Ontology schema for an agent belief store

K.L. Clark
klc@doc.ic.ac.uk

F.G. McCabe
frankmccabe@mac.com

November 20, 2006

Abstract

In this paper we explore the use of a formal ontology as a constraining framework for the belief store of a rational agent. The static beliefs of the agent are the axioms of the ontology. The dynamic beliefs are the descriptions of the individuals that are instances of the ontology classes. The individuals all have a unique identifier, an associated set of named classes to which they are believed to belong, and a set of property values. The ontology axioms act as a *schema* for the dynamic beliefs. Belief updates not conforming to the axioms lead to either rejection of the update or some other revision of the dynamic belief store to maintain consistency. Partial descriptions are augmented by inferences of property values and class memberships licensed by the axioms.

For concreteness we sketch how such an ontology based agent belief store could be implemented in a multi-threaded logic programming language with action rules and object oriented programming features called `Go!`. This language was specifically designed for implementing communicating rational agent applications. We shall see that its logic rules allow us to extend an ontology of classes and properties with rule defined n-ary relations and functions. Its action rules enable us to implement a consistency maintenance system that takes into account justifications for beliefs. The pragmatics of consistency maintenance is an issue not normally considered by the ontology community.

The paper assumes some familiarity with ontology specification using languages such as OWL DL and its subsets, and with logic programming.

1 Introduction

It is generally accepted that a shared ontology is essential for high level communication between agents (Huhns and Singh 1997), and messages in the agent communication languages such as KQML (Finin et al. 1994) can have a field that gives the name of the ontology used in the query or statement that is the content of the message. However, to our knowledge comparatively little work seems to have been done regarding the use of a formal ontology as the framework for the internal and dynamic beliefs of an agent, these beliefs being the descriptions of individuals belonging to the classes of the ontology.

Ontology specification languages such as OWL DL (McGuinness and van Harmelen 2004) are based on description logics (Horrocks et al. 2003). Inferences regarding the relationships between classes of an ontology are called T-box inferences, whereas those concerning descriptions of individuals are A-box inferences (Baader et al. 2004). There are very good inferences procedures for T-box reasoning - the sort of reasoning needed to answer queries about concepts, such as whether or not one class concept is subsumed by another based on their descriptions, and to test for inconsistency of the class and property axioms of an ontology. But for agents, whose beliefs mainly comprise descriptions of individuals, efficient A-box reasoning is as important, if not more so. Proposed query languages for OWL, such as OWL-QL (Fikes et al. 2003), need A-box inference as they are essentially queries to retrieve names of individuals and property values satisfying some class membership and property value constraints.

This issue is being addressed. There is an instance store implementation (Bechhofer et al. 2005) that can hold a relatively large set of individual descriptions stored in a data base. It supports the retrieval of all instances satisfying some OWL concept description. There is also an implementation (Nagy et al. 2006) that similarly finds all the individuals with descriptions stored in a data base that are instances of some OWL class description by translating the class description to Prolog, but, as reported, the approach does not currently make use of class and property axioms. (Grosof et al. 2003) show how a description logic, now called DLP (Description Logic Programs), can be converted into function free definite Horn Clauses and hence can be evaluated either using logic programming or Datalog systems, the latter being tailored for large sets of facts.

However, when the instance store is frequently updated, as it will be when used to record the dynamic beliefs of an agent, the pragmatics of consistency maintenance of such beliefs becomes an issue. This is especially true when forward inferencing with stored conclusions has been used, which is common for agent reasoning systems in order to support fast selection of an action or plan response (Nilsson 2001). One of the widely accepted hallmarks of agency is reactivity (Jennings et al. 1998).

Suppose that an agent is told some `NewF` that is inconsistent, given the ontology axioms, with a fact `OldF` it already believes about some individual. Does it reject `NewF`, or does it delete `OldF` and any other stored fact it might have inferred and stored using the ontology axioms and `OldF`? To handle these issues rationally, the agent needs to record justifications for all its dynamic beliefs. Then, if `NewF` comes from a source it deems more reliable than the source of `OldF`, it should remove the latter, and use techniques from truth maintenance systems (TMS) (Doyle 1979) to find other beliefs that should also be removed if `OldF` is removed by associating justifications for the inferred recorded facts. This is the approach we adopt.

Volz's thesis (Volz 2004), which considerably extends the work presented in (Grosof et al. 2003), deals with such consequential changes for Datalog rule sets. Instead of a TMS, he uses a rule set derived from the original rules to find the previously inferred and recorded facts that must be deleted when either a given fact or rule is deleted. (Broekstra and Kampman 2003) treats consistency maintenance for an RDFS ontology (Brickley and Guha 2004). Like us, they use a TMS. As we shall see, a key advantage of using a TMS is that the justifications can be used to decide between alternative ways of preserving consistence.

For concreteness, we shall describe the implementation of an ontology constrained belief store in a multi-paradigm, multi-threaded symbolic programming language `Go!` (Clark and McCabe 2004). It is a class based object oriented language combining logic, functional and imperative programming features. Our use of `Go!` is not essential, any other language offering Prolog style inference and dynamic updates of facts could have been used, but we do believe it is important to address the issue of implementation. We have used `Go!` as it has been expressly designed to be an agent programming language.

In a previous paper (Clark and McCabe 2006), we have explored using OWL like ontological concepts to shape the way knowledge is represented as a hierarchy of `Go!` classes echoing the ontology class hierarchy. Following Goldman (Goldman 2003), we call this approach *ontology oriented programming.* It is illustrated by the two example class definitions of section 2.2. However, this approach does not allow the ontology being used to be quickly changed. For agents that may need to process information from the semantic web that conforms to some formal ontology, or which need to communicate with other agents using such an ontology, having a generic agent architecture which we can specialise with different ontology axioms gives flexibility.

This paper explores the representation and use of the axioms and descriptions of the individuals of an ontology expressed in the DL-Lite (Galvanese et al. 2005b) subset of OWL DL, for use within a rational agent. We consider DL-Lite rather than OWL DL, as DL-Lite is a tractable subset (Grau 2006) of OWL DL specifically designed to allow efficient implementation of A-Box inferences using OWL-QL style queries. We shall use the abstract syntax for DL-Lite given in (Grau 2006), rather than the description logic syntax of (Galvanese et al. 2005b), as the abstract syntax axioms map almost directly into `Go!` syntax. We consider just the one ontology language as the focus of this paper is the issues raised balancing consistency maintenance of a dynamic agent belief store, where the consistency constraints are axioms of an ontology, with efficient inference of extra beliefs that we can infer using these axioms. For this purpose, consideration of one representative ontology was deemed sufficient. In addition, by restricting ourselves to the axiom scheme of DL-Lite we shall see that the belief store consistency maintenance problem, although non-trivial, is tractable. It offers a viable approach for the implementation of an ontology constrained agent belief store, or a knowledge store server that pairs DL-Lite ontology axioms with a dynamic fact base.

In the next section we introduce the `Go!` language and its labeled class notation. In section 3 we give a generic class definition for recording the descriptions of DL-Lite individuals as objects, and we show how we can build a layer of higher level relations and procedures for deductively accessing and manipulating these objects. Using `Go!`'s logic programming rules we can define more general n-ary relations in terms of the binary relations (unary class properties) of the base ontology. This gives us expressive power similar to the proposed rule extensions of OWL, such as OWL Rules (Horrocks et al. 2005). In section 4 we discuss the representation of DL-Lite class and property axioms and axioms about the identity of individuals. In section 5 we discuss the use of the axioms to constrain the insertion and manipulation of the individual descriptions and to infer new properties and class memberships for partially described individuals. We shall use a combination of forward and backward chaining deduction, with justifications associated with each dynamically asserted fact. The justifications are used in the consistency maintenance. We do not consider updates of the ontology axioms as, for many agent applications, these will remain fixed for the life time of the agent.

A `Go!` agent using the ontology belief store will typically be multi-threaded (Clark and McCabe 2004), with separate threads dealing with environment monitoring and the consequential belief updates, queries and data from other agents, and the execution of plans to achieve its own goals. We show how an agent might deal with OWL-QL style queries to its belief store in section 6 by defining an interpreter for such queries.

Even though the `Go!` language uses negation-as-failure(Clark 1978), which assumes closed world semantics of its logic rules, our OWL-QL query evaluation using the individual descriptions and the DL-Lite axioms does not use the negation-as-failure inference rule. It conforms to the open world semantics of DL-Lite. In the open world semantics, previous answers to queries must remain valid when new facts about individuals are added. It is not valid to draw a conclusion based on the absence of information - the closed world assumption - which is done by the negation-as-failure rule.

