Web Ontology Reasoning with Datatype Groups

Jeff Z. Pan and Ian Horrocks

Department of Computer Science,
University of Manchester, UK M13 9PL
{pan,horrocks}@cs.man.ac.uk

Abstract. When providing reasoning services for ontology languages such as DAML+OIL and OWL, it is necessary for description logics to deal with "concrete" datatypes (strings, integers, etc.) as well as "abstract" concepts and relationships. In this paper, we present a new approach, the datatype group approach, to integrating DLs with multiple datatypes. We discuss the advantages of such approach over the existing ones and show how a tableaux algorithm for the description logic $\mathcal{SHOQ}(\mathbf{D_n})$ can be modified in order to reason with datatype groups.

1 Introduction

Datatypes are important in the Semantic Web ontologies and applications, because most of which need to represent, in some way, various "real world" properties such as size, weight and duration, and some other complex user defined datatypes. Reasoning and querying over datatype properties are important and necessary if these properties are to be understood by machines.

For instance, e-shops may need to classify items according to their sizes, and to reason that an item which has height less than 5cm and the sum of length and width less than 10cm belongs to a class, called "small-items", for which no shipping costs are charged. Accordingly the billing system will charge no shipping fees for all the instances of the "small-items" class.

Various Web ontology languages, such as RDF(S) [4], OIL [5], DAML+OIL[1] [7] and OWL[2], have witnessed the importance of datatypes in the Semantic Web. All of them support datatypes. For instance, the DAML+OIL language supports unary datatype predicates and qualified number restrictions on unary datatype predicates, e.g. a "less than 21" predicate could be used with the datatype property *age* to describe objects having age less than 21.[3]

Description Logics (DLs)[1], a family of logical formalisms for the representation of and reasoning about conceptual knowledge, are of crucial importance to

[1] http://www.daml.org/
[2] http://www.w3.org/2001/sw/WebOnt/
[3] It is important to distinguish between a unary predicate such as "less than 21", which is true of any number x that is less than 21, and a binary predicate such as "less than", which is true of any pair of numbers x, y where x is less than y.

the development of the Semantic Web. Their role is to provide formal underpinnings and automated reasoning services for Semantic Web ontology languages such as OIL, DAML+OIL and OWL.

Using datatypes within Semantic Web ontology languages presents new requirements for DL reasoning services. Firstly, such reasoning services should be compatible with the XML Schema type system[3], and may need to support many different datatypes. Furthermore, they should be easy to extend when new datatypes are required.

DL researchers have been working on combining DLs and datatypes for quite a long time. Baader and Hanschke [2] first presented the concrete domain approach, Lutz [10] studied the effect on complexity of adding concrete domains to a range of DLs, Horrocks and Sattler [8] proposed the $\mathcal{SHOQ}(\mathbf{D})$ DL which combines DLs and type systems (e.g. the XML Schema type system), and more recently Pan and Horrocks [12] presented the $\mathcal{SHOQ}(\mathbf{D_n})$ DL, which extends $\mathcal{SHOQ}(\mathbf{D})$ with n-ary datatype predicates and qualified number restrictions.

To reason with $\mathcal{SHOQ}(\mathbf{D})$ and $\mathcal{SHOQ}(\mathbf{D_n})$, type checkers are introduced to work with DL "concept reasoners". By using a separate type checker, we can deal with an arbitrary conforming set of datatypes and predicates without compromising the compactness of the concept language or the soundness and completeness of the decision procedure [8]. Whenever new datatypes are required, only the type checkers need to be updated and the DL concept reasoner can be reused. The result is a framework that is both robust and flexible.

To support type systems and type checkers, in this paper, we present the datatype group approach, which extends existing approaches in order to overcome problems and limitations such as counter-intuitive negation, disjointness of different datatypes and mixed datatype predicates. We then describe an algorithm for reasoning with $\mathcal{SHOQ}(\mathbf{D_n})$, which improves on the one presented in [12] by allowing simpler "deterministic" type checkers to be used.

Next section, we will show the expressive power of the $\mathcal{SHOQ}(\mathbf{D_n})$ DL by an example of using n-ary datatype predicates.

2 An Example: Using Datatypes

Maybe you still remember the "small-items" example presented in last section, in which `the sum of ... less than 10cm` is an n-ary datatype predicate. In this section, we give another example of using n-ary (this time n=2) datatype predicates to support unit mapping.

Example 1 Miles and Kilometers[4] Unit mapping is important because of the variety of units. For instance, you can find more than one hundred and sixty length units in `http://www.chemie.de/`[5]. This example concerns the mapping between the units of mile and kilometer.

[4] This example is inspired by a discussion about datatypes in the www-rdf-logic mailing list: `http://lists.w3.org/Archives/Public/www-rdf-logic/2003Mar/0048.html`.

[5] `http://www.chemie.de/tools/units.php3?language=e\&property=m`

Firstly, we define two datatypes to represent the units of mile and kilometer. Since the positive float is not a built-in XML Schema datatype, we define two derived XML Schema datatypes[6] lengthInMile and lengthInKMtr, as follows:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                xmlns="http://www.example.org/length-units.xsd">
  <xsd:simpleType name="lengthInMile">
    <xsd:restriction base="xsd:float">
      <xsd:minInclusive value="0"/>
    </xsd:restriction base="xsd:float">
  </xsd:simpleType>
  <xsd:simpleType name="lengthInKMtr">
    <xsd:restriction base="xsd:float">
      <xsd:minInclusive value="0"/>
    </xsd:restriction base="xsd:float">
  </xsd:simpleType>
</xsd:schema>
```

We can then make use of these datatypes in DAML+OIL ontologies.[7] E.g., if we have datatype properties "length-mile" and "length-kmtr" defined in a river ontology:

```
<rdf:RDF
         xmlns="http://www.example.org/river#"
           ...
    xmlns:unit="http://www.example.org/length-units.xsd">
    <owl:DataTypeProperty rdf:ID="length">
      <rdfs:range rdf:resource="xsd:float"/>
    </owl:DataTypeProperty>
    <owl:Class rdf:ID="River"/>
    <owl:DataTypeProperty rdf:ID="length-mile">
      <rdfs:subPropertyOf rdf:resource="#length"/>
      <rdfs:domain rdf:resource="#River"/>
      <rdfs:range rdf:resource="unit:lengthInMile"/>
    </owl:DataTypeProperty>
    <owl:DataTypeProperty rdf:ID="length-kmtr">
      <rdfs:subPropertyOf rdf:resource="#length"/>
      <rdfs:domain rdf:resource="#River"/>
      <rdfs:range rdf:resource="unit:lengthInKMtr"/>
    </owl:DataTypeProperty>
```

we can describe the length of the Yangtze river as 3937.5 miles *and* 6300 kilometers:

