		

Table 4.1: Features of OWL DL present in OWL Flight

ontology ::= 'Ontology(' [**ontologyID**] { **directive** } ')'
directive ::= 'Annotation(' **ontologyPropertyID ontologyID** ')'
| 'Annotation(' **annotationPropertyID URIreference** ')'
| 'Annotation(' **annotationPropertyID dataLiteral** ')'
| 'Annotation(' **annotationPropertyID individual** ')'
| **axiom**
| **fact**

These are the different types of identifiers in OWL Flight. Notice that there are a few differences, compared with the OWL Abstract Syntax (AS). First of all, datatypeExpressionID is added and datatypeID is renamed to unaryDatatypePredicateID, because we use the OWL-E extension for datatypes [Pan and Horrocks, 2004]. Secondly, classID and individualID are replaced with objectID to make clear that we do not make a clear separation between classes and individuals.

datatypeExpressionID ::= **URIreference**
unaryDatatypePredicateID ::= **URIreference**
objectID ::= **URIreference**
ontologyID ::= **URIreference**
datavaluedPropertyID ::= **URIreference**
individualvaluedPropertyID ::= **URIreference**
annotationPropertyID ::= **URIreference**
ontologyPropertyID ::= **URIreference**

The non-functional properties of all the elements in the ontology are described using so-called annotations. All axioms and facts in the ontology have the possibility to attach annotations.

annotation ::= 'annotation(' **annotationPropertyID URIreference** ')'
| 'annotation(' **annotationPropertyID dataLiteral** ')'
| 'annotation(' **annotationPropertyID individual** ')'

Facts consist of information about individuals. Notice that we do not allow the assertion of individual (in)equality, because we do not have equality in the underlying formalism and an OWL Flight ontology adheres to the unique name assumption.

fact ::= **individual**
individual ::= 'Individual(' [**objectID**] { **annotation** } { 'type(' **type** ')' } { **value** } ')'
value ::= 'value(' **individualvaluedPropertyID objectID** ')'
| 'value(' **individualvaluedPropertyID individual** ')'
| 'value(' **datavaluedPropertyID dataLiteral** ')'
type ::= **description**

These are the literals we allow.

dataLiteral ::= **typedLiteral** | **plainLiteral**
typedLiteral ::= **lexicalForm^^URIreference**
plainLiteral ::= **lexicalForm** | **lexicalForm@languageTag**
lexicalForm ::= as in RDF, a unicode string in normal form C
languageTag ::= as in RDF, an XML language tag

We allow different types of class axioms. Our class axioms diverge somewhat from the OWL AS class axioms, because we distinguish between partial/complete class descriptions and because we distinguish between descriptions allowed on the left/right-hand side of the GCI (General Class Inclusion, `SubClassOf`).

Notice also that we do not allow enumerated classes.

axiom ::= 'Class(' **objectID** ['Deprecated'] 'partial' { **annotation** } { **rhs_description** } ')'
axiom ::= 'Class(' **objectID** ['Deprecated'] 'complete' { **annotation** } { **lhs_description** } ')'
axiom ::= 'DisjointClasses(' **lhs_description lhs_description** { **lhs_description** } ')'
| 'EquivalentClasses(' **description** { **description** } ')'
| 'SubClassOf(' **lhs_description rhs_description** ')'

We can designate a **unaryDatatypePredicateID** as datatype. Furthermore, user-defined datatypeexpressions (corresponding to user-defined datatypes or predicates) can be defined in the way defined by [Pan and Horrocks, 2004]. However, we omit the `or` and `oneOf` constructs.

axiom ::= 'Datatype(' **unaryDatatypePredicateID** ['Deprecated'] { **annotation** })'
axiom ::= 'DatatypeExpression(' **datatypeExpressionID** ['Deprecated'] **deComponent** { **annotation** })'
deComponent ::= **datatypePredicateID**
| 'domain(' { **unaryDatatypePredicateID** } ')'
| 'and(' **deComponent** ')'

A description is allowed to occur both on the left- and the right-hand side of the GCI. A lhs_description can only occur on the left-hand side; a rhs_description can only occur on the right-hand side.