In section 7 we conclude and mention other related work.

2 Quick introduction to Go!

`Go!` is a typed, multi-threaded and multi-paradigm programming language comprising Prolog style relation definitions, function definitions and action procedures which can be grouped together into labelled classes.

The threads execute action procedures, querying relations and calling functions as need be. They communicate by atomically updating and accessing shared objects, usually a dynamic relation or a mail box object. A thread may suspend waiting for an update to a shared object, either waiting for a tuple to be added or deleted from a dynamic relation, or waiting for a message to be added to a mailbox. A dynamic relation used in this way behaves as a generalisation of a Linda tuple store (Carriero and Gelernter 1989), as explained in (Clark and McCabe 2004).

2.1 Function and relation rules

Functions are defined using sequences of rewrite rules of the form:

$f(A_1,..,A_k)::Test \Rightarrow Exp$

where the guard *Test* is omitted if not required. The operator `::` can be read as *such that.* Function types are declared using type definitions of the form:

$f:[t_1,..,t_k]\Rightarrow t$

```
sons_of:[symbol]=>list[symbol].
sons_of(P) => {S||parent_of(P,S),male(S)}.

father_of:[family_person]=>family_person.
```

```
 father_of(C)::C.parent(F),F.gender()=male=>F.
```

An expression of the form {Trm||Cond} denotes the list of all the instantiations of Trm, given by the different solutions to Cond.

The two function definitions represent two styles of information representation in Go!. The first, in which properties of individuals are represented as n-ary relations that mention the individual's name - usually a symbol, is the classical logic/Prolog style. The second collects information about an individual into an object O. We then access properties of the individual by querying this object using an O.p(...) form of call. family_person is an object type.

Relation definitions comprise sequences of Prolog-style :- clauses with some modifications – such as permitting evaluable expressions as arguments of conditions in queries and rule bodies, and no cut operator to control backtracking. Instead of the general purpose cut, there are single solution calls, conditionals, and :-- committed choice clauses, which can be read as *iff* rules.

Each relation definition has an associated type declaration of the form:

r:[t_1,..,t_k]{}

An example is the test relation:

```
has_no_daughters:[family_person]{}.
has_no_daughters(P):-
    (P.child(C) *> C.gender()=male).
```

*> is Go!'s *forall.* has_no_daughters(P) holds of a family_person object P if every child of P (if any) has gender male. The relation could also be defined using:

```
has_no_daughters(P):-
    \+ (P.child(C),C.gender()=female).
```

\+ is Go!'s negation as failure operator (Clark 1978).

2.2 Labeled Classes

Below are type definitions defining object type personI and one of its subtypes dancerI, and two labeled classes for objects of these two types.

The first <~ rule defines the personI type. It gives the interface signature for objects of that type. The @= type declaration tells us that a person term with three arguments of specified types is the label of a class that implements the `personI` type. Immediately below is the class with this label. The arguments `Nm`, `Gndr`, `Places` of the label are global variables of the entire class. We can have more than one labeled class implementation for the same object type.

```
gender::=male | female.    type with literals male, female
desire::=wantToDance(symbol) | ...  type with constructors
belief::=haveDancedWithMale(symbol) | ...
personI <~ {gender:[]=>gender.
            name:[]=>symbol. lives:[string]{}}.
person:[symbol,gender,list[string]]@=personI.
person(Nm,Gndr,Places)..{
  gender()=>Gndr.
  name()=>Nm.
  lives(P):-P in Places.
  }.
dancerI <~ personI.
dancerI <~ {desires:[desire]{}. believes:[belief]{}}.
dancer:[symbol,gender,list[string],
        list[desire],list[belief]]@=dancerI.
dancer(Nm,Gndr,Places,_,_) <= person(Nm,Gndr,Places).
dancer(_,_,_,Desrs,Bels)..{
  desires(Des) :- Des in Desrs.
  believes(Bel) :- Bel in Bels.
  }.
```

The `dancerI` type is defined by two `<~` rules. The first says that it is an extension of the `personI` type, hence that `dancerI` is a sub-type of `personI`. The second gives the type signature for two extra relation methods of the `dancerI` type.

The `dancer` class definition is preceded by a `<=` class rule that says that any instance `dancer(Nm,Gndr,Places,_,_)` of the `dancer` class implicitly includes all the definitions of the instance `person(Nm,Gndr,Places)` of the `person` class.

We can create an instance of the `dancer` class, i.e. a `dancerI` object, and query it as follows:

```
 Mary:dancer = dancer('mary',female,["Cardiff"],
                [wantToDance('polka'),wantToDance('jive'),..],
                [haveDancedWithMale('john'),...]).
```

```
Mary.lives(Place)
   has one solution: Place="Cardiff"
Mary.desires(wantToDance(D))
   has solutions: D='polka'[1] D='jive'
Mary.believes(haveDancedWithMale(OthrDncr))
   has a solution: OthrDncr='john'
```

2.3 Go! dynamic relations

In Prolog we can use assert and retract clauses to change the definition of a dynamic relation. In Go!, a dynamic relation is an object with updateable state. It is an instance of a system class with polymorphic type dynamic[T], T being the type of the terms in the extension of the dynamic relation.

The dynamic relations class has procedure methods: add(Trm), for adding Trm to the end of the current extension of the relation, del(Trm) for removing the first term that unifies with Trm, delall(Trm) for removing all terms unifying with Trm. Finally, it has a relation method mem(Trm) for accessing terms in the current extension that unify with Trm.

To allow modification of the beliefs and desires we can redefine the Dancer class to use dynamic relations to store their initial values. Instances of the class are then objects with updateable state.

2.4 Action rules and threads

The locus of activity in Go! is a *thread*; each Go! thread executes an action procedure. Procedures are defined using non-declarative *action* rules. A procedure p is defined by a sequence of action rules of the form:

$p(A_1,..,A_k)::Test \text{ -> } Action_1;...;Action_n$

with an associated type definition of the form:

$p:[t_1,..,t_k]*$

The * signals procedure type. Go! uses ";" rather than "," to separate the actions to emphasise the imperative aspect of the rule. As with function rules, the first action rule for a procedure *p* that matches some call and has its *Test* succeed is used, with no backtracking. Actions should not fail.

[1]Unlike Prolog, Go! does not have a variable name convention. An identifier must always be quoted when used as a symbol unless it has been declared as a literal value of some type, such as the literals male and female of type gender.

The permissible actions of an action rule include: message send and receive, I/O, updating of dynamic relations, hash tables and re-assignable variables, the calling of a procedure, and the spawning of any action, or sequence of actions, to create a new action thread.

3 Representing DL-Lite individuals

In a DL-Lite ontology an individual is described by a individual description fact. Such a fact, called an `individual`, has the abstract syntax:

```
individual::=individual(individualId {type(classId)} {value})
value::= value(individualvaluedpropertyID individualId)
```

Here, the `{...}` brackets indicate a field that can occur any number of times, including zero[2].

Example `individual` facts conforming to this syntax are:

```
individual(mary type(femaleDancer)
           value(wantToDance polka)
           value(haveDancedWithMale bill))
individual(polka type(dance))
```

This describes an individual `femaleDancer` with identifier `mary` which has a values for the properties `wantToDance` and `haveDancedWithMale`. This value of the `wantToDance` property is a `dance` individual with identifier `polka`.

We can represent DL-Lite individual description usings values of the `Go!` `individual` data type:

```
individual::=individual(individualId,list[type],list[value])
individualId::=id(symbol).
type::=type(symbol).
value::= value(symbol,individualId).
```

The optional repetition of the type and value terms is captured by making these lists, which can be empty. We have used symbols wrapped in `id`

[2]DL-Lite syntax given in (Grau 2006) also allows the type of an individual to be defined by a class description, as described in section 4. However, at the cost of introducing new class identifiers and class axioms, we can always re-express an individual description to use just class identifiers. Similarly, it allows the value of a property to be the description of an unidentified individual. Again, we can avoid this at the expense of introducing new individual identifiers with associated descriptions

functors as individual identifiers, but just symbols for property identifiers and class identifiers. However, we could have used a defined type allowing the use of any OWL URI.

