Ontology schema for an agent belief store

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

F.G. McCabe
fgm@fla.fujitsu.com

January 5, 2006

Abstract

In this paper we explore the use of a formal Owl 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 considerable enrich the ontology with rule defined n-ary relations and functions. These rules combined with the 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, and with logic programming.

1 Introduction

Ontology specification languages such as Owl[22] are based on description logics[18] for which there are fast stand-alone inference procedures. However,

to our knowledge little work seems to have been done regarding the use of a formal ontology as the framework for the dynamic beliefs of an agent, the dynamic beliefs being the descriptions of individuals belonging to the classes of the ontology.

In description logic terms, inferences regarding the relationships between classes of an ontology are called T-box inferences, whereas those concerning descriptions of individuals are A-box inferences[2]. 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, particularly agents that retrieve and process information from the semantic web, efficient A-box reasoning is as important, if not more so. Proposed query languages for Owl, such as Owl-QL[11], 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[4] 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 individual description but it does not, as we understand it, support full Owl-QL queries. Moreover, when such an 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[1].

Suppose that an agent is told something that is inconsistent, given the ontology axioms, with a fact `F` it already believes about some individual. Does it reject the new information, or does it delete `F` and any other asserted fact `F'` it might have inferred using the ontology axioms that depends on `F`? To handle these issues rationally, the agent needs to record justifications for all dynamic beliefs and use something like a truth maintenance system[10] to maintain consistency when the beliefs are updated.

For concreteness, we shall describe the implementation of such an ontology constrained belief store in a multi-paradigm, multi-threaded symbolic programming language `Go!` [7]. It is a class based object oriented language combining logic, function and imperative programming features. It is typed and has multiple inheritance. In a previous paper[8], we have explored using Owl ontological concepts to shape the way the knowledge is represented as a hierarchy of `Go!` classes echoing the ontology class hierarchy. Following

[1] We shall assume that only the descriptions of individuals will change - that the class and property descriptions of the ontology remain fixed.

[14], 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 ontologies to be accessed and used dynamically. For agents that may need to process information from the semantic web that conforms to some Owl ontology, or which need to communicate with other agents using such an ontology, being able to dynamically load and use a new set of ontology axioms gives flexibility.

This paper explores the representation and use of the axioms and descriptions of the individuals of an Owl Lite ontology in `Go!` for use within a multi-threaded rational agent. We consider Owl Lite rather than Owl DL as the former has nearly all the expressive power of the latter[18] and is sufficient for our purposes. We also use the abstract syntax for Owl Lite[22], rather than the RDF/XML syntax[21], as Owl abstract syntax axioms map almost directly into `Go!` syntax.

Each Owl Lite class axiom becomes a `Go!` fact describing the named class. Each property axiom similarly becomes a `Go!` fact about the named property. Each Owl Lite fact, describing some individual, is then mapped into an instance of a single generic class. This has a method that gives access to the names of the ontology classes to which the individual has been declared or inferred to belong, and a generic method for accessing the current values of each of its properties. The generic class has methods for adding and deleting property values, and for updating an object's membership classes. Such dynamic manipulation is essential to allow for updating of an agent's beliefs.

In the next section we introduce the `Go!` language and its labeled theory/class notation. In section 3 we give the generic class definition for recording the descriptions of Owl 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[17]. In section 4 we discuss the representation and use of Owl Lite class and property axioms. 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.

A `Go!` agent using the ontology belief store will be multi-threaded. We

sketch the implementation of one such thread that can answer Owl-QL style queries to its belief store in section 6. In section 7 we conclude and mention some related work.

2 Quick introduction to Go!

`Go!` has many features in common with McCabe's *L&O*[20] object oriented extension of `Prolog`. A key feature of both languages is the ability to group a set of definitions into a lexical unit called a labeled theory. A labeled theory is essentially a class definition. Each instantiation of the theory label denotes an instances of the class - an object. The theory definitions are the object methods. Separate class rules allow new classes to be defined as extensions of existing classes - inheriting all their methods unless overridden in the new class. Multiple inheritance is allowed.

`Go!` differs from *L&O* in being typed, in having action procedures as well as functions and relations, and in being multi-threaded. A class implements a separately declared interface type. This interface type is the type signature of the visible methods of the class.

Threads execute procedures, querying relations and calling functions as need be. They communicate by atomically updating and accessing shared objects, usually a mail box object. A thread may suspend waiting for an update to a shared object. A dynamic relation used in this way behaves as a generalisation of a Linda tuple store[5], as explained in [7].

2.1 Function and relation rules

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

f`(`A_1`,..,`A_k`)::`*Test* `=>` *Exp*

where the guard *Test* is omitted if not required. The operator `::` can be read as *such that*.

For each function there must also be an associated type definition of the form:

f`:[`t_1`,..,`t_k`]=>`t

where t_i is the type of the i'th argument and t is the type of the value. These must all be data types. `Go!` is not higher order but we can program in a higher order way by passing in and returning object values.

As in most functional programming languages, the testing of whether a rule can be used to evaluate a function call uses *matching* not unification.

Once a function rule has been selected there is no backtracking to select an alternative rule.

Example function definitions with associated type definitions are:

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

father_of:[Person]=>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 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` about that individual. We then access properties of the individual by querying this object using an `O.p(...)` form of call. Note that both function definitions are preceded by a type declaration. For the second function, `person` is an object interface type to be discussed below.

Relation definitions comprise sequences of Prolog-style `:-` clauses with some modifications – such as permitting evaluable expressions as well as data terms, and no cut. We can also define relations using `:--` 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:

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

`*>` is `Go!`'s *forall*. This defines a property that holds of a `person` object if every child they have (if any) has gender `male`. The `+` annotation on the `person` type is a mode declaration. It says that the argument will be given

and will be an object of type `person`, or of type that is a declared sub-type of `person`. A `-` annotation signals output mode which requires the call argument to be an unbound variable. The default (no annotation) mode for a relation argument is that it can be input or output. The default mode for an function and action rule argument is that it is input.