```
    <River rdf:ID="Yangtze">
      <length-mile rdf:datatype="unit:lengthInMile">3937.5</length>
    </River>
    <River rdf:ID="Yangtze">
      <length-kmtr rdf:datatype="unit:lengthInKMtr">6300</length>
    </River>
```

We can specify the mapping between miles and kilometers using a binary datatype predicate. Sadly, XML Schema does not support n-ary datatype predicates. We have

[6] When we talk about XML Schema datatype in this paper, we mean XML Schema simple types.

[7] OWL currently does not support the use of derived datatypes.

used an XML style syntax to present this predicate in `http://www.example.org/length-units.xsd` as follows:[8]

```
<predicate name="kmtrsPerMile" arity="2">
  <par var="i" base="lengthInKMtr">
  <par var="j" base="lengthInMile">
  <constraint val="i=1.6*j">
</predicate>
```

The binary (with *arity* = 2) datatype predicate "kmtrsPerMile" is defined over the datatypes "lengthInKMtr" and "lengthInMile".

Now we can add a restriction to the "River" class and require that the value of the "length-kmtr" property be 1.6 times that of the "length-mile" property. Again, we could imagine an extension of a language such as OWL to support the use of n-ary datatype predicates:

```
<owl:Class rdf:about="River">
  <rdfs:subClassOf>
    <owl:NaryRestriction>
      <owl:onProperties rdf:parseType="Collection">
        <owl:DatatypeProperty rdf:ID="length-kmtr">
        <owl:DatatypeProperty rdf:ID="length-mile">
      </owl:onProperties>
      <owl:allTuplesSatisfy rdf:resource="unit:kmtrsPerMile"/>
    </owl:NaryRestriction>
  </rdfs:subClassOf>
</owl:Class>
```

Note that, in the above restriction, the order of the properties in the `onProperties` list is significant, and "kmtrsPerMile" is a datatype predicate whose arity must match the number of properties in the `onProperties` list. Such restriction is expressible in the $\mathcal{SHOQ}(\mathbf{D_n})$ DL (see section 4).

One example of reasoning with the above ontology and datatypes is to check whether a large set of instances of the ontology class River are consistent with the above restriction of the River class. ♢

As we hope the above example shows, n-ary datatype predicates would be very useful in Semantic Web ontologies and applications. In next section, we will start to investigate how to provide DL reasoning services for ontologies and datatypes.

3 Concrete Domains and Datatype Groups

As mentioned in Section 1, Description Logic researchers have been working on combining DLs and datatypes for quite a long time, although they might not always have used the term "datatype". It was Baader and Hanschke [2] who first presented a rigorous treatment of datatypes, which they called "concrete domains". Lutz [9] presents a survey of DLs with concrete domains, concentrating on the effect on complexity and decidability of adding concrete domains to

[8] Currently we are also working on extending the common DL Interface DIG/1.1 to support n-ary datatype predicates in a similar manner.

various DLs. More recently, Horrocks and Sattler [8] proposed a new approach to cope with datatypes structured by a type system. In the rest of this section, we will briefly describe the above two approaches, explaining their advantages and disadvantages in coping with multiple datatypes, then extend these approaches and present the datatype group approach.

3.1 The Concrete Domain Approach

A "concrete domain" is formally defined as followed:

Definition 1 (**Concrete Domain**) A concrete domain $\mathcal{D}$ consists of a pair ($\Delta_\mathcal{D}$,$\Phi_\mathcal{D}$), where $\Delta_\mathcal{D}$ is the domain of $\mathcal{D}$ and $\Phi_\mathcal{D}$ is a set of predicate names. Each predicate name P is associated with an arity n, and an n-ary predicate $P^\mathcal{D} \subseteq \Delta_\mathcal{D}^n$. Let $\mathbf{V}$ be a set of variables. A predicate conjunction of the form

$$c = \bigwedge_{j=1}^{k} P_j(v_1^{(j)}, \ldots, v_{n_j}^{(j)}), \tag{1}$$

where P_j is an n_j-ary predicate and the $v_i^{(j)}$ are variables from $\mathbf{V}$, is called *satisfiable* iff there exists a function δ mapping the variables in c to data values in $\Delta_\mathcal{D}$ s.t. $(\delta(v_1^{(j)}), \ldots, \delta(v_{n_j}^{(j)})) \in P_j^\mathcal{D}$ for $1 \leq j \leq k$. Such a function δ is called a *solution* for c. A concrete domain $\mathcal{D}$ is called *admissible* iff

1. $\Phi_\mathcal{D}$ is closed under negation[9] and contains a name $\top_\mathcal{D}$ for $\Delta_\mathcal{D}$ and
2. the satisfiability problem for finite conjunctions of predicates is decidable.

By $\overline{P}$, we denote the name for the negation of the predicate P, i.e., if the arity of P is n, then $\overline{P}^\mathcal{D} = \Delta_\mathcal{D}^n \setminus P^\mathcal{D}$. ◇

Here is an example of a concrete domain.

Example 2 The Concrete Domain $\mathcal{N} = (\Delta_\mathcal{N}, \Phi_\mathcal{N})$, where $\Delta_\mathcal{N}$ is the set of non-negative integers and $\Phi_\mathcal{N} = \{\geq, \geq_n\}$.[10] $\mathcal{N}$ is not admissible since $\Phi_\mathcal{N}$ doesn't satisfy condition 1 of admissible in Definition 1; in order to make it admissible we would have to extend $\Phi_\mathcal{N}$ to $\{\top_\mathcal{N}, \bot_\mathcal{N}\} \cup \{<, \geq, <_n, \geq_n\}$. ♢

The introduction of concrete domains in Baader and Hanschke [2] gives a firm basis for integrating DLs with datatypes. More examples of admissible concrete domains can be found in Section 2.4 of [10]. We can integrate an arbitrary concrete domain $\mathcal{D}$ into the $\mathcal{ALC}$ DL and give the $\mathcal{ALC}(\mathcal{D})$ DL. The syntax and semantics of this DL are described in [2, 10]. The new concept expression (about datatypes) of $\mathcal{ALC}(\mathcal{D})$ is the predicate restriction $P(u_1, \ldots, u_n)$, where P is a predicate name in $\Phi_\mathcal{D}$ and $u_1, \ldots, u_n$ are feature chains.[11]

Although the concrete domain approach does a very good job in combining DLs with *one* concrete domain, there are some problems when combining DLs with more than one concrete domain, especially in the context of Web ontology reasoning.