The above DL-Lite individual description of `mary` is mapped to the pair of `Go!` terms:

```
individual(id('mary'),[type('femaleDancer')],
         [value('wantToDance',id('polka')),
          value('haveDancedWithMale',id('bill'))]).
individual(id('polka'),[type('dance')]).
```

It is quite straightforward to write a parser in `Go!` using its DCG grammar rules (Pereira and Warren 1980) to map either DL-Lite abstract syntax strings, or the DL-Lite subset of the OWL RDF/XML syntax (Smith et al 2004), into `Go! individual` terms.

`individual` terms could be used as the stored terms of a dynamic relation to represent the beliefs of an agent. But this would not allow us to update the values of individual properties of a description without replacing the entire description. In order to efficiently update descriptions, we shall instead assume the agent stores them as the state-full objects of the `Go! Individual` class given below. `individual` data terms can still be communicated in messages between agents.

```
Individual_I <~ {class:[type]{}.
                 propVal:[symbol,individualId]{}.
                 addClass:[type]*.
                 addProp:[symbol,individualId]*}.
Individual:[list[type],list[value]]@>Individual_I.
Individual(Ts,Vs)..{
  Classes:dynamic[type]=dynamic(Ts).
  Properties:dynamic[value]=dynamic(Vs).
  class(C):-Classes.mem(type(C)).
  addClass(C) -> (class(T) ? {} | Classes.add(type(C))).
  propVal(P,V) :- Properties.mem(value(P,V)).
  addProp(P,V) ->
   (propVal(P,V) ? {} | Properties.add(value(P,V)).
}.
```

The `type` terms giving classes to which an individual belongs are held in a dynamic relation `Classes`, allowing recorded class membership to be both retrieved and updated. Property values are also stored in a dynamic relation.

Notice that the type declaration for the `Individual` class label uses the `@>` operator rather than `@=`. The use of `@>` declares that the class it labels has update-able state.

There are two procedures: `addClass` and `addProp` for adding a new membership class (a new type) and a new property value, respectively. Their action rules execute a `(Test?Then|Else)` conditional action that does nothing if the class name or property value is already recorded. Notice that the `Classes` and `Properties` dynamic relations are not declared in the interface type. This means that they are private and can only be manipulated by the two procedures `addClass` and `addProp`, and accessed by the relations `class` and `propVal`. A more complete definition would include procedures for removing class names and property values.

We use a global dynamic relation:

```
  Individuals:dynamic[(individualId,Individual)]=dynamic([]).
```

for storing the individual objects paired with their identifiers. We can now define a procedure which takes a term of type `individual` as argument, generates the `Individual` object describing that individual, and inserts it into `Individuals` paired with its indentifier.

```
  new:[individual]*.
  new(individual(Id,Types,Values)) ->
     Individuals.add((Id,Individual(Types,Values))).
```

We can store the description of `mary` using the two procedure calls:

```
 new(individual(id('polka'),[type('dance')],[])
 new(individual(id('mary'),[type('femaleDancer')],
                          [value('wantToDance',id('polka')),
                           value('haveDancedWithMale',id('bill'))]))
```

executed in any order.

3.1 Access relations and procedures

To facilitate access and manipulation of stored descriptions we can define utility relations and procedures:

```
 objectFor:[individualId,Individual]{}.
 objectFor(I,O):-Individuals.mem((I,O)).

 classOf:[individualId,symbol]{}.
```

```
classOf(I,C):-objectFor(I,O),O.class(C).

valueFor:[individualId,symbol,individualId]{}.
valueFor(I,Prop,Val):-objectFor(I,O),O.propVal(Prop,Val).

addPropFor:[individualId,symbol,individualId]*.
addProp(I,P,V) -> objectFor(I,O);O.addProp(P,V).
```

`valueFor(I,Prop,Val)` has several uses. A query:

```
valueFor(id('mary'),Prop,Val)
```

can be used to find each recorded property for `mary` and each of that property's values. A query:

```
valueFor(I,'wantToDance',_)
```

can be used to find the identifiers of individuals that we can infer have at least one value for the `wantToDance` property. More generally:

```
valueFor(I,Prop,Val)
```

can be used to find the identifiers of each of the currently described individuals, the name of each of their defined properties, and values that we can infer they have, or should have, for that property[3].

3.2 Extending the ontology with defined properties

Using `valueFor`, we can use `Go!`'s logic rules to define new properties, even new n-ary relations, over individuals and property names. For example, we can define the ternary relation:

```
shareADanceDesire:[individualId,individualId,individualId]{}.
shareADanceDesire(Dncr1,Dncr2,Dsr):-
  valueFor(Dncr1,'wantToDance',Dsr),
  valueFor(Dncr2,'wantToDance',Dsr).
```

A query:

[3]We shall see later that a class axiom can specify that each individual of some class `c` must have a value for a given property `p`. So, for an `i` that belongs to `c`, even though it might have no recorded value for `p`, we can infer `exists(Val)(valueFor(i,p,Val))`. So, `I=i,Prop=p,Val=...`, where `...` is some term denoting an unknown value, will be an answer to the query `valueFor(I,P,Val)`.

```
shareADanceDesire(id('mary'),id('bill'),Dsr)
```

can be used to find the name of a dance that is a shared desire of `mary` and `bill`. More generally:

```
shareAPropValue:[symbol,individualId,
                individualId,individualId]{}.
shareAPropValue(Prop,Id1,Id2,Val):-
  valueFor(Id1,Prop,Val), valueFor(Id2,Prop,Val).
```

relates two individuals, a property name and common value they have for the property.

By defining auxiliary relations in this way we can build on and significantly enhance the base binary relationships of an DL-Lite ontology whilst retaining efficient evaluation of conjunctive queries involving the defined relations. Providing such definitions do not make use of `Go!`'s `\+` (negation) or `*>` (forall), queries with such defined relations will still be evaluated using the open world semantics.

4 Representing ontology axioms

The above representation of descriptions of the individuals of an ontology is an efficient representation for use by an agent. However, as it stands there is no guarantee that the descriptions conform to the sort of constraints that one can express with an DL-Lite class or property axiom.

For example, we might have DL-Lite axioms:

```
Class(dancer complete person
         restriction(wantToDance someValuesFrom(owl:thing)))
Class(femaleDancer complete dancer female)
Class(maleDancer complete dancer male)
Class(dance partial activity)
Class(male partial person)
Class(female partial person)
DisjointClasses(male female).
ObjectProperty(wantToDance domain(dancer) range(dance))
ObjectProperty(haveDancedWithMale domain(femaleDancer)
                             range(maleDancer)
                             inverseOf(haveDancedWithFemale))
```

The first tells us a `dancer` is defined as a `person` that has at least one value for the `wantToDance` property. The second tells us that a `femaleDancer` is defined as a `dancer` that is `female`, and the third tells us that a `maleDancer` is, by definition, a `dancer` that is `male`. Since both these classes are subclasses of the `dancer` class, the restriction regarding `wantToDance` applies to these two classes as well. The fourth `Class` axiom tells us that a `dance` is an `activity`. The next three axioms tell us that the `male` and `female` are disjoint classes are subclasses of `person`. The first property axiom tells us that `wantToDance` is a property which is restricted to `dancer`s and has range the `dance` class. This axiom together with the `dancer` restriction for `wantToDance` tells us that every `dancer` the `wantToDance` property has at least one `dance` value. (In DL-Lite we cannot directly specify this restriction.) The last axiom tells us that `hasDancedWithMale` relates a `femaleDancer` to a `maleDancer` and that `hasDancedWithFemale` is its inverse. Hence it also tells us that this inverse property relates a `maleDancer` to a `femaleDancer`.