2.2 Labeled Theories

Below is set of type definitions defining object type `person` and one of its sub-types `dancer`, and two labeled classes which are mini-theories for objects of these two types.

```
sex::= male | female.    new type with literals male, female
dance::= polka | jive | ..    another enumerated type
desire::= toDance(dance,integer) | ...  type with constructors
belief::= hasDesire(symbol,desire) |
          hasDanced(symbol,dance) | ....

person <~ {gender:[]=>sex. name:[]=>symbol. lives:[string]{}}.
person:[symbol,sex,list[string]]$=person.

person(Nm,Sx,Places)..{   labeled theory
  gender()=>Sx.
  name()=>Nm.
  lives(P):-P in Places.
  }.

dancer <~ person.
dancer <~ {desires:(desire){}. believes:(belief){}}.
dancer:[symbol,sex,list[string],list[desire],list[belief]]$=
            dancer.

dancer(Nm,Sx,Places,_,_) <= person(Nm,Sx,Places).
dancer(_,_,_,Desires, Beliefs)..{
  desires(Des) :- Des in Desires.
  believes(Bel) :- Bel in Beliefs.
  }.
```

The first `<~` rule defines the `person` type. It gives the interface signature for objects of that type. The first `$=` rule declares that the theory with the `person` label, which takes three parameters of types `symbol`, `sex` and

list of `string`s respectively, is an implementation of this type. The theory is labeled `person(Nm,Sx,Places)`. The arguments `Nm`, `Sx`, `Places` are parameters to the labeled theory which, when known, make it the theory of a specific person. We have overloaded `person` and used it as both the interface type name and the functor of the class label. This is allowed but not required. We could have used something like `personC` for the class label.

The `dancer` type is defined by two `<~` rules. The first says that it is an extension of the `person` interface type, hence that `dancer` is a subtype of `person`. The second gives the type signature for two extra relation methods of the `dancer` type. The `dancer` class definition is preceded by a `<=` class rule that says that any instance `dancer(Nm,Sx,Places,_,_)` of this labeled theory implicitly includes all the definitions of the instance `person(Nm,Sx,Places)` of the `person` theory.

We can create two instances of these theories, i.e. two objects, and query them as follows:

```
Psn = $person('bill',male,["London","New York"]).
Dncr = $dancer('mary',female,["Cardiff"],
              [toDance(polka,1),toDance(jive,2),..],
              [hasDesire('john',toDance(jive,2)),...]).

Psn.name()          returns name 'bill'[2] of Psn
Dncr.lives(Place)   gives one solution: Place="Cardiff"
Dncr.desires(toDance(D,1))    has a solution: D=polka
Dncr.believes(hasDesire(Dncr,toDance(jive,N))),N>0
                                has a solution: Dncr='john'
```

2.3 `Go!` dynamic relations and hash tables

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 interface type `dynamic[T]`, `T` being the type of the argument of the dynamic relation. All `Go!` dynamic relations are unary, but the unary argument can be a tuple of terms.

[2]Note that `'bill'` is singly quoted. This is because, unlike Prolog, `Go!` does not have a variable name convention. Identifiers beginning with upper *and* lower case letters can be used as variable names. Such an identifier must therefore be quoted when used as a symbol unless it has been declared as a literal value of some type such as `dance`.

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

A *dynamic relation* object can be created and assigned to a variable using:

```
beliefs:dynamic[belief] =
     $dynamic([hasDesire('mary',toDance(polka,2)),....])
```

The `:` type annotation declares that `beliefs` is a `dynamic` relation object with argument terms of type `belief`.

Hash tables are similar to dynamic relation objects but in addition the stored terms are indexed by a numeric or symbolic key. They are objects of the polymorphic system class `dynamic[KT,VT]`, `VT` being the type of the values stored and `KT` being the type of the keys. They have a relation method `present(K,V)` for retrieving a value given its key, a function method `keys()` for returning a list of the current key values, and procedure methods `insert(K,V)` and `delete(K)` for changing entries. We can use a hash table to record beliefs about some individual identified by its unique name used as a key. We can even wrap up all the data about that individual in an object, storing that with the key.

```
dancerTbl = $hash([]).

dancerTbl.insert('mary',
   $dancer('mary',female,["Bath"],[toDance(polka,1),..],..));

dancerTbl.present('mary',O),O.hasDesire(Dsr)
         has first solution Dsr=toDance(polka,1)
```

To allow efficient update of the `dancerTbl` entries we could redefine the `dancer` class to use dynamic relations to store the beliefs and desires of the agent. Instances of the class are then objects with updateable state. Our generic object class of section 3 uses a dynamic relation to store property values.

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 o the form:

p(*A*$_1$,..,*A*$_k$)::*Test* -> *Action*$_1$;...;*Action*$_n$

with an associated type definition of the form:

p:[t_1,..,t_k]*

The * signals procedure type. We use ":" 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.

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

The following two action rules define a procedure for displaying the details of a named dancer object stored in `dancerTbl`.

```
display_dancer:[symbol]*.
display_dancer(DncrNm)::dancerTbl.present(DncrNm,Dncr) ->
   stdout.outLine("Dancer "<>DncrNm.show()<>
                  " believes "<>{B||Dncr.believes(B)}.show()<>
                  " desires " <>{D||Dncr.desires(D)}.show()).
display_dancer(DncrNm) ->
   stdout.outLine("Dancer "<>DncrNm.show()<>
                     " not yet described").
```

`stdout` is a `Go!` system object with various methods for sending strings to the standard output channel. `<>` is a polymorphic primitive for concatenating lists of any values. (`Go!` strings are lists of single character symbols.) `show` is a default function which converts any value into a string. It can be overridden in a class definition. The second rule is used only if a dancer object with the name `DncrNm` is not in `dancerTbl`.