Notice that for cardinality, we only allow the maximal cardinality construct on the right-hand side and a minimal cardinality of one on the left-hand side.

description ::= **objectID**
| **restriction**
| 'intersectionOf(' { **description** } ')'
lhs_description ::= **description**
| **lhs_restriction**
| 'unionOf(' { **lhs_description** } ')'
| 'oneOf(' { **objectID** } ')'
rhs_description ::= **description**
| **rhs_restriction**
restriction ::= 'restriction(' { **datavaluedPropertyID** } **dataRestrictionComponent**
{ **dataRestrictionComponent** } ')'
| 'restriction(' **individualvaluedPropertyID individualRestrictionComponent**
{ **individualRestrictionComponent** } ')'
dataRestrictionComponent ::= 'value(' **dataLiteral** ')'
individualRestrictionComponent ::= 'value(' **objectID** ')'
lhs_restriction ::= **restriction**
| 'restriction(' { **datavaluedPropertyID** } **lhs_dataRestrictionComponent**
{ **lhs_dataRestrictionComponent** } ')'
| 'restriction(' **individualvaluedPropertyID**
lhs_individualRestrictionComponent
{ **lhs_individualRestrictionComponent** } ')'
lhs_dataRestrictionComponent ::= 'someValuesFrom(' **dataRange** ')'
| 'minCardinality(1)'
lhs_individualRestrictionComponent ::= 'someValuesFrom(' **lhs_description** ')'

| 'minCardinality(1)'
rhs_restriction ::= **restriction**
| 'restriction(' { **datavaluedPropertyID** } **rhs_dataRestrictionComponent**
{ **rhs_dataRestrictionComponent** } ')'
| 'restriction(' **individualvaluedPropertyID rhs_individualRestrictionComponent**
{ **rhs_individualRestrictionComponent** } ')'
rhs_dataRestrictionComponent ::= 'allValuesFrom(' **dataRange** ')'
| **cardinality**
rhs_individualRestrictionComponent ::= 'allValuesFrom(' **rhs_description** ')'
| **cardinality**
cardinality ::= 'maxCardinality(' **non-negative-integer** ')'

Because of the extended datatype support, the dataRange is also a bit more elaborate.

dataRange ::= **unaryDatatypePredicateID** | 'rdfs:Literal'
| **datatypeExpressionID** | **deComponent**

We allow, in contrary to OWL$^-$, functional properties. This is because functional properties can, just as maximal cardinality, be expressed in Datalog(IC,$\neq$), when under the unique name assumption. And since we assume an assertion $o_i \neq o_j$ in the knowledge base for all distinct pairs of individuals o_i and o_j, the languages adheres to the unique name assumption.

axiom ::= 'DatatypeProperty(' **datavaluedPropertyID** ['Deprecated'] { **annotation** }
{ 'super(' **datavaluedPropertyID** ')'} ['Functional']
{ 'domain(' **description** ')' } { 'range(' **dataRange** ')' } ')'
| 'ObjectProperty(' **individualvaluedPropertyID** ['Deprecated'] { **annotation** }
{ 'super(' **individualvaluedPropertyID** ')' }
['inverseOf(' **individualvaluedPropertyID** ')'] ['Symmetric']
['Functional' | 'InverseFunctional' | 'Functional' 'InverseFunctional' | 'Transitive']
{ 'domain(' **description** ')' } { 'range(' **description** ')' } ')'
| 'AnnotationProperty(' **annotationPropertyID** { **annotation** } ')'
| 'OntologyProperty(' **ontologyPropertyID** { **annotation** } ')'
axiom ::= 'EquivalentProperties(' **datavaluedPropertyID datavaluedPropertyID** { **datavaluedPropertyID** } ')'
| 'SubPropertyOf(' **datavaluedPropertyID datavaluedPropertyID** ')'
| 'EquivalentProperties(' **individualvaluedPropertyID individualvaluedPropertyID**
{ **individualvaluedPropertyID** } ')'
| 'SubPropertyOf(' **individualvaluedPropertyID individualvaluedPropertyID** ')'

Until so far, the abstract syntax we have presented is merely a restriction of the OWL-E abstract syntax (OWL DL with the datatype groups extension) [Pan and Horrocks, 2004]. In order to capture all the features in OWL Flight, we define the following additions to the abstract syntax:

We define a notion of constraints:

rhs_description ::= **constraint**

constraint ::= 'constraint(' { **datavaluedPropertyID** } **dataConstraintComponent**
{ **dataConstraintComponent** } ')'

```
                | 'constraint(' individualvaluedPropertyID individualConstraintComponent
                              { individualConstraintComponent } ')'
dataConstraintComponent ::= rhs_dataRestrictionComponent
                          | 'minCardinality(' non-negative-integer ')'
                          | 'cardinality(' non-negative-integer ')'
                          | 'someValuesFrom(' dataRange ')'
individualConstraintComponent ::= rhs_individualRestrictionComponent
                                | 'minCardinality(' non-negative-integer ')'
                                | 'cardinality(' non-negative-integer ')'
                                | 'someValuesFrom(' description ')'
```

We also define the complement for descriptions with local-closed world assumption. Of course, this is only allowed to occur on the left-hand side, because default negation is only allowed in the body of a rule. Note that negation is only allowed if each component in the negation is specified, through the use of the lcw keyword, to be complete wrt. the knowledge in the local knowledge base.

```
lhs_description ::= 'lcw(' lhs_description ')'
lhs_description ::= 'complementOf(' objectID | restriction | lhs_restriction ')'
```

4.3 Translation to F-Logic(Datalog(IC,≠,not))

We have already provided a translation of OWL Full$^-$ to F-Logic(Datalog) in D20.1 [de Bruijn et al., 2004b]. Because OWL Full$^-$ is a strict subset of OWL Flight, we will extend this translation to provide a complete translation of the OWL Flight abstract syntax to F-Logic(Datalog(IC,≠,not)), which is the Horn fragment[1] of F-Logic, extended with a notion of integrity constraints, the inequality symbol and default negation.

Notice that because the semantics of OWL Flight is completely defined through the translation to F-Logic(Datalog(IC,≠,not)), an OWL Flight ontology can be straightforwardly extended with rules, without hurting any of the computational properties of the language.

4.4 Layering issues

OWL Flight is layered on top of OWL Full$^-$ in the following way:
Say, we have an OWL Full$^-$ knowledge base KB. Then for each sentence Φ, which is entailed by KB:

$$KB \models \Phi \text{ under OWL Full}^- \text{ semantics}$$

holds:

$$KB \models \Phi \text{ under OWL Flight semantics}$$

However, extending KB with OWL Flight statements it is not guaranteed that the above holds, because of the non-monotonicity of some of the features of OWL Flight (e.g. minimal cardinality and value constraints).

[1] With Datalog restrictions, i.e. allowing only *safe rules*, i.e. only variable occurring in the body are allowed to occur in the head and disallowing the use of function symbols.