DL-Lite class axioms can be represented as `Go!` facts with the type signatures:

```
Class:[symbol,modality,list[description]]{}.
EquivalentClasses:[list[description]]{}.
DisjointClasses:[list[description]]{}.
SubClassOf;[description,description]{}.
```

where:

```
modality ::= partial | complete.
description ::=  isa(symbol) |  hasAValue(symbol) | nothing
                | intersectionOf(list[description]).
```

A DL-Lite individual valued property axiom can be represented as a `Go!` fact with signature:

```
ObjectProperty(symbol,list[domain],list[range],list[Flag]){}
```

where:

```
Domain ::= domain(symbol).
Range ::= range(symbol).
Flag ::= Functional | inverseOf(symbol) | InverseFunctional.
```

Using this syntax, the above DL-Lite axioms become the following `Go!` facts:

```
Class('dancer',complete,[isa('person'),
                          hasAValue('wantToDance')]).
Class('femaleDancer',complete,[isa('dancer'),isa('female')]).
Class('maleDancer',complete,[isa('dancer'),isa('male')]).
Class('male',partial,[isa('person')]).
DisjointClasses([isa('male'),isa('female')]).
Class('female',partial,[isa('person')]).
Class('dance',partial,[isa('activity')]).
ObjectProperty('wantToDance',[domain('dancer')],
                              [range('dance')],[]).
ObjectProperty('hasDancedWithMale',[domain('femaleDancer')],
                           [range('maleDancer')]
                           [inverseOf('hasDancedWithFemale')]).
```

The list of `description` terms in a `Class` axiom is a list of class descriptions. An `isa(S)` term names a super class `S`, a `hasAValue(P)` denotes the class of things that have at least one value for the property `P` (we use this as a shorthand for the DL-Lite `restriction(P someValuesFrom(owl:thing))` description. `nothing` denotes the empty class and an `intersectionOf` term denotes the class which is the intersection of a list of described classes.

The `partial` modality indicates that class membership constraints of the list of `description` expressions give just necessary conditions for class membership, i.e. the described class is a subclass of the intersection of the classes of the `description` list. The `complete` modality indicates that the axiom gives necessary *and* sufficient conditions for class membership, i.e. the described class is identical to the intersection of the classes of the `description` list. This means that we can use a `complete` class axiom to infer membership of the described class by showing that an individual is a member of all the classes of the `description` list.

At the expense of adding extra `complete` class axioms, we can dispense with both `SubClassOf` axioms and `intersectionOf` descriptions, we can also specify the domain and range of a property using just one `isa` term, and we can ensure that both `EquivalentClasses` and `DisjointClasses` axioms contain only `isa` descriptions.

We can further ensure that the lists of class names appearing in different equivalent axioms are disjoint, and that for each such axiom only *one* of its class names appears in any of the other axioms. This can be achieved by first merging the lists of any equivalence axioms that overlap. We then choose the first class name in each `EquivalenceClass` list as the canonical name, `ConN`, for the equivalent classes of that axiom. For each other `C` appearing in the equivalence axiom, we replace all other occurrences of `C` in the ontology axioms by `ConN`. We shall also assume that there is no `complete` class axiom of the form `class(C1,complete,[isa(C2)])`. This is just alternative way of specifying that `C1` and `C2` are equivalent and can be replaced by such an equivalence axiom.

The following `Go!` relation links a class name to its canonical name.

```
canonicalNameOf:[symbol,symbol]{}.
canonicalNameOf(C,ConN):-
 (EquivalentClasses([isa(ConN),..Cs]),isa(C) in [ConN,..Cs] ?
     true
   | ConN=C).
```

We now redefine the `addClass` method for our `Go! Individual` objects so that only canonical names are stored:

```
addClass(C)->
    canonicalNameOf(C,ConN),
    (class(ConN) ? {}
     |
      Classes.add(type(ConN)).
```

4.1 Individual Identity Axioms

In addition to descriptions of individuals one can also have:

```
SameIndividual   DifferentIndividuals
```

axioms indicating that some set of individual identifiers are all aliases for the same individual, or that some set of names definitely are not aliases. To handle these axioms we modify our `Go!` object representation of the description of an individual. The approach we adopt gives us fast access to all the recorded class and property values of an individual taking into account its aliases, but, just as importantly, it also allows the agent to recover the separate ontology classes and property values of a pair of individuals should it decide to revise its opinion about their identity.

The new `Go! Individual` object for an individual `I` has extra dynamic relations: `diffFrom` and `sameAs`, which record the names of all the individuals believed to be different from or the same as `I`. In addition, the object contains a private re-assignable variable `Alias` with value `this`, the object itself, if `sameAs` is empty. The current value of this variable is returned by the function method `alias()`. There are methods for updating `Alias` and the `diffFrom` and `sameAs` relations.

For an object `O`, representing and individual `I` that has a non-empty `sameAs`, the `Alias` variable holds a different `Individual` object `A`. The objects for all its `sameAs` individuals also have `A` as their `Alias` value. The `type` and `value` facts recorded in `A` are always the union of the corresponding facts in each of the objects of the aliased individuals. `A` itself has no recorded aliases.

The methods for adding new types and property values to `O` also update its `Alias` object, if it is different from `O`. We now redefine the external `classOf` and `valueFor` relations so that they access the `type` and `value` relations in the alias object of an individual `I`, thereby accessing the types and property values for `I` or any of its aliases.

```
classOf(I,C) :-
   objectFor(I,O), O.alias().class(C).
valueFor(I,P,V) :-
   objectFor(I,O), O.alias().propVal(P,V).
```

The external procedure for adding a new identifier `J` to the `sameAs` for an individual `I` also unions the `type` and `value` facts of their respective alias objects to create a new alias object. It then updates both their `Alias` variables to point to this new alias object. `J`'s `sameAs` relation is also updated to record `I` as an extra alias. These single additions to the alias sets of the objects for `I` and `J` and the update of their `Alias` variables to point to a new common alias object are the only changes made to these objects. Deleting an individual `J` from the `sameAs` relation for `I` reverses this operation. A new alias object recording the union of the type and property values for `I` and its remaining aliases is created to become their new `Alias` value. The same is done for `J` and its remaining aliases.

5 Ontology axiom use

Both for answering queries regarding class relationships, and for determining what `hasAValue` restrictions might apply to an individual, our agent will find it useful to be able to access a pre-computed extension of a `subclass` relation that records all the over description term subclass relationships it can infer from the axioms. Similarly, it will be useful for it to have quick access to the extension of a `disjoint` relation, recording each pair of classes that it can infer to be disjoint taking into account the subclass relation and the

`DisjointClasses` axioms. It can generate both these extensions by a using a slight modification of the DL-Lite axiom normalisation process described in (Galvanese et al. 2005a).

The following `normalise` procedure generates the extensions of both relations by forward chaining inferences. The extensions are stored in two dynamic relation objects `SubClass` and `Disjoint`. A pair `(D1,D2)` of inferred disjoint class descriptions is stored only once, `(D2,D1)` is not stored.

```
SubClass:dynamic[(description,description)]=dynamic([]).
subclass:[description,description]{}.
subclass(D1,D2):-SubClass.mem((D1,D2)).
includedIn:[description,description]{}.
includedIn(D1,D2):- (D1=D2|subclass(D1,D2)).
        extends subclass to include pairs of identical terms
Disjoint:dynamic[(description,description)]=dynamic([]).
disjoint:[description,description]{}.
disjoint(D1,D2):-(Disjoint.mem((D1,D2))|Disjoint.mem((D2,D1))).

normalise:[]*.
normalise()-> eval{generateBase();extendUntilClosed()} in
  { noChange:logical:=true. noChange a re-assignable var

    generateBase:[]*.
    generateBase()->
      (Class(C,_,Des),D in Des *> recordSubclass(isa(C),D));
         C is subclass of each descr. in a C axiom
      (Class(C,complete,[hasAValue(P)]) *>
          recordSubclass(hasAValue(P),isa(C)));
          hasAValue(P) same as isa(C), so a subclass
      (ObjectProperty(P,[domain(C)],_,_) *>
          recordSubclass(hasAValue(P),isa(C)));
          hasAValue(P) subclass of P's domain
      (ObjectProperty(P,_,[range(C)],Flgs),
       inverseOf(IP) in Flgs  *>
          recordSubclass(hasAValue(IP),isa(C)));
          hasAValue(IP),IP inverse of P,subclass P's range
      (DisjointClasses(IsaDs),
        D1 in IsaDs,D2 in IsaDs,D1!=D2 *>
          recordDisjoint(D1,D2)).
          record pairs mentioned in a DisjointClasses axiom
    recordSubclass:[description,description]*.
    recordSubclass(D1,D2)::subclass(D1,D2)->{}.
    recordSubclass(D1,D2)->
       SubClass.add((D1,D2));noChange:=false.
       noChange made false if new subclass pair added

    recordDisjoint:[description,description]*.
    recordDisjoint(D1,D2)::disjoint(D1,D2)->{}.
    recordDisjoint(D1,D2)->
       Disjoint.add((D1,D2));noChange:=false.
       noChange made false if new disjoint pair added

     extendUntilClosed()::noChange->{}.
         noChange true, no additions last round, finished
     extendUntilClosed()->
       noChange:=true; noChange was false, reset to true
       (subclass(D1,D2),subclass(D2,D3),D1!=D3) *>
           recordSubclass(D1,D3));
           subclass is transitive
       (subclass(D1,D2),disjoint(D2,D3),D1!=D3) *>
           recordDisjoint(D1,D3));
           subclass of a disjoint class is also disjoint
       (Class(C,complete,Descrs),Des in Descrs,
        includedIn(D,Des),D!=isa(C),
        (OD in Descrs *> includedIn(D,OD))
          *>
           recordSubclass(D,isa(C)));
           any D now included in each descr. of complete
           axiom for some C, is a subclass of isa(C)
       extendUntilClosed().
  }.
```

`subclass` and `disjoint` are the relations used for accessing the extensions of the dynamic relations `SubClass` and `Disjoint`.