3 Representing descriptions of Owl individuals

In an Owl Lite ontology an individual is described by a individual description fact. Such a fact, called an `individual`, has the abstract syntax[22]:

```
individual::= Individual([symbol] {symbol} {value})
value::= value(individualvaluedpropertyID symbol)
value::= value(individualvaluedpropertyID individual)
value::= value(datavaluedpropertyID dataLiteral)
```

Here, the `[..]` brackets indicate an optional field and the `{...}` brackets indicate a field that can occur any number of times, including zero[3].

An example `individual` fact conforming to this syntax is:

```
Individual(mary dancer
                value(hasDesire
                            Individual(dance
                                   value(kind polka)
                                   value(howmany 2)))).
```

This describes an individual `dancer` with identifier `mary` which has one value for the individual valued property `hasDesires`. This value is an unnamed `dance` individual with two data valued properties, a `name` with value `polka` and a `howmany` property with value `2`.

In our representation in `Go!` we shall shall not distinguish between the individual valued property names and data valued property names but instead tag the property values so that we know what type the value is. We shall also assume that all individuals have identifiers (we can generate a new identifier if they have not). This means that we do not need to allow individual property values to be individuals, they can always be individual identifiers. We shall allow strings, symbols and numbers as data values. So a property value can be represented using the `Go!` data type:

```
propertyValue::=
    id(symbol) | sym(symbol) | str(string) | int(number).
```

We can then represent Owl Lite individuals using the `Go!` data type:

```
individual::=Individual(symbol,list[symbol],list[Value])
Value::= value(symbol,propertyValue).
```

The optional repetition of the class identifiers and value terms is captured by making these lists, which can be empty. We have used `Go!` symbols as individual and property identifiers, but we could have more `symbol` and `symbol` types allowing the use of any Owl URI. We shall also use symbols as class identifiers, but these could also have a more complex term representation.

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

[3]Owl Lite syntax also allows a membership class of an individual to be defined by a property restriction, as described in section 4, as well as by a class identifier. 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

```
Individual('mary',['dancer'],[value('hasDesire',id('dance1)]).
Individual('dance1',['dance'],
                    [value('kind',sym('polka')),
                     value('howmany',int(2))])).
```

It is quite straightforward to write a parser in `Go!` using its DCG grammar rules[23] to map either Owl Lite abstract syntax strings or its RDF/XML syntax[24] into such `Go!` terms. The range of data values can be extended to cover all the XML data types allowed in the RDF/XML syntax as special categories of strings.

`Go!` terms of type `individual` could be used as the stored terms of a dynamic relation to represent the beliefs of an agent. But this would not give us efficient access to the description of an individual given its name, nor will it allow us to update the values of individual properties of a description without replacing the entire description. To get fast access and the ability to efficiently update descriptions, we need to store the descriptions in a hash table indexed by the identifiers, with the descriptions themselves being updatable objects of a generic `IndividualObject` class. `individual` data terms can still be communicated between agents.

```
IndividualObject <~ {class:[symbol]{}. property:[symbol]{}.
                     propVal:[symbol,propertyValue]{}.
                     property:[symbol]{}.
                     addClass:[symbol]*.
                     addProp:[symbol,propertyValue]*}.
IndividualObject:[list[symbol],list[Value]]$=IndividualObject.

IndividualObject(Classes,Vals])..{
  Class:dynamic[symbol]=$dynamic(Classes).
  Properties:hash[symbol,list[propertyValue]]=
                              $hash(condense(Vals)).
  class(C):-Class.mem(C).
  addClass(C) -> (class(C) ? {} | Class.add(C)).
  property(P):- P in Properties.keys().
  addProp(P,V)::Properties.present(P,Vals) ->
     (V in Vals ?  {}
     | Properties.delete(P);Properties.insert(P,[V,..Vals])).
  addProp(P,V) ->   rule for when P not present
     Properties.insert(P,[V]).
  propVal(Prop,Val)::nonvar(Prop) :--
     properties.present(Prop,Vals),Val in Vals.
```

```
    propVal(Prop,Val)::var(Prop) :--
       property(Prop),
       properties.present(Prop,Vals),Val in Vals.
  }.
```

The membership classes of the individual are represented as a dynamic relation `Class`, allowing recorded class membership to be both retrieved and updated. The list of all the current values for a property are stored in a hash table using the property name as key. The relation method `property` can be used to test or find the current defined property names. There are two procedures: `addClass` and `addProp` for adding a new membership class and a new property value. Notice that the `Class` dynamic relation and `Properties` hash table are not declared in the interface type. This means that the dynamic relation and hash table are private and can only be manipulated and accessed by the two procedures `addClass` and `addProp`, and the relations `class`, `property` and `propVal`. A more complete definition would include procedures for removing class names and property values. Notice that `propVal` is defined using two committed choice rules as they cover disjoint cases.

If we use a hash table:

```
IndividualTbl:hash[symbol,IndividualObject]=$hash([]).
```

for storing the individual objects indexed by their identifers, we can define a procedure which takes a term of type `individual` as argument, generates the object describing that individual, and inserts it into `IndividualTbl` using its indentifier.

```
new:[symbol,list[symbol],list[Value]]*.
new(Individual(Id,Classes,Values)) ->
   IndividualTbl.insert(Id,
               $IndividualObject(Classes,condense(Values))).
```

We must `condense` the list `Values` of `value` terms to collect together all the values of a given property. Thus, `condense` will map:

```
[value('likes', id('bill')), value('likes',id('mary'))]
```

into:

```
[('likes', [id('bill'),id('mary')])].
```

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

```
new(Individual('dance1',['dance'],
                              [value('kind',sym('polka'),
                               value('howmany',int(2))])));
new(Individual('mary',Individual('mary',['dancer'],
                         [value('hasDesire',id('dance1'))]))).
```

executed in any order.