OWL Flight Abstract Syntax	**F-Logic**
Mapping OWL Flight Class Axioms to F-Logic	
`Class(`A `partial` C_1 ... C_n`)`	$tr(?x_1 : A, \texttt{intersectionOf}(C_1 \ \ldots \ C_n), ?x_1)$
`Class(`A `complete` C_1 ... C_n`)`	$\left\{ \begin{array}{l} tr(?x_1 : A, \texttt{intersectionOf}(C_1 \ \ldots \ C_n), ?x_1) \wedge \\ tr(\bigwedge tr_{lhs}(C_i, ?x_1), A, ?x_1) \end{array} \right.$
`DisjointClasses(`C_1 ... C_n`)`	$\bigwedge(\neg tr_l hs(C_i, X) \vee \neg tr_l hs(C_n, X))$ ****
`EquivalentClasses(`C_1 ... C_n`)`	$\bigwedge tr(tr_{lhs}C_i, ?x_1), C_j, ?x_1)$
`SubClassOf(`C D`)`	$tr(tr_{lhs}(C, ?x_1), D, ?x_1)$
Mapping OWL Flight Descriptions to F-Logic	
$tr_{lhs}(A, X)$	$X : A$
$tr_{lhs}(\texttt{intersectionOf}(C_1 \ \ldots \ C_n), X)$	$\bigwedge(tr_{lhs}(C_i), X)$
$tr_{lhs}(\texttt{unionOf}(C_1 \ \ldots \ C_n), X)$	$\bigvee(tr_{lhs}(C_i), X)$ *
$tr_{lhs}(\texttt{complementOf}(\texttt{lcw}(C)), X)$	$not\ (tr_{lhs}(\mathrm{C}, X))$
$tr_{lhs}(\texttt{oneOf}(o_1 \ \ldots \ o_n), X)$	$\bigvee X \sim o_i$
$tr_{lhs}(\texttt{restriction}(R\ \texttt{value}(o)), X)$	$X[R \twoheadrightarrow o]$
$tr_{lhs}(\texttt{restriction}(R\ \texttt{someValuesFrom}\ C), X)$	$X[R \twoheadrightarrow ?x_{newIndex}] \wedge tr_{lhs}(C, ?x_{newIndex})$ **
$tr_{lhs}(\texttt{restriction}(R\ \texttt{allValuesFrom}\ C), X)$	$X[R \twoheadrightarrow ?x_{newIndex}] \rightarrow tr_{lhs}(C, ?x_{newIndex})$ *
$tr_{lhs}(\texttt{restriction}(R\ \texttt{minCardinality(1)}), X)$	$X[R \twoheadrightarrow ?x_{newIndex}]$ **
$tr(Expr, \texttt{intersectionOf}(C_1 \ \ldots \ C_n), X)$	$\bigwedge tr(Expr, C_i, X)$
$tr(X : A_1, A_2, X)$	$A_1 :: A_2$
$tr(Expr, A, X)$	$\forall Expr \rightarrow X : A$ ***
$tr(Expr, \texttt{restriction}(R\ \texttt{value}(o)), X)$	$\forall Expr \rightarrow X[R \twoheadrightarrow o]$ ***
$tr(Expr, \texttt{restriction}(R\ \texttt{allValuesFrom}\ C), X)$	$tr(Expr \wedge ?x_i[R \twoheadrightarrow ?x_{newIndex}], C, ?x_{newIndex})$ **
$tr(Expr, \texttt{restriction}(R\ \texttt{maxCardinality}\ n), X)$	$\neg(tr_{lhs}(Expr, X)) \vee \bigvee_{i=1}^{n} \neg R(X, y_i) \vee \bigvee_{i=1}^{n} \bigvee_{j=1}^{n} \neg y_i \neq y_j$ ****
$tr(Expr, \texttt{constraint}(R\ \texttt{minCardinality}\ n), X)$	$(\neg(tr_{lhs}(Expr, X)) \vee \neg not\ ok_{newindex}) \wedge$ $(ok_{newindex}(X) \vee \neg(tr_{lhs}(Expr, X)) \vee \bigvee_{i=1}^{n} \neg R(X, y_i) \vee$ $\bigvee_{i=1}^{n} \bigvee_{j=1}^{n} \neg y_i \neq y_j)$
$tr(Expr, \texttt{constraint}(R\ \texttt{maxCardinality}\ n), X)$	$tr(Expr, \texttt{restriction}(R\ \texttt{maxCardinality}\ n), X)$ $(tr(Expr, \texttt{constraint}(R\ \texttt{minCardinality}\ n), X)) \wedge$
$tr(Expr, \texttt{constraint}(R\ \texttt{cardinality}\ n), X)$	$(tr(Expr, \texttt{constraint}(R\ \texttt{maxCardinality}\ n), X))$
$tr(Expr, \texttt{constraint}(R\ \texttt{allValuesFrom}\ C), X)$	$\neg tr_{lhs}(Expr, X) \vee \neg X[R \twoheadrightarrow X] \vee \neg(not\ tr_l hs(C, y))$ ****