The generation of the extensions of the two relations is in two phases. First `generateBase` finds all the immediate `subclass` and `disjoint` relationships, those that are directly inferable from the axioms. Notice that `recordSubclass` only adds a new pair if it is not already recorded, whilst `recordDisjoint` only adds a pair `(D1,D2)` if neither `(D1,D2)` nor `(D2,D1)` is already recorded. When either procedure adds a new pair, its sets the logical variable `noChange` to `false` to signal an update has been made. This variable is shared by both these procedures and `extendUntilClosed`.

The second phase uses the iterative `extendUntilClosed` procedure to compute the closure of these base extensions. It uses one step transitivity and `complete` class axioms to find new subclass relationships, and it uses the `subclass` and `disjoint` extensions so far computed to infer new `disjoint` pairs. The first time `extendUntilClosed` is called `noChange` will be `false` because subclass and disjoint relationships will have been recorded by `generateBase`. It is reset to `true` at the start of each round of extensions. If it does not get set to `false` during an extension round, no new subclass or disjoint relationships can be inferred, so `extendUntilClosed` terminates, otherwise it attempts another round of extensions.

Using the `Class` and `DisjointClasses` axioms we gave in section 4, the normalise procedure generates extensions for `SubClass` and `Disjoint` such that:

```
subclass(isa('male'),isa('person'))
subclass(isa('female'),isa('person')
subclass(isa('dancer'),isa('person'))
subclass(isa('dancer'),hasAValue('wantToDance'))
subclass(isa('maleDancer'),isa('male'))
subclass(isa('maleDancer'),isa('dancer'))
subclass(isa('maleDancer'),isa('person'))
subclass(isa('maleDancer'),hasAValue('wantToDance'))
... simlarly for femaleDancer
subclass(isa('dance'), isa('activity')).
disjoint(isa('male'), isa('female')).
disjoint(isa('maleDancer'), isa('female')).
disjoint(isa('maleDancer'), isa('femaleDancer')).
disjoint(isa('femaleDancer'), isa('male')).
```

If we include the property axioms given earlier we also get:

```
subclass(hasAValue('wantToDance'),isa('dancer'))
subclass(hasAValue('wantToDance'),isa('person'))
subclass(hasAValue('hasDancedWithMale'),isa('femaleDancer'))
subclass(hasAValue('hasDancedWithMale'),hasAValue('wantToDance'))
...
disjoint(hasAValue('hasDancedWithMale'),isa('male')).
disjoint(hasAValue('hasDancedWithMale'),isa('male')).
```

```
  disjoint(hasAValue('hasDancedWithFemale'),
           hasAValue('hasDancedWithMale')).
  ...
```

If we include the axioms:

```
  Class('student',partial,[isa('person'),hasAValue('college')]).
  Class('dancingStudent',complete,
             [isa('student'),hasAValue('wantToDance')]).
```

we get:

```
  subclass(isa('student'),isa('person'))
  subclass(isa('student'),hasAValue('college'))
  subclass(isa('dancingStudent'),isa('student'))
  subclass(isa('dancingStudent'),hasAValue('wantsToDance')).
  ...
  subclass(isa('dancingStudent'),isa('dancer'))
```

`subclass(isa('dancingStudent'),isa('dancer'))` can be inferred in two ways. Firstly, by transitivity, `isa('dancingStudent')` is a subclass of `hasAValue('wantToDance'))`, which is a subclass of the domain `dancer` of `wantToDance`. Secondly, using the `complete` class axiom for `dancer`, `isa('dancingStudent')` is a subclass of each of the description classes of this class axiom. This second inference holds even if we remove the property axiom for `wantToDance` giving its domain.

We assume that our agent calls `normalise` as part of its initialisation. It can then use `subclass` and the `disjoint` to see if there is any subclass pair that is inferable disjoint. It does this by evaluating the set expression:

```
  DisjSubs={(C1,C2)||subclass(C1,C2),disjoint(C1,C2)}
```

For the axioms given so far this set is empty, but suppose that in addition to the `Class` axioms for `student` and `dancingStudent` we added the axiom:

```
  DisjointClasses([isa('dancer'),isa('student')])
```

`DisjSubs` will now contain:

```
 (isa('dancingStudent'),isa('student'))
 (isa('dancingStudent'),isa('dancer'))
 (isa('dancingStudent'),hasAValue('wantsToDance'))
```

So, as described, `dancingStudent` can have no members - the only way it can be a subclass of another class and yet share no members with that class.

5.1 Forward vs backward chaining inference

When we use the ontology axioms for inference with respect to individual description, there are two ways they can be used. We can use them in a forward chaining way to find and record implied class memberships and property values for other individuals immediately some description is added or modified, or we can delay doing the inferences until we need to answer queries. If the former, when the agent adds its description of `mary`, with value `john` for its `hasDancedWitMale` property where `john` has no recorded description, it also adds a description for `john` with membership class `maleDancer` and the value `mary` for the `hasDancedWithFemale` property. If the latter, we only add `mary`'s description, and, if the agent needs to check if `john` is a `maleDancer`, it searches its recorded descriptions to see if `john` is the value of a property with range `maleDancer`.

We believe the following are reasonable choices for the efficient implementation of an DL-Lite ontology conforming belief store in a language such as `Go!`:

Backward chaining inferences

- An individual's membership of a class implied by the ontology axioms, and its recorded types and property values, is inferred whenever needed, and not remembered.

Forward chaining inferences

- Whenever an hitherto un-described individual is named as a property value a new description object for the individual is created.
- When an individual `I` has a new value `J` for a property `P` stored, and `P` has an inverse `IP`, the agent ensures that the value `I` for `IP` is stored in the description of `J`.
- A new class membership implied by a property range is inferred and recorded, if need be, when the property value is stored.
- The identity of individuals implied because a property is functional, or inverse functional, is inferred as the new property value is stored.
- Class memberships or property values implied by the identity of two individuals are inferred when the identity first becomes known. Combined values are stored in an alias object as described in section 4.1.

- When a new `type(C)` class membership is recorded for an individual `I`, and `subclass(isa(C),hasAValue(P))` holds, a special value `some(I,P)` is added for property `P` of `I`, if not already recorded.

5.2 Rules used for backward chaining inference

Below is the definition for a new relation `ClassOf` that extends the `classOf` relation defined earlier. `classOf` just accesses the recorded types of an individual. `ClassOf` makes use of ontology axioms. It also handles the special `some` property values that are added by forward chaining inference. We have changed the `individualId` type to include these terms.

`ClassOf` uses two intermediary relations. First, `baseClassOf` extends `classOf` to cover `some` terms and to make use of `complete` class axioms. `canonicalClassOf` then extends this to cover super-classes as recorded in the `subclass` relation. These will all be canonical class names. `ClassOf` covers class name aliases for these canonical names.

```
individualId::=id(symbol) | some(symbol,symbol).

ClassOf:[individualId,symbol]{}.
ClassOf(Id,C) :-
   canonicalClassOf(Id,ConN),canonicalNameOf(C,ConN).

canonicalClassOf:[individualId,symbol]{}.
canonicalClassOf(Id,C):-
   baseClassOf(Id,BC),(C=BC|subclass(BC,isa(C))).

baseClassOf:[individualId,symbol]{}.
baseClassOf(id(I),C):-
   classOf(id(I),C).
baseClassOf(some(_,P),R):-
   ObjectProperty(P,_,[range(R)],_).
baseClassOf(id(I),C):-
   Class(C,complete,Descrs),
   (isa(C) in Desrcs *> canonicalClassOf(id(I),C)),
   (hasAValue(P) in Desrcs *> valueFor(id(I),P,_).
```

The last clause of the `baseClassOf` allows is to infer that an individual is a member of a class `C` if we can infer that it is also a member of every class mentioned in the list of classes of a complete class axiom for `C`. For the class axiom:

```
Class('dancer',complete,[isa('person'),
                         hasAValue('wantToDance')])
```

its use is equivalent to having an extra rule:

```
baseClassOf(id(I),'dancer'):-
   canonicalClassOf(id(I),'person'),
   valueFor(id(I),'wantToDance',_).
```

So, suppose our agent believes only that `bill` is a `person` and is then told that `bill` has a value `polka` for `wantToDance` . It can now infer that `bill` is a `dancer`.