[9] Closed under negation requires that if $P \in \Phi_\mathcal{D}$, then $\overline{P} \in \Phi_\mathcal{D}$.
[10] Note that $\geq$ is a binary predicate and $\geq_n$ is a unary predicate.
[11] See [2, 10] for more about feature chains.

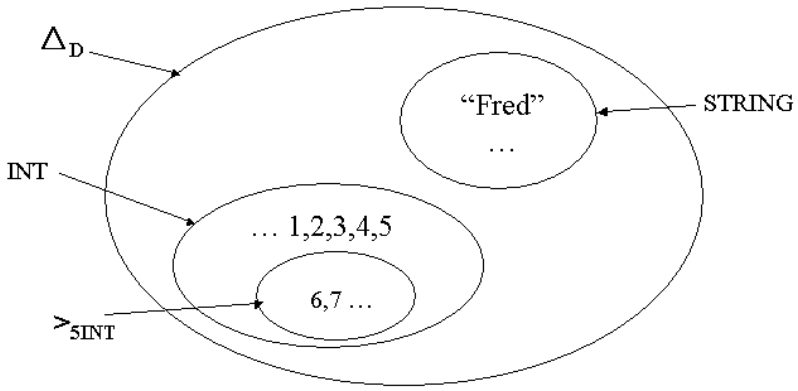

Fig. 1. The $>_5$ and $\leq_5$ Datatype Predicates

Negated Datatypes In the concrete domain approach, when two admissible concrete domains $\mathcal{D}_1$ and $\mathcal{D}_2$ are combined to form a new concrete domain $\mathcal{D}_1 \oplus \mathcal{D}_2$, the interpretation of the negation of each predicate name $P_j \in \Phi_{\mathcal{D}_1}$ will be modified from $\overline{P_j}^{\mathcal{D}_1} = \Delta_{\mathcal{D}_1}^{n_j} \setminus P_j^{\mathcal{D}_1}$ to $\overline{P_j}^{\mathcal{D}_1 \oplus \mathcal{D}_2} = (\Delta_{\mathcal{D}_1} \cup \Delta_{\mathcal{D}_2})^{n_j} \setminus P_j^{\mathcal{D}_1}$. We can see that $\overline{P_j}^{\mathcal{D}_1 \oplus \mathcal{D}_2}$ equals to $\overline{P_j}^{\mathcal{D}_1}$ union a set of n_j-ary tuples involving type errors, where at least one of the variables of P_j is mapped to a data value in $\mathcal{D}_2$ instead of $\mathcal{D}_1$. Counting type errors in the negations of datatype predicates may be counter-intuition. Let's see an example.

Example 3 The $>_5$ and $\leq_5$ Datatype Predicates We first only consider the concrete domains *INT*, where datatype predicates $>_5, \leq_5 \in \Phi_{INT}$. We have $\overline{>_5}^{INT} = \leq_5^{INT}$ (see figure 1). When we consider two concrete domain *INT* and *STRING*, we might still expect that $\overline{>_5}^{INT \oplus STRING}$ only include the *integers* that are not in $>_5^{INT}$, e.g., the integer 3. However, since $\overline{>_5}^{INT \oplus STRING} = (\Delta_{INT} \cup \Delta_{STRING}) \setminus >_5^{INT}$, the string "Fred" is also in $\overline{>_5}^{INT \oplus STRING}$. ♢

As well as being counter-intuitive, the change in the interpretation of $\overline{P_j}$ does not fit well with the idea of using type checkers to work with DL reasoner. We will come back to this in Section 3.3.

Disjoint Domains The concrete domain approach requires that two concrete domains be disjoint with each other if they are to be combined to form a new concrete domain. This does not accord with XML Schema datatypes, where some datatypes can be sub-types of other datatypes.

Mixed Datatype Predicates The "kmtrsPerMile" example illustrates another limitation of the concrete domain approach: all of the arguments to a predicate must be from the same concrete domain. The example shows that in some cases it may be useful to have predicates taking arguments of different datatypes.

3.2 The Type System Approach

To solve the "disjoint domains" problem mentioned in the previous section, Horrocks and Sattler [8] proposed a new approach to combine DLs and *type systems* (e.g. XML Schema type system). A type system typically defines a set of "base datatypes", such as *integer* or *string*, and provides a mechanism for deriving new datatypes from existing ones. In this approach, multiple datatypes may be defined over a *universal concrete domain*.

Definition 2 (**Universal Concrete Domain**) The universal concrete domain $\mathbf{D}$ consists of a pair $(\Delta_{\mathbf{D}},\Phi^1)$, where $\Delta_{\mathbf{D}}$ is the domain of all datatypes and Φ^1 is a set of datatype (unary datatype predicate) names. Each datatype name d is associated with a unary datatype predicate $d^{\mathbf{D}} \subseteq \Delta_{\mathbf{D}}$. Let $\mathbf{V}$ be a set of variables, a datatype conjunction of the form

$$c_1 = \bigwedge_{j=1}^{k} d_j(v^{(j)}), \tag{2}$$

where d_j is a (possibly negated) datatype from Φ^1 and the $v^{(j)}$ are variables from $\mathbf{V}$, is called *satisfiable* iff there exists a function δ mapping the variables in c_1 to data values in $\Delta_{\mathbf{D}}$ s.t. $\delta(v^{(j)}) \in d^{\mathbf{D}}$ for $1 \leq j \leq k$. Such a function δ is called a *solution* for c_1. By $\overline{d}$, we denote the name for the negation of the datatype d, and $\overline{d}^{\mathbf{D}} = \Delta_{\mathbf{D}} \setminus d^{\mathbf{D}}$.

A set of datatypes Φ^1 is called *conforming* iff $\Delta_{\mathbf{D}}$ is disjoint with the abstract domain $\Delta^{\mathcal{I}}$ and the satisfiability problem for finite conjunctions of datatypes over Φ^1 is decidable. ◇

In this approach, the universal concrete domain $\mathbf{D}$ is treated as the only concrete domain,[12] with datatypes being unary predicates over $\mathbf{D}$. Datatypes are considered to be sufficiently structured by type systems, which may include a derivation mechanism and built-in ordering relation. The satisfiability problem (2) is, therefore, much easier than that of the concrete domain approach (1).

Example 4 Miles and Kilometers (cont.) As shown in Example 1, lengthInMile and lengthInKMtr can be defined as derived XML Schema datatypes of FLOAT. In $\mathbf{D}$, lengthInMile, lengthInKMtr and FLOAT are datatype names of Φ^1. Since there are no n-ary datatype predicate names in Φ^1, it is not possible to represent the binary datatype predicate kilosPerMile using this approach. ♢

The type system approach provides an easy way to combine DLs with XML Schema datatypes. Horrocks and Sattler [8] integrated the universal concrete domain ($\mathbf{D}$) as well as nominals ($\mathcal{O}$) into the $\mathcal{SHQ}$ DL to give the $\mathcal{SHOQ}(\mathbf{D})$ DL. In order to make $\mathcal{SHOQ}(\mathbf{D})$ decidable, feature chains are not allowed. In addition, roles are divided into disjoint sets of abstract roles and concrete roles. The datatype constructs included in $\mathcal{SHOQ}(\mathbf{D})$ are datatype exists $\exists T.d$ and datatype value $\forall T.d$, where T is a concrete role and d is a datatype name in Φ^1. Detailed discussions on the differences in datatype handling between $\mathcal{SHOQ}(\mathbf{D})$ and $\mathcal{ALC}(\mathcal{D})$ can be found in [11].

The main disadvantage of the type system approach is that it doesn't support n-ary datatype predicates. Furthermore, since the interpretation of $\overline{d}$ is defined as $\Delta_{\mathbf{D}} \setminus d^{\mathbf{D}}$, the negated datatypes problem mentioned in Section 3.1 still exists.