$tr(Expr, \texttt{constraint}(R\ \texttt{someValuesFrom}\ C), X)$	$\neg(tr_l hs(Expr, X)) \vee \neg(not\ X[R \twoheadrightarrow ?y])$ ****
* Note that the disjunction here causes non-Horn rules which however, can be easily transformed using the transformation from [Lloyd and Topor, 1984].	
** Here, $?x_{newIndex}$ denotes a new variable with an index not used anywhere else in the translation before.	
*** Note that $\forall$ here means that all variables $?x_i$ are all closed in the implications by an all-quantor.	
**** Notice that a disjunction with only clasically negated literals is transformed into an integrity constraint.	
Mapping OWL Flight Property Axioms to F-Logic	
`ObjectProperty(`R `super(`R_1`)...super(`R_l`)`	$\bigwedge ?x[R_i \twoheadrightarrow ?y] \leftarrow ?x[R \twoheadrightarrow ?y]$
`domain(`C_1`) ... domain(`C_m`)`	$\bigwedge \left\{ \begin{array}{lll} (1) & ?x[R \twoheadrightarrow ?y] \rightarrow ?x : C_i & \text{if } C_i \text{ is a named class} \\ (2) & ?x[R \twoheadrightarrow ?y] \rightarrow ?x[P \twoheadrightarrow o] & \text{if } C_i \text{ is of the form } \texttt{restriction}(R\ \texttt{value}(o)) \\ (3) & \text{apply (1)+(2) recursively if } C_i \text{ is of the form } \texttt{intersectionOf}(C_{i,1} \ \ldots \ C_{i,k}) & \end{array} \right.$
`range(`D_1`) ... range(`D_n`)`	$\bigwedge \left\{ \begin{array}{lll} (4) & ?x[R \twoheadrightarrow ?y] \rightarrow ?y : D_i & \text{if } D_i \text{ is a named class} \\ (5) & ?x[R \twoheadrightarrow ?y] \rightarrow ?y[P \twoheadrightarrow o] & \text{if } D_i \text{ is of the form } \texttt{restriction}(R\ \texttt{value}(o)) \\ (6) & \text{apply (4)+(5) recursively if } D_i \text{ is of the form } \texttt{intersectionOf}(D_{i,1} \ \ldots \ D_{i,k}) & \end{array} \right.$
`[inverseOf(`R'`)]`	$\left\{ \begin{array}{l} \forall ?x, ?y ?x[R \twoheadrightarrow ?y] \rightarrow ?y[R' \twoheadrightarrow ?x] \wedge \\ \forall ?x, ?y ?x[R' \twoheadrightarrow ?y] \rightarrow ?y[R \twoheadrightarrow ?x] \end{array} \right.$
`[Symmetric]`	$?y[R \twoheadrightarrow ?x] \leftarrow ?x[R \twoheadrightarrow ?y]$
`[Functional]`	$\neg ?x[R \twoheadrightarrow ?y] \vee \neg ?x[R \twoheadrightarrow ?z] \vee \neg ?y \neq ?z$
`[InverseFunctional]`	$\neg ?y[R \twoheadrightarrow ?x] \vee \neg ?z[R \twoheadrightarrow ?x] \vee \neg ?y \neq ?z$
`[Transitive])`	$?x[R \twoheadrightarrow ?z] \leftarrow ?x[R \twoheadrightarrow ?y] \wedge ?y[R \twoheadrightarrow ?z]$
`SubPropertyOf(`R_1 R_2`)`	$?x[R_2 \twoheadrightarrow ?y] \leftarrow ?x[R_1 \twoheadrightarrow ?y]$
`EquivalentProperties(`R_1 ... R_n`)`	$\bigwedge ?x[R_i \twoheadrightarrow ?y] \leftarrow ?x[R_j \twoheadrightarrow ?y]$
Mapping OWL Flight Facts to F-Logic	
`Individual(`o `type(`C_1`) ... type(`C_n`)` `value(`R_1 o_1`) ... value(`R_n o_n`))`	$\bigwedge o : C_i$ $\bigwedge o[R_i \twoheadrightarrow o_i]$