Such implicit rule extensions to `baseClassOf` are equivalent to some of the rules generated by the mapping from DLP to Datalog in (Grosof et al. 2003). The DL-Lite `complete` axiom for `dancer` cannot be used in DLP. The class would have to be described using two axioms:

```
Class('dancer',partial,[isa('person')]).
SubClassOf(intersectionOf(isa('person'),
                          hasAValue('wantToDance')),
              isa('dancer')).
```

These will be mapped into the two Datalog rules:

$\texttt{person(I)} \leftarrow \texttt{dancer(I)}$
$\texttt{dancer(I)} \leftarrow \texttt{person(I)} \wedge \texttt{wantToDance(I,D)}$

We do not need the equivalent of the first rule as the `dancer`, `person` subclass relationship is computed by the `normalise` procedure and stored as a `subclass` pair. This will accessed by the `canonicalClass` definition allowing the agent to infer that an individual described as a `dancer` is also a `person`. Our implicit extension of `baseClassOf` is equivalent to the second rule.

5.3 Role of `some` terms

The storing of a `some(I,P)` term as a value of a property `P` for individual `id(I)`, if `subclass(isa(C),hasAValue(P))` holds for one of the recorded types `C` of `P`, enables us to correctly answer a query about `id(I)` that asks if it has a `P` value. For, even when there are no explicitly identified individuals (denoted by an `id` term) recorded as `P` values, we will have the `some` term stored. In addition, if a property axiom tells us that `P` has range `C`, it will enable us to always answer queries that ask if `I` has a `P` value that belongs to `C`, or any other class of which `C` is a subclass.

For example, our dancer ontology says that `dancer` is a `person` that has a value for the `wantToDance` property, which is declared to have range `dance`. Now suppose the agent believes that `jim` is a dancer but does not yet know the identities of any dance `jim` might want to do. It will still get `id('jim')` as an element of the query set:

```
{I || valueFor(I,'wantToDance',D),ClassOf(D,'dance')}
```

This is because `jim` will have the term `some('jim','wantToDance')` as a recorded value for its `wantToDance` property, and

```
ClassOf(some('jim','wantToDance'),'dance')
```

holds because of the range declaration for `wantToDance`.

If the agent only wanted the names of individuals who wanted to do some identified dance, it can use the set expression:

```
{I || valueFor(id(I),'wantToDance',id(DId)),
      ClassOf(id(DId),'dance'))}
```

To find all the `dancer` individuals that as yet have no `identified` dance desire, it can use:

```
{I || ClassOf(id(I),'dancer'),
      \+ valueFor(id(I),'wantToDance',id(_))}
```

This is, of course, *not* the same as the set of dancers who have no dance desire, as, by the definition of the `dancer` class, there are *none*. However, for an agent this query can be used as a precursor to an information gathering activity in which the names of the dance desires for these dancers are sought. In OWL-QL(Fikes et al. 2003), similar distinctions regarding known and unknown values for properties can be made by specifying `must-bind`, `may-bind` and `don't-bind` modes for the answer variables.

5.4 Checking consistency of individual descriptions

When an individual description is about to be added to the agent's belief store, or a stored description is modified, the description is checked to see that is consistent with the axioms of the ontology.

The description of an individual `I` is deemed to be *acceptable* if, after replacing each of its class names by their canonical names, its description satisfies the following:

- Its alias and difference sets are disjoint.
- No two of its recorded or inferable types are for `disjoint` classes.
- For each property `P` of `I`:
  - If `P` is functional, *either* all `P`'s values are already recorded as aliases, *or*, merging their descriptions would produce an acceptable description.
  - If `P` is inverse functional with a value `J`, and `J` already has a value `I'` for the special property `inverseP`, then *either* `I` and `I'` are already aliases *or* the merger of their descriptions is acceptable.
  - In addition, any implied new property values for other individuals leaves their descriptions acceptable. For example, if `I` has `value(P,J)`, where `P` has an inverse `IP`, adding `value(IP,I)` to `J` will leave its description acceptable. Likewise, if `P` has range `R`, adding `type(R)` to `J` would leave it acceptable.

5.5 Forward chaining inferences and justifications

- A new description for an individual `I` can be added to the agent's object store providing it is acceptable. When it is added:
  - Each of its class names is replaced by its canonical name
  - If some property `P` of `I` is such that `P` has a `range(C)`, then each value `J` of `P` has an implied `type(C)`. If `type(C)` is already a recorded class of `J`, then `(I,value(P,J))` is added as an extra justification. If it is not yet recorded, nor inferable from `J`'s existing description, the agent modifies its description of `J` to include `type(C)`, with this as initial justification.
  - The descriptions of individuals referenced as values of each property that has an inverse, are updated. If the implied value is already recorded, just an extra appropriate justification is added.

  - If a recorded `type(C)` for `I` is such that for some property `P` `subClass(isa(C),hasAValue(P))` holds, `value(P,some(I,P))` is added to `I` as a reminder of this restriction, with `type(C)` as justification.
  - If a property `P` of an individual `I` is inverse functional with value `J`, and no property axiom gives the name of the inverse of `P`, we ensure that `value(inverseP,I)` is recorded in object `J`. We add `(I,value(P,J))` as a justification.
  - Individuals inferred as being the same because of functionality or inverse functionality of some property, that are not already recorded as aliases, are aliased as described at the end of section 4. Justifications are added recording the reason, or the extra reason for the identification.

- A new membership class `C` can be added to a description providing the description remains acceptable. If this is a subclass of `hasAValue(P)`, for some `P`, the appropriate `some` value is added as a value for `P` .

- A new value for a property `P` can be added to a stored description for individual `I` providing the description remains acceptable. Other updates may be made as when a new individual description is added.

- Each individual property value, class membership or identification with another individual, stored as a result of an observation or a message from another agent, has the source of that belief recorded as a justification.

5.6 Using the justifications

The justifications provide a mechanism for garbage collecting stored conclusions when all the forward chaining inferences that generated the justifications rely on facts that have been deleted. They also help an agent to decide how to revise its beliefs should it be given information that is inconsistent with current beliefs. For example, suppose our agent `Ag` is told that some individual `I` is the same as `J` by an agent `AgSus` with suspect reliability. Further suppose that `I` and `J` have different recorded values `I'` and `J'` for some property `P`, axiomatized in the ontology as functional, and that both these values for `P` came from reliable sources, or observations by `Ag`. Let is further suppose that `I'` and `J'` are believed to be different - each has the other in its `diffFrom` set either because of an ontology axiom or information from a reliable source. Recording that `I` and `J` are aliases would require that either the agent to drop one of these `P` values, or to assume that `I'` and `J'` are not different but are actually aliases. `Ag` may prefer to reject the information that `I` and `J` are the same individual from `AgSus`.

Now, suppose the agent wants to remove a belief `B` either because it has observed that it is false or been told this by a reliable source. It does forward inference from `B` using the ontology axioms as described in section 5.5 to find each consequential belief `CB` that it might have been inferred and stored when `B` was added. It removes each justification for `CB` containing `B`. It will remove `CB` if its justification set is now empty. Before removing `CB` it must recursively apply the same justification pruning process starting from `CB`.

As an example, suppose `B` is the belief `value(P,J)` about an individual `I` where `P` has range `C`. It will remove `B` from the justification set that `I` has `type(C)`.

The above prunes the belief store of those beliefs that can be inferred by forward chaining using `B`. The agent must also ensure that `B` *cannot* be re-inferred. This will usually lead to the pruning of more beliefs that can be determined as false.

Thus, when it is removing `B` from justification sets by doing forward inference from `B`, it must also check that each consequential belief `CB` does not similarly imply `B`. If it does, `CB` must also be removed, even if it has other justifications. `B` being the value of a property with an inverse, and `CB` being the stored value for the inverse property, is an example.