[12] In [8]'s notation, $\mathbf{D}$ refers to Φ^1 the set of datatypes. We call it Φ^1 in order to make it more consistent with $\Phi_{\mathcal{D}}$ in Definition 1.

Example 5 The $>_5$ and $\leq_5$ Datatype Predicates (cont.) Let us revisit Figure 1 in the type system approach. Now $INT, >_5, \leq_5, STRING$ are datatype names in Φ^1. Although the interpretation of $\overline{INT}$ is fine (the integer 3 is not in $\overline{INT}^{\mathbf{D}}$ and the string "Fred" is), the interpretation of $\overline{>_5}$ is still quite "strange": both the integer 3 and the string "Fred" are in $\overline{>_5}^{\mathbf{D}}$. ♢

This example suggests that we might want to deal with base datatypes and derived datatypes in different manners.

3.3 The Datatype Group Approach

In this section we describe an extension of the type system approach which we call the datatype group approach. Our motivation is to provide an easy and intuitive way to cope with datatypes structured by type systems, and to support n-ary datatype predicates such that (i) the interpretations of negations of datatype predicates does not change when new datatypes are introduced into a datatype group, and (ii) it is possible to reuse existing concrete domain algorithms for the satisfiability problem of predicate conjunctions (1). A "datatype group" is formally defined as follows.

Definition 3 (**Datatype Group**) *A datatype group* $\mathcal{G}$ is a tuple $(\Delta_{\mathbf{D}}, \mathbf{D}_{\mathcal{G}}, \Phi^1_{\mathcal{G}}, \Phi_{\mathcal{G}})$, where $\Delta_{\mathbf{D}}$, which is disjoint with the abstract domain $\Delta^{\mathcal{I}}$, is the datatype domain covering all datatypes, $\mathbf{D}_{\mathcal{G}}$ is a set of base datatype names, $\Phi^1_{\mathcal{G}}$ is a set of derived datatype names and $\Phi_{\mathcal{G}}$ is a set of predicate names. Each base datatype name $d \in \mathbf{D}_{\mathcal{G}}$ is associated with a base datatype $d^{\mathbf{D}} \subseteq \Delta_{\mathbf{D}}$, each derived datatype name $d' \in \Phi^1_{\mathcal{G}}$ is associated with a derived datatype $d'^{\mathbf{D}} \subseteq d^{\mathbf{D}}$, where $d \in \mathbf{D}_{\mathcal{G}}$, and each predicate name $P \in \Phi_{\mathcal{G}}$ is associated with an arity n ($n > 1$) and a n-ary predicate $P^{\mathbf{D}} \subseteq d_1^{\mathbf{D}} \times \ldots \times d_n^{\mathbf{D}} \subseteq \Delta^n_{\mathbf{D}}$, where $d_1 \ldots d_n \in \mathbf{D}_{\mathcal{G}} \cup \Phi^1_{\mathcal{G}}$.

The domain function $\mathsf{dom}(p, i)$ returns the domain of the i-th argument of the (possibly unary) predicate p, where datatypes can be regarded as unary predicates. According to the above definition, $\mathsf{dom}(p, i)$ is defined as

1. for each $d \in \mathbf{D}_{\mathcal{G}}$, $\mathsf{dom}(d, 1) = \Delta_{\mathbf{D}}$;
2. for each $d' \in \Phi^1_{\mathcal{G}}$, $\mathsf{dom}(d', 1) = d^{\mathbf{D}}$;
3. for each $P \in \Phi_{\mathcal{G}}$, $\mathsf{dom}(P, i) = d_i^{\mathbf{D}}$ $(1 < i \leq n)$ if the arity of P is n.

Let $\mathbf{V}$ be a set of variables. We will consider predicate conjunctions over $\mathcal{G}$ of the form

$$\mathscr{C} = \bigwedge_{j=1}^{k} p_j(v_1^{(j)}, \ldots, v_{n_j}^{(j)}), \tag{3}$$

where p_j is an n_j-ary predicate in $\mathbf{D}_{\mathcal{G}} \cup \Phi^1_{\mathcal{G}} \cup \Phi_{\mathcal{G}}$, and the $v_i^{(j)}$ are variables from $\mathbf{V}$. A predicate conjunction $\mathscr{C}$ is called *satisfiable* iff there exists a function δ mapping the variables in $\mathscr{C}$ to data values in $\Delta_{\mathbf{D}}$ s.t. $\langle \delta(v_1^{(j)}), \ldots, \delta(v_{n_j}^{(j)}) \rangle \in p_j^{\mathbf{D}}$ for $1 \leq j \leq k$. Such a function δ is called a *solution* for $\mathscr{C}$. A datatype group $\mathcal{G}$ is *conforming* iff

1. $\mathbf{D}_{\mathcal{G}}$, $\Phi^1_{\mathcal{G}}$ and $\Phi_{\mathcal{G}}$ are closed under negation,
2. a binary inequality predicate $\neq_i \in \Phi_{\mathcal{G}}$ is defined for each datatype $d_i \in \mathbf{D}_{\mathcal{G}}$, and
3. the satisfiability problem for finite predicate conjunctions over $\mathcal{G}$ is decidable.

By $\overline{p}$, we denote the negation of the (possibly unary) predicate p, if the arity of p is n ($n \geq 1$), then $\mathsf{dom}(\overline{p}, 1) = \mathsf{dom}(p, 1), \ldots, \mathsf{dom}(\overline{p}, n) = \mathsf{dom}(p, n)$ and $\overline{p}^{\mathbf{D}} = \mathsf{dom}(p, 1) \times \ldots \times \mathsf{dom}(p, n) \setminus p^{\mathbf{D}}$.

For convenience, we use $\top_{\mathbf{D}}$ as the name of $\Delta_{\mathbf{D}}$ and $\bot_{\mathbf{D}}$ as the name of the negation of $\top_{\mathbf{D}}$. $\diamond$

A datatype group $\mathcal{G}$ is a natural n-ary predicate extension (introducing $\Phi_{\mathcal{G}}$) of the universal concrete domain $\mathbf{D}$, where the set Φ^1 of datatype names is divided into the set $\mathbf{D}_{\mathcal{G}}$ of base datatype names and the set $\Phi^1_{\mathcal{G}}$ of derived datatype names. The division is motivated by having different interpretation settings for their negations. The domain function $\mathsf{dom}(p, i)$ is defined for this purpose, so that the interpretation of the negation of datatypes and datatype predicates can be more intuitive, and do not change when new datatypes are introduced. Here is an example.

Example 6 The $>_5$ and $\leq_5$ Datatype Predicates (cont.) A datatype group $\mathcal{G}$ can be defined as

$$\begin{array}{l}(\Delta_{\mathbf{D}}, \\ \mathbf{D}_{\mathcal{G}} := \{INT, \overline{INT}, STRING, \overline{STRING}\}, \\ \Phi^1_{\mathcal{G}} := \{>_5, \leq_5\}, \\ \Phi_{\mathcal{G}} := \{\neq_{INT}, =_{INT}, \neq_{STRING}, =_{STRING}\}),\end{array}$$

where $\overline{INT}^{\mathbf{D}} = \mathsf{dom}(INT, 1) \setminus INT^{\mathbf{D}} = \Delta_{\mathbf{D}} \setminus INT^{\mathbf{D}}$, and $\overline{>_5}^{\mathbf{D}} = \mathsf{dom}(>_5, 1) \setminus >_5^{\mathbf{D}} = INT^{\mathbf{D}} \setminus >_5^{\mathbf{D}} = \leq_5^{\mathbf{D}}$. Therefore the integer 3 is in $\overline{>_5}^{\mathbf{D}}$ but not in $\overline{INT}^{\mathbf{D}}$, while the string "Fred" is in $\overline{INT}^{\mathbf{D}}$ but not in $\overline{>_5}^{\mathbf{D}}$. $\diamondsuit$