Table 4.2: Translating OWL Flight Abstract syntax into F-Logic(Datalog(IC,≠,not))

5 Conclusions and Future Work

In this document, we have presented the OWL Flight ontology language, which is an extension of OWL$^-$ with support for datatypes, cardinality and value constraints and local closed-world reasoning.

Because OWL Flight is strictly layered on top of OWL$^-$, it inherits the nice computational properties of this language. Because OWL$^-$ stays inside plain Datalog[1], reasoning with OWL$^-$ can be done using COTS (Common Off-The-Shelf) Datalog engines.

Because of the features added in OWL Flight, the underlying formalism also requires some extensions on top of Datalog, namely:

Integrity Constraints For the implementation of many of the features in OWL Flight, integrity constraints are required. However, implementation of support for integrity constraints in Datalog engines is fairly trivial (see Section 1.1).

Inequality The inequality symbol ($\neq$) is can occur in the body of a rule. Support for the symbol is already built-in in many Datalog engines, but can also be axiomatized in the language.

Default Negation Different types of constraints, as well as (local) closed-world reasoning, require default negation. Many Datalog engines support a form of default negation. When using stratified negation, the complexity of query answering in Datalog stays in the P complexity class. Thus, theoretically, no computational complexity by introducing stratified negation.

true, false The symbols *true* and $false$ need to be supported in the language. We have outlined in Section 1.1 how this support should be implemented. Many Datalog engines today already provide support for these symbols.

The next step after finishing the OWL Flight ontology language, is to create a concrete syntax for the language, called WSML-Flight, which will be developed in deliverable D16.10 of the WSML Working Group. This syntax will be an extension of WSML-Core [de Bruijn et al., 2004a].

Furthermore, a rule language will be developed based on OWL Flight. This should not be a major effort, since the semantics of OWL Flight is already given through a translation to a rule language. One could think of simply allowing all possible rules in F-Logic(Datalog(IC,$\neq$,not)), instead of allowing only those rules that result from the translation from the OWL Flight abstract syntax, as was given in Table 4.2.

Acknowledgements

We would especially like to thank Michael Kifer for useful comments and discussions around this deliverable

The work is funded by the European Commission under the projects DIP, Knowledge Web, SEKT, SWWS, and Esperonto; by Science Foundation Ireland

[1] Notice that only a translation to F-Logic exists for OWL Full$^-$. However, the result of the translation stays within the Datalog fragment of F-Logic and can thus be evaluated using a Datalog engine.

under the DERI-Lion project; and by the Vienna city government under the CoOperate program.

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

Bibliography

[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.

[Baader and Hanschke, 1991] Baader, F. and Hanschke, P. (1991). A scheme for integrating concrete domains into concept languages. Technical Report RR-91-10, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH.

[de Bruijn et al., 2004a] de Bruijn, J., Foxvog, D., and Oren, E. (2004a). WSML-Core. Deliverable D16.7, WSML, http://www.wsmo.org/wsml/. Available from http://www.wsmo.org/2004/d16/d16.7/v0.1/.

[de Bruijn et al., 2004b] de Bruijn, J., Polleres, A., Lara, R., and Fensel, D. (2004b). OWL$^-$. Deliverable d20.1v0.2, WSML.
Available from http://www.wsmo.org/2004/d20.1/v0.2/.

[Etzioni et al., 1997] Etzioni, O., Golden, K., and Weld, D. (1997). Sound and efficient closed-world reasoning for planning artificial intelligence. *Artificial Intelligence*, 89(1-2):113–148.

[Frühwirth, 1998] Frühwirth, T. (1998). Theory and practice of constraint handling rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37(1-3):95–138.

[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.

[Heflin and Muñoz-Avila, 2002] Heflin, J. and Muñoz-Avila, H. (2002). Lcw-based agent planning for the semantic web. In *Ontologies and the Semantic Web. Papers from the 2002 AAAI Workshop WS-02-11*, pages 63–70, Menlo Park, CA, USA. AAAI Press.

[Horrocks and Sattler, 2001] Horrocks, I. and Sattler, U. (2001). Ontology reasoning in the SHOQ(D) description logic. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI2001)*.

[Jaffar et al., 1998] Jaffar, J., Maher, M., Marriott, K., and Stuckey, P. (1998). The semantics of constraint logic programs. *Journal of Logic Programming*, 37(1–3):1–46.

[Jaffar and Maher, 1994] Jaffar, J. and Maher, M. J. (1994). Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581.

[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.

[Lloyd and Topor, 1984] Lloyd, J. W. and Topor, R. W. (1984). Making prolog more expressive. *Journal of Logic Programming*, 1(3):225–240.

[Marriott and Stuckey, 1998] Marriott, K. and Stuckey, P. J. (1998). MIT Press.

[Pan and Horrocks, 2003] Pan, J. Z. and Horrocks, I. (2003). Web ontology reasoning with datatype groups. In *2nd International Semantic Web Conference (ISWC2003)*, pages 47–63.

[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.

[Patel-Schneider et al., 2004] Patel-Schneider, P. F., Hayes, P., and Horrocks, I. (2004). OWL web ontology language semantics and abstract syntax. Recommendation 10 February 2004, W3C.

[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/.