To facilitate access and manipulation of stored descriptions we can define utility relations and procedures with the following types:

```
objectFor:[symbol,IndividualObject]{}.

classOf:[symbol,symbol]{}.

propValOf:[symbol,symbol,propertyValue]{}.

addPropFor:[symbol,symbol,propertyValue]*.
```

`objectFor` can be used to retrieve an object given its identifier or to find each individual identifier and its associated object.

`classOf` can be used to test whether or not an individual is a recorded member of some class, to find each of its recorded classes, or to find all the individuals recorded as being a member of some class.

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

```
propValOf('mary',Prop,Val)
```

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

```
propValOf(I,'hasDesires',_)
```

can be used to find the identifiers of individuals with at least one value for the `'hasDesires'` property. More generally:

```
propValOf(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 the values for each property.

`addPropFor` can be used to add a new value to a named property of an identified individual. It does nothing if the property value is already stored, or if the identifier is not linked with an object. An example use is:

```
addPropFor('mary','hasDanced',id('polka'));
addPropFor('mary','hasDancedWith',id('john'))
```

Using `propValOf`, we can also define new properties, even new n-ary relations, for the described individuals.

```
wantsToDance:[symbol,symbol,integer]{}.
wantsToDance(Dncr,DnceKind,HowMany):-
  propValOf(Dncr,'hasDesire',id(Dnce)),
  propValOf(Dnce,'kind',sym(DnceKind)),
  propValOf(Dnce,'howmany',int(HowMany)).
```

This can be used by an agent to infer that a dancer identified by `Dncr` wants to do the `DnceKind` kind of dance `HowMany` times.

Given the stored information about `'mary'`,

```
wantsToDance('mary','polka',2)
```

can be inferred.

We can now define:

```
shareADanceDesire:[symbol,symbol,symbol]{}.
shareADanceDesire(Dncr1,Dncr2,DnceKind):-
  wantsToDance(Dncr1,DnceKind,HowMany1),HowMany1>0,
  wantsToDance(Dncr2,DnceKind,HowMany2),HowMany2>0,
  Dncr1!=Dncr2.
```

for finding a shared dance desire of two different dancers, if one exists. A query:

```
shareADanceDesire('mary',Dncr,DnceKind)
```

can be used to find the name of another dancer `Dncr` sharing a desire to dance a `DnceKind` dance with `'mary'`. By defining auxiliary relations in this way we can build on and significantly enhance the base binary relationships of an ontology.

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 Owl Lite class or property axiom.

For example, we might have Owl Lite axioms (using its abstract syntax):

```
Class(dancer partial person
            restriction(hasDesires allValuesFrom(dance)))
Class(dance partial restriction(kind cardinality(1))
                    restriction(howmany cardinality(1)))
ObjectProperty(hasDancedWith domain(dancer) range(dancer)
                             Symmetric)
```

The first imposes the constraint that in any individual description of a dancer every value of the `hasDesire` property (if any) is an individual from the `dance` class. It also tells us that a `dancer` is a subclass of the `person`, so constraints on property values for a `person` also apply to a `dancer`. The second axiom constrains a `dance` description to have exactly one value for its `kind` and `howmany` properties. The third tells us that `hasDancedWith` has range and domain the `dancer` class and is symmetric.

In its essentials, an Owl Lite class axiom can be represented as a `Go!` fact with the type signature:

```
Class(symbol,modality,list[Super])
```

where:

```
modality ::= partial | complete.
Super ::= isa(symbol) |
          restriction(PropertyId,Restriction).
Restriction ::= someValuesFrom(symbol) |
                allValuesFrom(symbol) |
                minCardinality(zeroOrOne) |
                maxCardinality(zeroOrOne) |
                Cardinality(zeroOrOne).
zeroOrOne ::= zero | one.
```

The list of `Super` terms is a list of class descriptions - either a named class or the class of things that satisfy a given property restriction. The `partial` modality indicates that class membership constraints of the list of `Super` expressions give just necessary conditions for class membership, i.e. the described class is a subclass of the intersection of the classes of the `Super` 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 `Supers` 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 `Super` list.

An Owl Lite individual valued property axiom can be represented as a `Go!` fact with signature:

```
ObjectProperty(PropertyID,list[SuperProp],list[Domain],
                    list[Range], list[PropType])
```

where:

```
SuperProp ::= super(PropertyID).
Domain ::= domain(symbol).
Range ::= range(classsID).
PropType ::= Functional | Symmetric | Transitive |
             inverseOf(PropertyId) | InverseFunctional.
```

There is similar form of fact for an axiom for a data valued property. The difference is that range identifiers have to be data set identifiers and the only `PropType` value allowed is `Functional`.

Using this syntax, we get represent the above Owl Lite axioms as the following `Go!` facts:

```
Class('dancer',partial,[isa('person'),
           restriction('hasDesires',allValuesFrom('dance'))]).
Class('dance',partial,[restriction('kind',cardinality(one)),
           restriction('howmany',cardinality(one))]).
ObjectProperty('hasDancedWith',[domain('dancer')],
                            [range('dancer')],[Symmetric]).
```

In addition to descriptions of individuals one can also have:

```
SameIndividuals   DifferentIndividuals
```

facts indicating that some set of individual identifiers are all aliases for the same individual, or that some set of names definitely are not aliases. We shall assume that these facts are part of the agent's dynamic beliefs, along with the descriptions of individuals. We store the non-alias and alias information for an individual `I` in its object `O`. Each such `O` now has two extra dynamic relations: `diffFrom` and `sameAs` which store the names of individuals believed to be different from or the same as `I`. The closure of the `samesAs` relation for `I` is the *alias* set for `I`. The union of the extensions of the `diffFrom` relations for the individuals in the alias set of `I` is the *difference* set for `I`. The alias and difference sets for an individual must always be disjoint. We could now extend the `propValOf` relation so that it also looks up property values in each member of the alias set for `I`, and similarly extend the `classOf` relation. An alternative, is to union property values and class memberships for an alias set each time it is updated and record

these separately from the property values and classes of each individual in the set. That is, we union all the attributes of the individuals in an alias set each time one is created or modified, perhaps storing the results in an alias object table indexed by a new unique identifier `A`. `A` is then be remembered as an extra attribute for each individual of the alias set.

We must of course redefine `classOf` and `propValOf` so that they call the corresponding relations in the alias object for an individual `I` when `I` has aliases. We must also redefine the object methods updating the recorded classes and properties so that they perform the same update on the alias object, if there is one[4].

The advantage of this approach is that we have fast access to all the classes and property values of an individual taking into account its aliases, but the agent can still recover the separate membership classes and property values of a pair of individuals should it decide to revise its opinion about their identity.

5 Ontology axiom use

Let us suppose that we have a set of facts encoding class and property axioms of some Owl Lite ontology. We can make use of them to check updates and to make inferences by modifying the definitions for the procedures for `new` and `addPropFor`, and the property value accessing relation `propValOf`.

Constraints on properties of a class apply to all its subclasses. Similarly for properties and their subproperties. So we need to define relations that can be used to traverse this hierarchy. Here is the `superClass` definition:

```
Isa(C,S) :-
     Class(C,_,Supers),
     isa(S) in Supers.

superClass(C,S):-Isa(C,S).
superClass(C,S):-Isa(C,S1),superClass(S1,S).
```

There are similar definitions for `subClass`, `superProperty` and `subProperty`. Using `superClass` we can define the relation `RestrictionOfClass(C,R)` that holds when `R` is a property restriction for class `C` or one of its super classes:

[4]The new objects representing individuals can be defined as a sub-type of `IndividualObject` of section 3. The alias table will just contain `IndividualObject`s.

```
RestrictionOfClass(C,R) :-
     Class(C,_,Supers),
     restriction(R) in Supers.
RestrictionOfClass(C,R) :-
     superClass(C,S),
     Class(S,_,Supers),
     restriction(R) in Supers.
```

We can likewise define the relations:

```
DomainOfProperty  RangeOfProperty  TypeOfProperty
```

which can be used for finding or checking all the declared and inherited domain, range and type constraints for a property, or for finding properties with a given domain, range or type.

We can use the ontology axioms both for data validation and for inference. Thus, suppose an agent is sent a description of a dancer `'mary'` with an `'hasDancedWith' 'john'` property value where it has no other information about `'john'`. If this description comes from a reliable source, it might accept the description and use it to infer that `'john'` identifies another `'dancer'` with a property `'hasDancedWith' 'mary'`. Alternatively, if the source of the information about `'mary'` was suspect, it might query an alternative source to check that `'john'` identifies an individual which is a `'dancer'`, rejecting the information about `'mary'` if this is not confirmed.

When we use the ontology axioms for inference, there are two ways they can be used. We can use them in a forward chaining way to infer and record 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 inserts its description of `'mary'` it immediately adds another description for the hitherto unknown `'john'` with membership class `dancer` and a property value `'hasDancedWith' 'mary'`. If the latter, we only add `'mary'`'s description but if the agent is asked whether `'john'` is a `dancer` it uses backward chaining inference to infer this from the recorded `'hasDancedWith'` property for `'mary'`. For our representation of the individual descriptions in an object table indexed by individual identifiers, doing forward chaining with recorded conclusions is much more efficient. When we are checking if `johh'` is a dancer, it saves us having to search descriptions of all other dancers to see if they have a property that relates them to `'john'` which implies that he is a `'dancer'`.

There is however a disadvantage to recording inferred conclusions. An agent can be requested, or decide of its own accord, to remove a belief.

(Agent communication languages, such as KQML [12], usually allow `untell` and `deny` messages.) If it removes some belief `TB` it needs also to remove all stored inferred beliefs that depend upon `TB`. Conversely, if it deletes an inferred belief `IB`, it also needs to delete at least one of the inferred or told beliefs from which `IB` has been inferred. This is the classic AI *truth maintenance* problem of inference systems that store results of inferences *and* allow revision of beliefs.

Following Doyle[10], we can handle it by associating with every belief `B`, the different subsets of beliefs about individuals from which it has been inferred using the ontology axioms. Each such subset `J` of beliefs is a *justification* for `B`. Initial beliefs, and beliefs that are representations of information the agent has been told or acquired through observation, can have an initial justification indicating the source of the information. We do not need to remember which ontology axioms are used as we have assumed they are never revised.

Now, suppose the agent decides to remove a belief `B`. It does forward inference from `B` using the ontology axioms and its other beliefs to find each belief `IB` that can be inferred from `B`. It removes each justification for `IB` containing `B`. It deletes `IB` if it now has an empty justification set. It now examines the justifications for `B` if this was itself inferred. It must ensure that `B` cannot be re-inferred by removing at least one belief from each of its justifications. The removal of beliefs directly linked to `B` will typically result in the examination and removal of other beliefs indirectly linked to it.

Justifications provide a mechanism for garbage collecting stored conclusions that should no longer be inferred. 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` has an alias `J` by an agent `AgSus` with suspect reliability. Further suppose that `I` and `J` have different recorded values for some data valued property `P`, axiomatized in the ontology as functional, and that both these values came from reliable sources or observations by `Ag`. Recording that `I` and `J` are aliases will require the agent to drop one of these `P` values. `Ag` may prefer to reject the alias information from `AgSus`.

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

- Property values implied by values for sub-properties are inferred by backward chaining and not stored.
- Property values implied because the property is transitive are also inferred using backward chaining and not stored.

- An individual's membership of a class that is a super class of one of its recorded classes is inferred using backward chaining, and not stored.
- The description of an individual `I` is deemed to be *acceptable* if its description satisfies the following:
  - Its alias and difference sets are disjoint.
  - If one of its properties `P` is a data valued property then its given values must satisfy all the ontology constraints for `P`. Thus, if `P` is functional, or one of the classes for `I` has a restriction that `P` has maximum cardinality one, `P` can have at most one value.
  - For each object valued property `P` of `I`:
    - If `P` is functional, or a restriction for one of the classes for `I` says that `P` has cardinality or maximum cardinality one, and `P` has more than one object identifier value, *either* all `P`'s values are aliases, *or*, merging their descriptions would produce an acceptable description.
    - If `P` is inverse functional with given value `I'`, and `I'` already has a value `I"` for the property `inverseP`, then *either* `I'` and `I"` are aliases *or* the merger of the descriptions of `I'` and `I"` must be acceptable.
    - In addition, any implied new property values for individuals that are the values of object properties of `I` leaves their descriptions acceptable. For example, if `I` has property value `P I'`, where `P` is symmetric, adding the property `P I` to `I'` will leave its description acceptable.
- A new description for an individual `I` can be added to the agent's object store providing it is acceptable. When it is added:
  - If some property `P` of `I` is such that:
    - `P` has a `range(C')`,
    - `P` is functional and there is a `someValuesFrom(C')` restriction for a class `C` of `I`,
    - there is a `allValuesFrom(C')` restriction for a class `C` of `I`,

    then each value `I'` of `P` has an implied class `C'`. If `C'` is already a recorded class for `I'` then `[(I,P)]`, or `[(I,P),(I,C)]`[5], is added

[5]This indicates that both the belief that `I` has property `P` and the belief that it is a member of class `C` are used in the inference.

as an extra justification. If it is not yet recorded, the agent modifies its description of `I'` to include `C'` as a membership class with the appropriate initial justification.

- The descriptions of individuals referenced as values of each property that is symmetric or has an inverse, are updated, if need be, to record the implied property value with appropriate justifications.
- If one of the classes `C` for `I` has a `someValuesFrom(C')` restriction for a property `P` an extra value `some(C')` is added to property `P` as a reminder of this requirement.
- If there is a class `DefC` with a `complete` axiom such that the stored description of `I` satisfies all the membership constraints for `DefC`, `DefC` is included to the description of `I` as a complete class. A justification is not recorded but the complete class memberships for each individual are rechecked and possibly updated whenever its description is changed.
- If a property `P` of an individual `I` is inverse functional with value `I'` we record `I` in the object for `I'` as a value of `inverseP` if not already recorded.
- The objects for all the individuals inferred as being the same because of functionality, inverse functionality or cardinality constraints not already recorded as aliases are aliased as described at the end of section 4.

- A new membership class `C` can be added to a stored description providing the description remains acceptable. After it has been added, implied updates to descriptions of referenced individuals are made, as when a new individual description is added.

- A new value for a property `P` can be added to a stored description for individual `I` providing the description remains acceptable. If `P` is symmetric, or has an inverse, the implied property for the value `I'` of `P` is added to its description, if need be, with appropriate associated justification..

The above rules correspond to an assumption that information is monotonically added to the belief store and that new information should be rejected if it is detectably inconsistent with existing information and the ontology. This is rather simplistic. As we mentioned earlier, one could also use the recorded justifications to decide, for example, to delete previously

stored information that comes from a less reliable source in favour of new information from a more reliable source in order to maintain consistency.

Here are definitions for two new relations `PropVal` and `ClassOf` that make use of ontology axioms to augment explicitly stored property values and membership classes. We need to extend the `propertyValue` type to allow `some` term values.

```
propertyValue::=
   id(symbol) | some(symbol) |
   sym(symbol) | str(string) | int(integer).

ClassOf:[propertyValue,symbol]{}.
ClassOf(id(I),Class) :-
  recordedClass(I,C),
  (Class=C | superClass(C,Class)).
ClassOf(some(Class),Class).

recordedClass:[symbol,symbol]{}.
recordedClass(I,C):-
  (classOf(I,C) | completeClass(I,C)).

PropHasVal:[symbol,propertyValue,propertyValue]{}.
PropHasVal(P,IdTrm,Val)::TypeOfProperty(P,Transitive):--
  PropHasValExcept(P,IdTrm,Val,[]).
PropHasVal(P,id(I),Val):--
  PropValOf(I,P,Val).

PropHasValExcept:[symbol,propertyValue,propertyValue,
                  list[symbol]]{}.
PropHasValExcept(P,id(I),Val,Seen) :-
  PropValOf(I,P,Val),
  \+ Val in Seen.
PropHasValExcept(P,IdTrm,Val,Seen) :-
  PropValOf(I,P,NxtIdTrm),
  \+  NxtIdTrm in Seen,
  PropHasValExcept(P,NxtIdTrm,[NxtIdTrm,..Seen]).

PropValOf:[symbol,symbol,propertyValue]{}.
PropValOf(I,P,Val) :-
  propValOf(I,P,Val).
```