Since the datatype group approach supports n-ary datatype predicates, we can now present the binary predicate kilosPerMile in the miles and kilometers example.

Example 7 Miles and Kilometers (cont.) A datatype group $\mathcal{G}_2$ can be defined as

$$\begin{array}{l}(\Delta_{\mathbf{D}}, \\ \mathbf{D}_{\mathcal{G}_2} := \{FLOAT, \overline{FLOAT}\}, \\ \Phi^1_{\mathcal{G}_2} := \{lengthInMile, \overline{lengthInMile}, lengthInKMtr, \overline{lengthInKMtr}\}, \\ \Phi_{\mathcal{G}_2} := \{kmtrsPerMile, \overline{kmtrsPerMile}, \neq_{FLOAT}, =_{FLOAT}\}),\end{array}$$

where $\overline{kmtrsPerMile}^{\mathbf{D}} = \mathsf{dom}(kmtrsPerMile, 1) \times \mathsf{dom}(kmtrsPerMile, 2) \setminus kmtrsPerMile^{\mathbf{D}} = lengthInKmtr^{\mathbf{D}} \times lengthInMile^{\mathbf{D}} \setminus kmtrsPerMile^{\mathbf{D}}$. $\diamondsuit$

There is a close relationship between a conforming datatype group with only one base datatype and an admissible concrete domain.

Lemma 4 An admissible concrete domain $(\Delta_{\mathcal{D}}, \Phi_{\mathcal{D}})$ is a conforming datatype group $\mathcal{G} = (\Delta_{\mathbf{D}},\ \mathbf{D}_{\mathcal{G}} := \{\mathcal{D}\},\ \Phi^1_{\mathcal{G}}, \Phi_{\mathcal{G}})$, where $\Phi^1_{\mathcal{G}}$ is the set of unary predicate names in $\Phi_{\mathcal{D}}$ and $\Phi_{\mathcal{G}}$ is set of n-ary ($n > 1$) predicate names in $\Phi_{\mathcal{D}}$, if there exists a binary inequality predicate $\neq_{\mathcal{D}} \in \Phi_{\mathcal{D}}$.

Proof. Immediate consequence of Definition 1 and 3. $\square$

Now we show how two conforming datatype groups $\mathcal{G}_1$ and $\mathcal{G}_2$ can be combined to form a new datatype groups $\mathcal{G}_1 \oplus \mathcal{G}_2$. It turns out (Lemma 6 and 7) that the combination is also conforming in many cases.

Definition 5 Assume that $\mathcal{G}_1$ and $\mathcal{G}_2$ are conforming datatype groups. Then $\mathcal{G}_1 \oplus \mathcal{G}_2$ can be constructed as $(\Delta_{\mathbf{D}},\ \mathbf{D}_{\mathcal{G}_1 \oplus \mathcal{G}_2} := \mathbf{D}_{\mathcal{G}_1} \cup \mathbf{D}_{\mathcal{G}_2},\ \Phi^1_{\mathcal{G}_1 \oplus \mathcal{G}_2} := \Phi^1_{\mathcal{G}_1} \cup \Phi^1_{\mathcal{G}_2},\ \Phi_{\mathcal{G}_1 \oplus \mathcal{G}_2} := \Phi_{\mathcal{G}_1} \cup \Phi_{\mathcal{G}_2})$. ◇

Note that for a predicate p in either $\mathcal{G}_1$ or $\mathcal{G}_2$ $\overline{p}^{\mathbf{D}}$ doesn't changed after the combination.

Lemma 6 If $\mathcal{G}_1$ and $\mathcal{G}_2$ are conforming datatype groups where $\mathbf{D}_{\mathcal{G}_1} \cap \mathbf{D}_{\mathcal{G}_2} = \emptyset$, then $\mathcal{G}_1 \oplus \mathcal{G}_2$ is also a conforming datatype group.

Proof. Obviously $\mathcal{G}_1 \oplus \mathcal{G}_2$ satisfies the first two conditions of a conforming datatype group. Now we only focus on the third condition. Assume that a predicate conjunction

$$\mathscr{C} = \bigwedge_{j=1}^{k} P_j(v_1^{(j)}, \ldots, v_{n_j}^{(j)})$$

is given, where P_j are predicates of $\mathcal{G}_1 \oplus \mathcal{G}_2$.

1. If a variable v occurs as an argument of (possibly unary) predicates from both datatype groups, then $\mathscr{C}$ is not satisfiable, because $\mathbf{D}_{\mathcal{G}_1}$ and $\mathbf{D}_{\mathcal{G}_2}$ are disjoint.
2. Otherwise, $\mathscr{C}$ can be split into two predicate conjunctions $\mathscr{C}_1$ and $\mathscr{C}_2$ such that they are predicate conjunctions in $\mathcal{G}_1$ and $\mathcal{G}_2$ respectively and no variable occurs in both conjunctions. Therefore, we observe that $\mathscr{C}$ is satisfiable iff the satisfiability tests of the respective datatype groups succeed for $\mathscr{C}_1$ and $\mathscr{C}_2$. □

Note that we don't need to cope with disjunctions in the combination, while in the corresponding Lemma for $\mathcal{D}_1 \oplus \mathcal{D}_2$ (Lemma 2.4 in [2]) in the concrete domain approach, disjunctions must be handled because of type errors.

If the set of datatypes and predicates in a conforming datatype group $\mathcal{G}_1$ is a sub-set of the set of datatypes and predicates in another conforming datatype group $\mathcal{G}_2$, then trivially $\mathcal{G}_1 \oplus \mathcal{G}_2$ is also conforming.

Lemma 7 If $\mathcal{G}_1$ and $\mathcal{G}_2$ are conforming datatype group where $\mathbf{D}_{\mathcal{G}_1} \cup \Phi^1_{\mathcal{G}_1} \cup \Phi_{\mathcal{G}_1} \subseteq \mathbf{D}_{\mathcal{G}_2} \cup \Phi^1_{\mathcal{G}_2} \cup \Phi_{\mathcal{G}_2}$, then $\mathcal{G}_1 \oplus \mathcal{G}_2$ is also a conforming datatype group.

Proof. Immediate consequence of Definition 5, obviously $\mathcal{G}_1 \oplus \mathcal{G}_2$ is equivalent to $\mathcal{G}_2$. □

Lemma 6 and 7 give guidelines on how to build complex conforming datatype groups from simple ones. Based on the definition of a datatype group, we can define a type checker which works with a DL reasoner to answer datatype queries.

Definition 8 A *type checker* is a program that takes as input a finite predicate conjunction $\mathscr{C}$ over (one of) the conforming datatype group(s) it supports, and answers *satisfiable* if $\mathscr{C}$ is satisfiable and *unsatisfiable* otherwise. ◇

It is possible for a DL reasoner to work with many type checkers. Firstly, in the datatype group approach, the interpretation of $\overline{p}$ ($p \in \mathbf{D}_{\mathcal{G}} \cup \Phi^1_{\mathcal{G}} \cup \Phi_{\mathcal{G}}$) doesn't change when new datatypes are introduced, so the interpretation of each $\overline{p}$ supported by a type checker won't be affected by the existence of other type checkers. Secondly, assuming that the set of base datatypes of the conforming datatype group supported by each type checker is disjoint from each other, Lemma 6 shows that the combined datatype group of all these datatype groups supported by the type checkers is also conforming.

4 $\mathcal{SHOQ}(\mathbf{D_n})$

In this section, we give the definition of the $\mathcal{SHOQ}(\mathbf{D_n})$ DL that supports reasoning with datatype groups. Note that in DLs we talk about *concepts* and *roles* where in Web ontology languages we usually talk about *classes* and *properties*.