Finally, if any `B` that is deleted has associated justifications, the agent must remove at least one belief mentioned in each these justifications. As an example of this last step, suppose `B` is the belief that some individual `J` has `type(C)` and this has `(I,value(P,J)` as justification, added because `P` has range `C`. The agent must also remove `value(P,J)` from its beliefs about `I`.

The removal of a belief deemed to be false typically results in the examination and removal of other beliefs directly and indirectly linked to it. At least at the end of this recursive removal of justifications and beliefs the set of recorded individual descriptions will be consistent with the ontology axioms and reflect what the agent considers to be a true state of affairs. A prior checking of what belief store revisions will result from the removal of a belief that an agent considers *may* be false can be used to gather and weigh evidence for the belief. As a result it may decide to leave the belief in its store. How such pragmatic decisions are made will be application dependent and is beyond the scope of this paper.

6 External Queries

Internally an agent can use the `ClassOf` and `valueFor` relations to query its belief store. But it is useful for agent's to be able to query each others beliefs, to ask questions of one another. We can envisage that inside each agent is a dedicated query answering thread accepting queries from other agents. We use a query syntax loosely based on OWL-QL (Fikes et al. 2003).

Queries are terms of type:

```
answerTerm::=i(individualId) | p(symbol).
query::=all(list[answerTerm],list[queryCond]).
queryCond::= holds(symbol,individualId,individualId) |
             typeOf(individualId,symbol).
```

OWL-QL uses `type` instead of `typeOf` and does not use `holds` to prefix property value conditions. Instead it uses just an unlabelled 3-tuple. The `p` answer terms will contain property names.

The query term:

```
all([i(I),i(J)],[holds('child',I,J),typeOf(J,'male')])
```

corresponds to the `Go!` set expression:

```
{[i(I),i(J)] || valueFor(I,'child',J),ClassOf(J,'male')}
```

The agent evaluates a query term by invoking the function `evalQ`:

```
evalQ:[query]=>list[answerTerm].
evalQ(all(AnsTms,QConds)) => {AnsTms||evalConds(QConds)}.

evalConds:[list[queryCond]]{}.
evalConds([]).
evalConds([Cond,..RConds]):-
  evalCond(Cond),
  evalConds(RConds).
evalCond(typeOf(I,C)):-ClassOf(I,C).
evalCond(holds(P,I,V)):-valueFor(I,P,V).
```

Two more example query terms are:

```
all([i(J)],[holds('child',id('bill'),J),holds('wife',J,_)])

all([i(I),p(P)],[typeOf(I,'person'),holds(P,I,id('bill'))])
```

The first is to find all `bill`'s children that have wife. The second is to find each person `I` and that is related to `bill` by a property `P`, and the name of the property.

We can easily generalise our `query` and `queryCond` types, extending `evalQ` appropriately, to allow single solution queries, disjunctive query conditions, and conditional queries. If an agent has auxiliary relations defined using relation rules, like the `shareADanceDesire` relation of section 3, we can also allow external queries to access these by extending the `queryCond` type to include query conditions of the form:

```
rel(symbol,list[individualId])
```

To evaluate such conditions, we simple add extra rules to the `evalCond` definition, one for each defined relation that external queries can use. For example:

```
evalCond(rel('shareADanceDesire',[D1,D2,Dnce])):-
   shareADanceDesire(D1,D2,Dnce).
```

7 Related Work and Final Remarks

7.1 Agents using formal ontologies

Regarding the use of the axioms of a formal ontology within an agents belief store we believe that (Barbuceaneau and Fox 1996) describes the first agent architecture to adopt this approach. The paper describes an agent building shell in which the agent's beliefs can be represented using a description logic managed using a truth maintenance system. Details are not given.

More recently, the EU Agent Cities project (Willmott et al. 2001) used information agents that converted theatre and restaurant information from web pages into an ontology instance store represented as RDF triples. Two ontologies, `Shows` and `Restaurant` (AgentCities 2003), expressed in DAML/OIL were used. The instance stores were then queried by personal agents using FIPA ACL (FIPA 2002) messages. However, the ontology axioms were not used to check for consistency of the restaurant and shows data as it was collected and stored - instead web scraper agents were hand programmed to generate only correct descriptions with respect to the ontology. In addition, when the information was queried, only the axioms defining the subclass hierarchy of the ontology were used to reason about class memberships. So the inference layer was quite weak.

GraniteNights (Grimnes et al. 2003), another information agent application, similarly only makes use of RDF triples to encode the instance store data about restaurants and shows in Aberdeen. It also only does inference using RDFS sub-class axioms.

In (Chen et al. 2005) a broker agent supporting user agents in a ubiquitous computing environment is described. It uses an OWL based ontology describing the location structure of the physical environment, the types, roles and BDI (Beliefs, Desires, Intentions) structures of the user agents, and policy issues. The broker agent stores and makes available shared location and activity information about the users. It also uses the ontology to check for the consistency of information the user agents provide and to augment this information in answering their queries. The information is stored on a relational data base and accessed using a rule inference engine implemented in Java. This is a real application of the approach advocated in this paper but details of the inference rules used to support reasoning using the ontology axioms, and the consistency checking, are not given.

Another interesting application making use of an ontology to aid reasoning is its proposed use in (Schenoff et al 2004) within an autonomous vehicle as an aid to perception. The intention is to use an ontology based on description logic to describe obstacles that may be encountered in the path of the vehicle. The ontology will then be used to classify an obstacle from its perceived features.

(Alechina et al. 2006) contains a description of a proposed belief revision algorithm for Jason, a Java implementation of an extension of the BDI agent language AgentSpeak(L)(Rao 1996). The belief store comprises a set of atomic beliefs. Beliefs are added or removed either as a result of a message, agent observations, or forward inference plans that are triggered by belief update events. The algorithm makes use of justifications and preferences. The preferences are used to determine which belief to drop from a justification set for an inferred belief when the belief is deemed to be false. Which of a set of alternative beliefs to drop to maintain consistency is an issue we have not dealt with.

7.2 Multi-agent truth maintenance

We have only considered the revision of ontology facts to maintain consistency with the axioms within a single agent. However, we have assumed that agents will query one another and perhaps directly transfer beliefs using KQML style (Finin et al. 1994) `tell` and `deny` messages. Keeping track of which beliefs were used to answer every OWL-QL query an agent has received, so that a new reply may be sent if the answer changes due to changes in these beliefs, is a costly proposition. KQML does have the notion of a subscription query. This is a persistent query that should be re-answered each time a different answer will be given, and an agent may or may not support that type of query.

On the other hand, if an agent directly transfers a belief to another agent using a `tell` message, which it later decides to delete from its won belief store because it considers it to be false, it might want to update all the agents to which the belief has been told by sending them a `deny` message. This leads us into the realm of multi-agent truth maintenance, as treated in (Huhns and Bridgeland 1991).

The consequential revisions of the belief stores of the agent's receiving the denials may generate the need for them to send out other denials, and successive rounds of communication may be needed before the combined beliefs of the various agents become consistent with their respective ontology axioms. We assume that these axioms, if different, are collectively consistent. Doing this in an efficient manner which is guaranteed to terminate is an issue we have not yet considered.

7.3 Ontologies with rule defined relations

We have shown how `Go!`'s logic rules can be used to extend the range of ontological relationships that can be expressed. There are several proposals to augment description logic based ontologies with rules.

We have already mentioned the work of (Grosof et al. 2003) which maps a description logic into Datalog. (Volz 2004) shows how more expressive description logics can be mapped into Datalog extended with equality and constraints.

The proposed SWRL (Semantic Web Rule Language) (Horrocks et al. 2005) is an extension of OWL DL to include a Horn clause rule language.

WRL (Web Rule Language) (de Bruijn 2005), which builds upon F-Logic (Kifer et al. 1995), a frame based logic programming language, is another recent proposal to complement OWL with rules.

Flora-2 (Yang et al. 2003), is another extension of F-Logic that can be used for rule based ontological knowledge representation. Like `Go!`, Flora-2 is a OO logic programming language with multiple inheritance. It also includes transaction logic rules for specifying updates. In Flora-2 one could implement an agent with an ontology constrained belief store. This is also true of any Prolog or Prolog extension.

7.4 Final Remarks

The `Go!` definitions of sections 3, 4, 5 give the flavour of the way in which we can use individual descriptions of some DL-Lite ontology, augmented and constrained by the axioms of the ontology, as a `Go!` agent's belief store. We have also seen how we can further augment the ontology using `Go!` relation definitions that build upon the binary relations of the ontology.