```
PropValOf(I,P,Val) :-
  subProperty(SubP,P),
  PropValOf(I,SubP,Val).
```

`ClassOf` relates an individual to each of its classes taking into account subclass relationships and recorded class memberships inferred using complete class axioms - the `completeClass` test. `PropHasVal` relates a property name to an individual and a value. Both these relations denote individuals as `id` terms. (We do this in order to be able to more easily handle Owl-QL queries as described in the next section.)

`PropHasValExcepts` recurses through a connected sequence of `P` values that are `id` terms until the sequence ends or cycles. It cycles when an `id(NextI)` is encountered such that `NextI` appears in the `Seen` list, which is incremented each time a next identifier in the sequence is followed. At this point `PropHasValExcept` fails. `\+` is `Go!`'s negation as failure operator[6]. `propValOf` is the relation for accessing recorded property values for an individual whereas `PropValOf` also returns recorded values for sub-properties. Both of these denote an individual just by its symbolic name. We do not need to handle the case of `P` being symmetric as extra values implied by the symmetry property are recorded using forward chaining when a new `P` value is added to an individual's description.

A `some(C')` term will appear as a value for a property for an individual `I` when an ontology class axiom for some class `C` of `I` says that there is some `C'` value for property `P` for individuals of class `C`. By recording the `some(C')` term we shall be able to correctly answer a query about `I` that asks if it has a `P` value from `C'`, even when there are no individuals from `C'` recorded as its `P` values. For example, suppose the ontology says that `'parent'` is a subclass of the `'person'` class and that every instance of the `'parent'` class has a value for its `'offspring'` property that is a `'person'`. Now suppose the agent believes that `'jim'` is a parent but does not know the identities of any offsprings of `'jim'`. It will still get `'jim'` as an element of the query set:

```
{I || PropHasVal('offspring',id(I),OSprg),
      ClassOf(OSprg,'person')}
```

since `'jim'` will have the term `some('person')` as a recorded value for its `'offspring'` property and:

```
ClassOf(some('person'),'person')}
```

holds. If the agent only wanted the names of individuals who have some identified offspring, it can use the set expression:

```
{I || PropHasVal('offspring',id(I),id(OSprgNm)),
      ClassOf(id(OSprgNm),'person'))}
```

To find all the identifiers for `'parent'`s that have no known offspring it can use:

```
{I || ClassOf(id(I),'parent'),
      \+PropHasVal('offspring',id(I),id(_))}
```

6 External Queries

Internally an agent can use the `ClassOf` and `PropHasVal` 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 based on Owl-QL[11]. A query comprises an answer template, which is list of terms of type `propertyValue` - usually variables appearing inside the query body, and a query body, which is a list of query conditions. The answer template and query body are arguments of a term whose functor indicates whether one or all answers are required. The query term type definition is:

```
query::=askall(list[propertyValue],list[queryCond]) |
        askone(list[propertyValue],list[queryCond]).
queryCond::= prop(symbol, propertyValue, propertyValue) |
             type(propertyValue,symbol)
```

The query term:

```
askall([I],[prop('offspring',I,OSVal),type(OSVal,'person')])
```

corresponds to the set expression:

```
{I || PropHasVal('offspring',I,OSVal),ClassOf(OSVal,'person')}
```

When the agent receives a query term it must infer an answer by evaluating the list of conditions of the query body using its `PropHasVal` and `ClassOf` relations. It does this by invoking the function:

```
evalQ(askall(AnsT,QConds)) => {AnsT||evalConds(QConds)}.
evalQ(askone(AnsT,QConds)) => (evalConds(QConds) ? [AnsT] | []).

evalConds([]):--true.
evalConds([Cond,..RConds]):--
  evalCond(Cond),
  evalConds(RConds).
evalCond(type(I,C)):- ClassOf(I,C).
evalCond(prop(P,I,V)) :- PropHasVal(P,I,V).
```

Two more example query terms are:

```
askall([I,A],[prop('offspring',id('bill'),I),prop('age',I,A)])

askone([sym(P)],[prop(P,id('bill'),id('mary'))])
```

The first is to find the names and ages of all `'bill'`'s offspring, The second is to find the name, returned as a `sym` value, of a relationship between `'bill'` and `'mary'`. If there is no such relationship the evaluation of this query term returns the empty list `[]`.

The following is a simple procedure that can be spawned as the agent's external query answering thread:

```
queryAsk::=ask(query,symbol,dropbox[queryReply]).
queryReply::=reply(symbol,list[list[propertyValue]]).
QueryBox:[mailbox[queryAsk]]=$mailbox().

handleQueryMessages() ->
   QueryMBox.next()=ask(Query,QId,ReplyDBox);
   ReplyDBox.post(reply(QId,evalQ(Query)));
   handleQueryMessages().
```

This procedure just handles one message format, an `ask` message. The `next` function call will suspend until such a message appears in the `QueryMBox mailbox` object. As soon as one does appear, it evaluates the embedded query using the `evalQ` function and posts a `reply` message to the `ReplyDBox dropbox` object given for the reply[6]. The query handler then loops to handle the next `ask` message.

[6] A drop box is an associated object for a mail box allowing only insertion of messages and not the reading of messages. The `ask` message will have been posted into a drop box associated with `QueryBox` and made public by our ontology belief store agent.

`QId` is a symbol identifying the query, it is a KQML[12] style `reply-with` label. Note that the mail box and drop box are typed - at compile time there is a check to ensure that only messages of the required type are posted to or extracted from these communication objects.

We can easily generalise our query format to allow disjunctive queries and negated conditions. 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[propertyVal])
```

For example:

```
rel('shareADanceDesire',[id('mary'),DncrId,DnceKndVal])
rel('wantsToDance',[id(I),sym('polka'),IntVal])
```

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

```
evalCond(rel('shareADanceDesire',
                     [id(D1),id(D2),sym(DnceKnd)])):-
   shareADanceDesire(D1,D2,Dnce).
evalCond(rel('wantsToDance',[id(D),sym(DnceKnd),int(N)])):-
   wantsToDance(D,DnceKnd,N).
```

Using the DCG grammar rules[23] of `Go!` and its TCP/IP communication primitives, we can straightforwardly modify our query server to accept queries expressed as XML strings. These would be converted to `query` terms for evaluation by our query interpreter, then converted back to XML string replies.

7 Conclusions and Related Work

The `Go!` definitions of sections 3, 4, 5 give the flavour of the way in which we can use individual descriptions of some Owl Lite ontology, augmented and constrained by 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 property relations of the ontology.