Definition 9 ($\mathcal{SHOQ}(\mathbf{D_n})$ **Syntax and Semantics**) Let $\mathbf{C}$, $\mathbf{R} = \mathbf{R}_A \uplus \mathbf{R}_\mathbf{D}$, $\mathbf{I}$ be disjoint sets of concept, abstract and concrete role and individual names.

For R and S roles, a *role axiom* is either a role inclusion, which is of the form $R \sqsubseteq S$ for $R, S \in \mathbf{R}_A$ or $R, S \in \mathbf{R}_\mathbf{D}$, or a transitivity axiom, which is of the form $\mathsf{Trans}(R)$ for $R \in \mathbf{R}_A$. A *role box* $\mathcal{R}$ is a finite set of role axioms. A role R is called *simple* if, for $\sqsubseteq^*$ the transitive reflexive closure of $\sqsubseteq$ on $\mathcal{R}$ and for each role S, $S \sqsubseteq^* R$ implies $\mathsf{Trans}(S) \notin \mathcal{R}$.

The set of concept terms of $\mathcal{SHOQ}(\mathbf{D_n})$ is inductively defined. As a starting point of the induction, any element A of $\mathbf{C}$ is a concept term (atomic terms). Now let C and D be concept terms, o be an individual, R be a abstract role name, $T_1, \ldots, T_n$ be concrete role names, S be a simple role name, P be an n-ary datatype predicate name. Then complex concepts can be built using the operators shown in Figure 2.

The semantics is defined in terms of an interpretation $\mathcal{I} = (\Delta^\mathcal{I}, \cdot^\mathcal{I})$, where $\Delta^\mathcal{I}$ (the abstract domain) is a nonempty set and and $\cdot^\mathcal{I}$ (the interpretation function) maps atomic and complex concepts, roles and nominals according to Figure 2. Note that $\sharp$ denotes set cardinality, $\Delta_\mathbf{D}$ is the datatype domain and $\mathsf{dom}(P, i)$ is the domain function in a datatype group.

An interpretation $\mathcal{I} = (\Delta^\mathcal{I}, \cdot^\mathcal{I})$ satisfies a role inclusion axiom $R_1 \sqsubseteq R_2$ iff $R_1^\mathcal{I} \subseteq R_2^\mathcal{I}$, and it satisfies a transitivity axiom $\mathsf{Trans}(R)$ iff $R^\mathcal{I} = (R^\mathcal{I})^+$. An interpretation satisfies a role box $\mathcal{R}$ iff it satisfies each axiom in $\mathcal{R}$. A $\mathcal{SHOQ}(\mathbf{D_n})$-concept C is satisfiable w.r.t. a role box $\mathcal{R}$ iff there is an interpretation $\mathcal{I}$ with $C^\mathcal{I} \neq \emptyset$ that satisfies $\mathcal{R}$. Such an interpretation is called a *model* of C w.r.t. $\mathcal{R}$. A concept C is *subsumed* by a concept D w.r.t. $\mathcal{R}$ iff $C^\mathcal{I} \sqsubseteq D^\mathcal{I}$ for each interpretation $\mathcal{I}$ satisfying $\mathcal{R}$. Two concepts are said to be equivalent (w.r.t. $\mathcal{R}$) iff they mutually subsume each other (w.r.t. $\mathcal{R}$). $\diamond$

Note that the use of domain function $\mathsf{dom}(P, i)$ in the semantics of datatype constructs in $\mathcal{SHOQ}(\mathbf{D_n})$ is to ensure that every $\mathcal{SHOQ}(\mathbf{D_n})$-concept can be converted into an equivalent one in negation normal form (NNF), i.e., with negations only applying to concept names. This is important for tableau algorithms, which typically operate only on concepts in NNF.

Example 8 Miles and Kilometers (cont.) The *River* concept can be defined as a sub-concept of the $\mathcal{SHOQ}(\mathbf{D_n})$-concept $\forall$*length-kmtr*,*length-mile*.`kmtrsPerMile`, that is, *River* $\sqsubseteq \forall$*length-kmtr*,*length-mile*.`kmtrsPerMile`. $\diamondsuit$

5 A Tableaux Algorithm for $\mathcal{SHOQ}(\mathbf{D_n})$

A key feature of DLs is the provision of reasoning services. These services can be used to support the design and deployment of ontologies using DL based ontology languages such as DAML+OIL and OWL. In common with other expressive DLs, we can use a tableau algorithm that checks concept satisfiability w.r.t. a role box in order to provide reasoning services for $\mathcal{SHOQ}(\mathbf{D_n})$: a range of reasoning problems, including subsumption and satisfiability w.r.t. a terminology

Construct Name	Syntax	Semantics
atomic concept	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
abstract role	R	$R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
concrete role	T	$T^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta_{\mathbf{D}}$
nominals	$\{o\}$	$\{o\}^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}, \sharp\{o\}^{\mathcal{I}} = 1$
conjunction	$C \sqcap D$	$(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$
disjunction	$C \sqcup D$	$(C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$
negation	$\neg C$	$(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
exists restriction	$\exists R.C$	$(\exists R.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \exists y.\langle x,y\rangle \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$
value restriction	$\forall R.C$	$(\forall R.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \forall y.\langle x,y\rangle \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$
atleast restriction	$\geqslant nS.C$	$(\geqslant nS.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \sharp\{y.\langle x,y\rangle \in S^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\} \geq n\}$
atmost restriction	$\leqslant nS.C$	$(\leqslant nS.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \sharp\{y.\langle x,y\rangle \in S^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\} \leq n\}$
datatype exists	$\exists T_1, \cdots, T_n.P_n$	$(\exists T_1, \cdots, T_n.P_n)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \exists y_1 \cdots y_n.y_i \in \mathsf{dom}(P_n, i)$ $\wedge \langle x, y_i\rangle \in T_i^{\mathcal{I}}$ (for $1 \leq i \leq n$) $\wedge \langle y_1, \cdots y_n\rangle \in P_n^{\mathbf{D}}\}$
datatype value	$\forall T_1, \cdots, T_n.P_n$	$(\forall T_1, \cdots, T_n.P_n)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \forall y_1 \cdots y_n.y_i \in \mathsf{dom}(P_n, i)$ $\wedge \langle x, y_i\rangle \in T_i^{\mathcal{I}}$ (for $1 \leq i \leq n$) $\rightarrow \langle y_1, \cdots y_n\rangle \in P_n^{\mathbf{D}}\}$
datatype atleast	$\geqslant mT_1, \ldots, T_n.P_n$	$(\geqslant mT_1, \ldots, T_n.P_n)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \sharp\{\langle y_1 \cdots y_n\rangle \mid y_i \in \mathsf{dom}(P_n, i)$ $\wedge \langle x, y_i\rangle \in T_i^{\mathcal{I}}$ (for $1 \leq i \leq n$) $\wedge \langle y_1, \cdots y_n\rangle \in P_n^{\mathbf{D}}\} \geq m\}$