Our view is that the use of formal ontologies to frame the beliefs of an agent is an idea whose time has come. As we have argued in this paper, this means that we must also address the issue of belief revision to maintain consistency with the ontology because it is the essence of an agent that its contingent beliefs about the state of the world are in flux. As another key requirement of an agent is reactivity - the timely response to changes in its environment - the inferences must also be performed in a timely fashion.

We have outlined the implementation of a concrete approach to handling dynamic beliefs about individuals of an ontology expressed in DL-Lite. It balances gaining faster inference by sometimes doing forward chaining, with recorded conclusions, with the resultant cost of having to use a justification based truth maintenance system to be able to efficiently maintain ontological consistency of the belief store when it is modified. We have not dealt with the formal aspects of the soundness or completeness of our implementation of a DL-Lite reasoner in `Go!` as our primary concern in this paper was the practical issues concerning use of a simple ontology framework inside a reactive, rational agent.

We intend to test the feasibility of our approach by using such an ontology constrained belief store in an existing `Go!` implemented multi-threaded BDI agent architecture in which plans are an elaboration of Nilsson's Teleo-Reactive programs (Nilsson 2001).

References

AgentCities (2003). Shows and Restaurant Ontologies. Technical report, www-agentcities.doc.ic.ac.uk/Services/ontologies.html.

Alechina, N., R. Bordini, J. Hubner, M. Jago, and B. Logan (2006). Belief Revision for Agentspeak Agents. In *Proceedings of DALT Workshop at AAMAS-06 Conference.*

Baader, F., I. Horrocks, and U. Sattler (2004). Description logics. In S. Staab and R. Studer (Eds.), *Handbook on Ontologies*, International Handbooks on Information Systems, pp. 3–28. Springer.

Barbuceaneau, M. and M. Fox (1996). The Architecture of an Agent Building Shell. In *Intelligent Agents II*, pp. 235–250. Springer-Verlag, LNAI, Vol 1037.

Bechhofer, S., I. Horrocks, and D. Turi (2005). The OWL instance store (system description). In *Proc. of the 20th Int. Conf. on Automated Deduction (CADE-20)*, Lecture Notes in Artificial Intelligence. Springer. To appear.

Brickley, D. and R. Guha (2004). RDF Vocabulary Description Language 1.0 RDF Shema. W3C recommendation, http://www.w3.org/TR/rdf-schema.

Broekstra, J. and A. Kampman (2003). Inferencing and Truth Maintenance in RDF Shema. In *Proceedings of First WS on Practical and Scalable Semantic Systems*. http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-89/.

Carriero, N. and D. Gelernter (1989). Linda in context. *Communications of the ACM 32*(4), 444–458.

Chen, H., T. Finin, and A. Joshi (2005). The SOUPA Ontology for Pervasive Computing. In *Ontologies for Agents: Theory and Experiences*, pp. 233–258. BirkHauser.

Clark, K. L. (1978). Negation as failure. In H. Gallaire and J. Minker (Eds.), *Logic and Databases*, pp. 293–322. Plenum press.

Clark, K. L. and F. G. McCabe (2004). Go! – a Multi-paradigm programming language for implementing Multi-threaded agents. *Annals of Mathematics and Artificial Intelligence 41*(2-4), 171–206.

Clark, K. L. and F. G. McCabe (2006). Ontology oriented programming in Go! *Applied Intelligence 24*(3), 3–37.

de Bruijn, J. (2005). Web Rule Language (WRL), version 1.0. W3C member submission, http://www.w3.org/Submission/WRL/.

Doyle, J. (1979). A truth maintenance system. *Artificial Intelligence 12*(3).

Fikes, R., P. Hayes, and I. Horrocks (2003). OWL-QL - A Language for Deductive Query Answering on the Semantic Web. SL Technical Report 03-14, http://ksl.stanford.edu/KSL_Abstracts/KSL-03-14.html.

Finin, T., R. Fritzson, D. McKay, and R. McEntire (1994). KQML as an agent communication language. In *Proceedings 3rd International Conference on Information and Knowledge Management.*

FIPA (2002). Fipa communicative act library specification. Technical report, Foundation for Intelligent Physical Agents, www.fipa.org.

Galvanese, D., G. de Giacome, D. Lembo, M. Lenzerini, and R. Rosati (2005a). DL-Lite: Tractable Description Logics for Ontologies. In *Proceedings of 20th National Conference on Artificial Intelligence*, pp. 602–607. AAAI.

Galvanese, D., G. de Giacome, D. Lembo, M. Lenzerini, and R. Rosati (2005b). Tailoring OWL for Data Intensive Ontologies. In *Proceedings of First OWL:Experiences and Directions*. http://www.mindswap.org/2005/OWLWorkshop.

Goldman, N. (2003). Ontology oriented programming - static typing for the inconsistent programmer. In *The Semantic Web, Proceedings of ISWC 2003*, Sanibel Island, Florida. Springer-Verlag, LNAI, Vol 2870.

Grau, B. (2006). Tractable Fragments of the OWL 1.1 Web Ontology Language. Technical report, http://owl-workshop.man.ac.uk/Tractable.html.

Grimnes, G. A., S. Chalmers, P. Edwards, and A. Preece (2003). Granitenights - a multi-agent visit scheduler utilising semantic web technology. In *Seventh International Workshop on Cooperative Information Agents*, pp. 137–151.

Grosof, B., I. Horrocks, R. Volz, and S. Decker (2003). Description Logic Programs: Combining Logic Programs with Description Logics. In G. Hencsey and B. White (Eds.), *Proc. of the WWW-2003*.

Horrocks, I., P. F. Patel-Schneider, S. Bechhofer, and D. Tsarkov (2005). OWL rules: A proposal and prototype implementation. *J. of Web Semantics 3*(1), 23–40.

Horrocks, I., P. F. Patel-Schneider, and F. van Harmelen (2003). From $\mathcal{SHIQ}$ and RDF to OWL: The making of a web ontology language. *J. of Web Semantics 1*(1), 7–26.

Huhns, M. and D. Bridgeland (1991). Multiagent truth maintenance. *IEEE Transactions on Systems, Man and Cybernetics 21*(6), 1437–1445.

Huhns, M. and M. Singh (1997). Ontologies for agents. *IEEE Internet Computing* (Nov-Dec), 81–83.

Jennings, N., K. Sycara, and M. Wooldridge (1998). A Roadmap of Agent Research andf Ddevelopment. *Autonomous Agents and Multi-agent Systems 1*(1), 7–38.

Kifer, M., G. Lausen, and J. Wu (1995). Logical foundations of object-oriented and frame-based languages. *Journal of the ACM 42*, 741–843.

McGuinness, S. and F. van Harmelen (2004). Owl Web Ontology Language - Overview. W3C recommendation, http://www.w3.org/TR/owl-features.

Nagy, Z., G. Lukcsy, and P. Szeredi (2006). Translating Description Logic Queries to Prolog. In *Proceedings of PADL 06.* to appear in LNCS, Springer-Verlag.

Nilsson, N. (2001). Teleo-reactive programs and the Triple Tower Architecture. *Electronic Transactions on Artificial Intelligence 5, Section B*, 99–110.

Pereira, F. and D. H. Warren (1980). Definite clause grammars compared with augmented transition network. *Artificial Intelligence 13*(3), 231–278.

Rao, A. S. (1996). AgentSpeak(L): BDI agents speak out in a logical computable language. In *Agents Breaking Away*, LNAI 1038, pp. 42–55. Springer-Verlag.

Schenoff et al, C. (2004). Using ontologies to aid navigation planning in autonomous vehicles. *The Knowledge Engineering Review 18*(3), 243–245.

Smith et al, M. (2004). Owl web ontology language guide. W3C recommendation, http://www.w3.org/TR/owl-guide/.

Volz, R. (2004). *Web Ontology Reasoning with Logic Databases.* Ph. D. thesis, University of Karlsruhe.

Willmott, S. N., J. Dale, B. Burg, C. Charlton, and P. O'Brien (2001, November). Agentcities: A Worldwide Open Agent Network. *Agentlink News* (8), 13–15.

Yang, G., M. Kifer, and C. Zhao (2003). Flora-2: A rule-based knowledge representation and inference infrastructure for the semantic web. In R. King, M. Orlowska, and R. Studer (Eds.), *Proceedings on Ontologies, Databases and Applications of Semantics'03, LNAI 2888*, pp. 671–688. Springer Verlag.