To our knowledge there has been little work to date using a formal ontology as part of an agent's belief store. An exception is [3], which describes

an agent building shell in which the agent's beliefs can be represented using a description logic, very similar in expressive power to Owl, managed using a truth maintenance system. But details are not given.

Mention [25] and [**?**]

In the EU Agent Cities project[26] there were information agents that converted theatre and restaurant information from web pages into ontology instance stores represented as RDF triples. Two ontologies, `Shows` and `Restaurant`[1], expressed in DAML/OIL were used. The instance stores were then queried by personal agents using FIPA ACL[13] 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 the web scraper agents were hand programmed to generate only correct descriptions with respect to the ontology. In addition, when the information was queried, only axioms defining the subclass hierarchy of the ontology were used to reason about class memberships. So the inference layer was quite weak.

GraniteNights [15], another information agent application, similarly only made use of RDF triples to encode the instance store data about restaurants and shows in Aberdeen. It also only did inference using RDFS sub-class axioms.

We have shown how `Go!`'s logic rules can be used to extend the range of ontological relationships that can be expressed. They can be used to define quite general relationships between class properties (the definition of the `wantsToDance` relation being an example) and to define n-ary relations. The ontology community recognizes the benefit of augmenting ontology languages based on description logics with rules[16].

The SWRL language[17] (Semantic Web Rule Language) is an extension of Owl DL to include a Horn clause rule language. Using such as extension, one can augment an Owl ontology with n-ary relations. Here is the abstract syntax version of a SWRL rule that defines the `wantsToDance` relation:

```
Implies(Antecedent(
    hasDesire(i-variable(Dncr) i-variable(Dnce))
    kind(i-variable(Dnce) d-variable(DnceKind))
    howmany(i-variable(Dnce) d-variable(HowMany))
       Consequent(wantsToDance(i-variable(Dncr)
                               d-variable(DnceKind)
                               d-variable(HowMany)))
```

SWRL indicates that a variable ranges over individuals or data values by wrapping it by the functors `i-variable`, `d-variable` respectively.

WRL [9] (Web Rule Language), which builds upon F-Logic[19], a frame based logic programming language, is another recent proposal to complement Owl with rules. In WRL surface syntax the `bothAged` relation is defined by the rule:

```
wantsToDance(?Dncr,?DnceKind,?HowMany) :-
      ?Dncr[hasDesire  hasValue ?Dnce]  and
      ?Dnce[kind  hasValue ?DnceKind]  and
      ?Dnce[howmany  hasValue ?HowMany] .
```

That `hasDesire` is a property of an individual is indicated by the use of the keyword `hasValue`. Juxtaposition of an `hasValue` condition to a variable or name indicates access to a property value of an individual. It is similar to `Go!`'s use of dot as in `Dncr.hasDesire(Dnce)`.

Flora-2[27], is another development 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. Type information, analogous to `Go!`'s interface type declarations, can be asserted as facts. The Flora-2 equivalent of the `Go!` type declaration:

```
person <~ {name:()=>string. age:[]=>integer.
           home:[]=>string. lives:[string]{}}.
```

is the assertion:

```
person[name=>string, age=>integer,
       home=>string, lives=>>string].
```

The `=>>` indicates that the `lives` attribute is multi-valued. The type assertions do not seem to be used for type checking but they can be queried (see below).

As in `Go!`, one can also define attributes using rules. For example, we can state that for any person the value of the `home` property is a value of the `lives` property using the rule:

```
P[lives-->H] :- person::P, P[home->H].
```

The infix `::` is used to indicate class membership and is the equivalent to our `ClassOf` predicate.

Flora-2 has two other components. One is a higher order, or perhaps more accurately a meta-order component, called HLog. This enables one to query an object to find its attribute names, and whether they are single and multiple valued, or to query a class type declaration to find the methods

for that class. The other Flora-2 component is transaction logic rules for specifying updates. Transaction logic rules are similar to `Go!`'s action rules.

In Flora-2, but not in SWRL or WRL, one could implement an agent with an ontology schema belief store. This is also true of any Prolog or Prolog extension.

References

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

[2] F. Baader, I. Horrocks, and U. Sattler. Description logics. In S. Staab and R. Studer, editors, *Handbook on Ontologies*, International Handbooks on Information Systems, pages 3–28. Springer, 2004.

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

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

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

[6] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum press, 1978.

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

[8] K. L. Clark and F. G. McCabe. Ontology oriented programming in Go! *Applied Intelligence, to appear*, 2006.

[9] J. de Bruijn et al. Web Rule Language (WRL), version 1.0. Rule Markup Initiative Technical Report, http://www.wsmo.org/wsml/wrl/wrl.html, 2005.

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

[11] R. Fikes et al. 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, 2003.

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

[13] FIPA. Fipa communicative act library specification. Technical report, Foundation for Intelligent Physical Agents, www.fipa.org, 2002.

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

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

[16] B. Grosof. Description Logic Programs: Combining Logic Programs with Description Logics. In G. Hencsey and B. White, editors, *Proc. of the WWW-2003*, 2003.

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

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

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

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

[21] F. G. McCabe. *L&O: Logic and Objects*. Prentice-Hall International, 1992.

[22] S. McGuiness et al. Owl Web Ontology Language - Overview. W3C candidate recommendation, http://www.w3.org/TR/owl, 2000.

[23] P. F. Patel-Schneider et al. Owl web ontology language - semantics and abstract syntax. W3C Candidate Recommendation, http://www.w3.org/TR/2004/REC-owl-semantics-20040210/, 2003.

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

[25] M. K. Smith et al. Owl web ontology language guide. W3C recommendation, http://www.w3.org/TR/owl-guide/, 2003.

[26] R. Volz, S. Staab, and B. Motik. Incementally Maintaining Materializations of Ontologies Stored in Logic Databases. In *Journal of Data Semantics II*, LNCS, Vol 3360, pages 1–34. Springer, 2005.

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

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