datatype atmost	$\leqslant mT_1, \ldots, T_n.P_n$	$(\leqslant mT_1, \ldots, T_n.P_n)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \sharp\{\langle y_1 \cdots y_n\rangle \mid y_i \in \mathsf{dom}(P_n, i)$ $\wedge \langle x, y_i\rangle \in T_i^{\mathcal{I}}$ (for $1 \leq i \leq n$) $\wedge \langle y_1, \cdots y_n\rangle \in P_n^{\mathbf{D}}\} \leq m\}$

Fig. 2. Syntax and Semantics of $\mathcal{SHOQ}(\mathbf{D_n})$

(ontology) can be reduced to concept satisfiability [8]. As space is limited, and as the algorithm is similar to those presented in [8, 12], we will not describe it in detail here. Instead, we will sketch some of its more interesting features, and in particular those related to reasoning with datatype groups. The interested reader is referred to the online technical report[13] [13] for full details and proofs of the algorithm's soundness and completeness.

As with any tableau algorithm, the basic idea is to try to prove the satisfiability of a concept C (w.r.t. a role box $\mathcal{R}$) by building a model of C, i.e., (a structure that closely corresponds to) an interpretation $\mathcal{I}$ that satisfies $\mathcal{R}$ and in which $C^{\mathcal{I}}$ is not empty. The algorithm works on a (set of) tree(s), where nodes are labeled with sets of sub-concepts of C, and edges are labeled with sets of roles occurring in C. Nodes (edges) in the tree correspond to elements (tuples) in the interpretation of the concepts (roles) with which they are labeled. Normally, a single tree is initialised with a root node labeled $\{C\}$.

The algorithm exhaustively applies tableau rules that decompose the syntactic structure of the concepts in node labels, either expanding node labels, adding new edges and nodes to the tree(s), or merging edges and nodes. The application of a rule effectively explicates constraints on the interpretation implied by the concepts to which the rule was applied. E.g., if $A \sqcap B$ is in the label of a node x then the $\sqcap$-*rule* adds both A and B to the label, explicating the fact that if $x \in (A \sqcap B)^{\mathcal{I}}$, then both $x \in A^{\mathcal{I}}$ and $x \in B^{\mathcal{I}}$. Similarly, if $\exists R.A$ is in the label of a node x, then the $\exists$-*rule* adds a new node y labelled $\{A\}$ with an edge between

[13] `http://DL-Web.man.ac.uk/Doc/shoqdn-proofs.pdf`

x and y labeled $\{R\}$, explicating the fact that if $x \in (\exists R.A)^{\mathcal{I}}$, then there must exist a node y such that $\langle x, y\rangle \in R^{\mathcal{I}}$ and $y \in A^{\mathcal{I}}$.

An attempt to build a model fails if an obvious contradiction, often called a *clash*, is generated, e.g., if the label of some node contains both D and $\neg D$ for some concept D; it is successful if no more rules can be applied, and there are no clashes. It is relatively straightforward to prove that a concept is satisfiable if and only if the rules can be applied in such a way that a model is successfully constructed. The computational complexity of the algorithm stems from the fact that some rules are non-deterministic (e.g., the rule dealing with disjunctions); in practice this is dealt with by backtracking when a clash is detected, and trying a different non-deterministic rule application.

Various refinements of this basic technique are required in order to deal with a logic as expressive as $\mathcal{SHOQ}(\mathbf{D_n})$. In the first place, the algorithm operates on a forest of trees, as an additional tree must be constructed for each nominal in C (see Figure 2); a form of cycle check called *blocking* must also be used in order to guarantee termination [6]. In the second place, the algorithm needs to use a type checker to check constraints on the interpretation derived from datatype exists, value, atleast and atmost concepts.

5.1 Datatype Reasoning

Logics like $\mathcal{SHOQ}(\mathbf{D})$ and $\mathcal{SHOQ}(\mathbf{D_n})$ are designed so that reasoning about datatypes and values can be separated from reasoning about concepts and roles—this is the reason for the strict separation of the domains and of abstract and concrete roles. The result of the separation is that node labels can contain *either* concepts *or* datatypes and values, but never a mixture. This allows node labels containing datatypes and values to be checked using a separate type checker, with inconsistencies in such labels being treated as an additional clash condition. E.g., if a node label includes the concepts $\exists T.\texttt{string}$ and $\forall T.\texttt{real}$, then a new *concrete node* will be generated labeled $\{\texttt{string}, \texttt{real}\}$, and when checked with the type checker this would (presumably) return a clash on the grounds that there is no element that is in the interpretation of both `string` and `real`.

With $\mathcal{SHOQ}(\mathbf{D_n})$ the situation is more complex because it is necessary to deal with both n-ary predicates and datatype cardinality constraints that may be qualified with n-ary predicates. The algorithm deals with n-ary predicates by keeping track of tuples of concrete nodes that must satisfy datatype predicates, and it deals with datatype cardinality constraints by keeping track of inequalities between tuples of concrete nodes that were generated by the datatype $\geqslant_P$*-rule* in order to explicate datatype atleast concepts (merging such tuples could lead to non-termination as the $\geqslant_P$*-rule* might be applied again and cause new tuples to be generated).

The predicate relationships between (the values represented by) concrete nodes are taken into consideration by the type checker, which can check predicate conjunctions (see Definition 8). In the algorithm described in [12], the type checker must also ensure that the solution is consistent with inequalities between

tuples of concrete nodes. This means that it must be extended to deal with non-deterministic reasoning, because $\langle t_{j1}, \ldots, t_{jn}\rangle \dot{\neq} \langle t_{k1}, \ldots, t_{kn}\rangle$ is equivalent to

$$(t_{j1} \dot{\neq} t_{k1}) \sqcup \ldots \sqcup (t_{jn} \dot{\neq} t_{kn}). \tag{4}$$

In the new algorithm, this non-deterministic reasoning is pushed back into the tableau reasoner, which is already able to cope with non-determinism arising, e.g., from disjunction concepts. This is done by adding concepts "equivalent" to (4) to the node labels containing the relevant datatype atleast concept. For this purpose, we use concepts of the form

$$(\forall T_1^{C_{j1}}, T_1^{C_{k1}}. \neq_{d_{P_n,i}}) \sqcup \ldots \sqcup (\forall T_n^{C_{jn}}, T_n^{C_{kn}}. \neq_{d_{P_n,i}}), \tag{5}$$

where $C = \geqslant m T_1, \ldots, T_n.P_n$, is the datatype atleast concept in question, $T_1^{C_{j1}}, T_1^{C_{k1}}, \ldots, T_n^{C_{jn}}, T_n^{C_{kn}}$ are *superscripted concrete roles* and $\neq_{d_{P_n,i}}$ is the inequality predicate for the datatype $\mathsf{dom}(P_n, i)$ (recall that each datatype in a datatype group must be equipped with an inequality predicate). The superscripted concrete roles are generated by the $\geqslant_P$*-rule* and used to link the node x containing C to the new concrete nodes that the rule generates. The form of the superscript means that a superscripted role acts as a unique (w.r.t. the node x) name for a given concrete node. The result is that if ever two tuples created by applying the $\geqslant_P$*-rule* to C are merged, then the type checker will return a clash. This is because, whichever way the $\sqcup$*-rule* is applied to the disjunction (5), the predicate relationships will include $x \neq x$ for some concrete node x.

6 Discussion

As we have seen, using datatypes within Semantic Web ontology languages (such as DAML+OIL and OWL) presents new requirements for DL reasoning services. We have presented the datatype group approach, which extends the type system approach with n-ary predicates and a new treatment of predicate negation, so as to make it possible to use type checkers with DL reasoners. We have also sketched an improved algorithm for reasoning with $\mathcal{SHOQ}(\mathbf{D_n})$ using datatype groups. Type checkers for datatype groups should be easy to implement as we do not have to deal with disjunctions of predicate terms. Moreover, the similarity of conforming datatype groups and admissible concrete domains can be exploited in order to identify suitable datatype groups.

The resulting framework is both robust and extensible. On the one hand, most complex reasoning tasks take place within the well understood and provably correct tableau algorithm. On the other hand, it is relatively easy to add support for new datatypes and predicates, and this does not require any changes to the tableaux algorithm itself. An implementation of the algorithm along with a simple type checker (supporting integers and strings) is currently underway (based on the FaCT system), and will be used to evaluate empirical performance.

Although existing Web ontology languages such as DAML+OIL and OWL do not support n-ary predicates, we believe that they are useful/essential in

many realistic applications, and will be a prime candidate for inclusion in future extensions of these languages. The algorithm we have presented could be used to provide reasoning support for such extended Web ontology languages.

Acknowledgements

We would like to thank Ulrike Sattler, since the work presented here extends the original work on $\mathcal{SHOQ}(\mathbf{D})$. Thanks are also due to Carsten Lutz for his helpful discussion on inequality predicates.

Bibliography

[1] F. Baader, D. L. McGuiness, D. Nardi, and P. Patel-Schneider, editors. *Description Logic Handbook: Theory, implementation and applications*. Cambridge University Press, 2002.

[2] Franz Baader and Philipp Hanschke. A Schema for Integrating Concrete Domains into Concept Languages. In *Proc. of the 12th Int. Joint Conf. on Artificial Intelligence (IJCAI'91)*, pages 452–457, 1991.

[3] Paul V. Biron and Ashok Malhotra. Extensible Markup Language (XML) Schema Part 2: Datatypes – W3C Recommendation 02 May 2001. Technical report, World Wide Web Consortium, 2001. Available at `http://www.w3.org/TR/xmlschema-2/`.

[4] Dan Brickley and R.V. Guha. Resource Description Framework (RDF) Schema Specification 1.0. W3C Recommentdation, `URL http://www.w3.org/TR/rdf-schema`, Mar. 2000.

[5] Dieter Fensel, Frank van Harmelen, Ian Horrocks, Deborah L. McGuinness, and Peter F. Patel-Schneider. OIL: An ontology infrastructure for the semantic web. *IEEE Intelligent Systems*, 16(2):38–45, 2001.

[6] I. Horrocks, U. Sattler, and S. Tobies. Practical Reasoning for Expressive Description Logics. In H. Ganzinger, D. McAllester, and A. Voronkov, editors, *Proc. of the 6th Int. Conf. on Logic for Programming and Automated Reasoning (LPAR'99)*, pages 161–180, 1999.

[7] Ian Horrocks and Peter F. Patel-Schneider. The generation of DAML+OIL. In *Proc. of the 2001 Description Logic Workshop (DL 2001)*, pages 30–35. CEUR Electronic Workshop Proceedings, `http://ceur-ws.org/Vol-49/`, 2001.

[8] Ian Horrocks and Ulrike Sattler. Ontology reasoning in the $\mathcal{SHOQ}$(D) description logic. In *Proc. of the 17th Int. Joint Conf. on Artificial Intelligence (IJCAI 2001)*, pages 199–204, 2001.

[9] C. Lutz. Description logics with concrete domains—a survey. In *Advances in Modal Logics Volume 4*. World Scientific Publishing Co. Pte. Ltd., 2002.

[10] Carsten Lutz. *The Complexity of Reasoning with Concrete Domains*. PhD thesis, Teaching and Research Area for Theoretical Computer Science, RWTH Aachen, 2001.

[11] Jeff Z. Pan. Web Ontology Reasoning in the $\mathcal{SHOQ}(\mathbf{D_n})$ Description Logic. In *Carlos Areces and Maartin de Rijke, editors,Proceedings of the Methods for Modalities 2 (M4M-2)*, Nov 2001. ILLC, University of Amsterdam.

[12] Jeff Z. Pan and Ian Horrocks. Reasoning in the SHOQ(Dn) Description Logic. In Ian Horrocks and Sergio Tessaris, editors, *Proc. of the 2002 Int. Workshop on Description Logics (DL-2002)*, Apr. 2002.

[13] Jeff Z. Pan and Ian Horrocks. Reasoning in the $\mathcal{SHOQ}(\mathbf{D_n})$ Description Logic (Online Proofs), 2003. URL `http://DL-Web.man.ac.uk/Doc/shoqdn-proofs.pdf`.