An Initial Response to the OAS'03 Challenge Problem

Ian Dickinson
Hewlett-Packard Laboratories
Filton Road, Stoke Gifford
Bristol BS34 8QZ
U.K.
ian.dickinson@hp.com

Michael Wooldridge
Department of Computer Science
University of Liverpool
Liverpool L69 7ZF
U.K.
m.j.wooldridge@csc.liv.ac.uk

ABSTRACT
We present our initial response to the OAS '03 Challenge Problem. The Challenge Problem proposes an agent-assisted travel scenario, and asks what the role of ontologies would be to support the agent's activity. We discuss a belief-desire-intention (BDI) approach to the problem using our Nuin agent platform, and illustrate various ways in which ontology reasoning supports BDI-oriented problem solving and communications by the agents in the system.

Keywords
Agent applications, BDI agents, Ontology, Semantic web

1. INTRODUCTION
The call for papers for the AAMAS '03 workshop on Ontologies and Agent Systems (OAS'03) includes a challenge problem, adapted from an exercise by the OntoWeb project to assess different ontology environments. The challenge problem outlines a set of objectives for an agent-assisted travel planning system, in which an agent-based travel agent must co-operate with other agents to book a trip for a human client.

We have been investigating the design and development of belief-desire-intention (BDI) [16] agents for use in the Semantic Web [6]. One outcome of this research is a BDI agent platform, Nuin, which has been designed *ab initio* to work with Semantic Web information sources. At the time of writing, Nuin is still a work in progress. Nevertheless, we have investigated how key parts of the OAS challenge problem would be addressed by our platform.

This paper reviews the salient features of the Challenge Problem, in the context of a BDI agent. We briefly review some of the characteristics of the Nuin platform, before presenting a series of vignettes that show how we address some of the challenges in the Challenge Problem. As it represents a rich and plausible scenario, we are continuing development of a complete solution to the Challenge Problem using the Nuin platform.

2. OUTLINE PROBLEM
The scenario for the Challenge Problem is based on a travel agent in New York, for which we are asked to develop an agent-based application. The travel agency's clients come to make bookings for trips they wish to take, and the agency is responsible for making reservations with various travel service providers (airlines, hotels, train companies, etc) to satisfy the client's needs. The Challenge Problem description gives a rich description of the kinds of knowledge possessed by various players in the scenario, from which we distil the following principal objectives and assumptions:

1. Clients come to the travel agency with more-or-less specific objectives for their trip, for example a departure date and destination, a tourist attraction to visit or an academic conference to attend.
2. Clients have individual preferences about many aspects of the travel services that may be booked, including dietary choice, smoking/non-smoking, cost, comfort level, choice of provider, etc.
3. The travel agency does not posses the data to arrange trips or trip segments, but must request this information from other agents. For simplicity in building the model, we assume that the travel agency does know the identities of the supplier agents (a more realistic interpretation would be to require that the agency contacts suppliers through a brokerage or advertising service).
4. Requests from the travel agency to the suppliers may be made at varying levels of specificity (for example "a flight from Washington to London" vs. "a seat on BA1234 from Washington to London").
5. Interaction with the suppliers will produce multiple potential solutions to the client's initial request. Priority should be given to solutions that match the clients' preferences, noting that the preferences may not be unambiguously consistent.
6. Solutions may specify constraints that are not relevant to the client ("no dogs in the hotel"), or may be of unknown relevance.

The vocabularies used by different suppliers and the travel agency may vary – for example, one may use kilometres while another uses miles.

2.1 Issues from the challenge problem
From the distilled problem statement, we highlight the following challenges for agents to address in this scenario. Note: this is not intended to be an exhaustive list.

- modelling the motivations and attitudes of the actors – there are a number of actors in the scenario, and we assume each to be predominantly self-interested. The BDI model accounts for the mental attitudes of a given agent, but to correctly represent the scenario, the travel agency agent, for example, must also account for at least the goals and preferences of the user. We do not assume that user preferences can, in practice, be reduced to a utility function.
- ownership and responsibility – arguably, the motivations of the travel agency itself should be accounted for. For example, should the agent recommend suppliers that have high commission rates for agency, even if the utility to the client is neutral or reduced? Should the agent explicitly model its contract to the client and the travel agency?
- reconciling vocabularies – different agents or services will use different vocabularies, for the same or

overlapping concepts. A simple example is the use of miles and kilometres for distance, but other examples will be more subtle or complex.

- varying degrees of detail in queries – at different stages in the trip design process, queries will have differing degrees of specificity. For example, "is it possible to take a train from London to Paris", compared to "when would a train departing Waterloo at around 10:00 on the 27th arrive in Paris?"
- checking solutions for acceptability – testing for basic feasibility, including such constraints as not being in two separate vehicles at the same time
- ranking and critiquing candidate solutions – given that more than one possible solution exists, the user's preferences should be used to rank the solutions. This is likely to be needed incrementally, to control the growth of the search space.
- choosing which constraints to relax during negotiation – if a good solution is not available with the current constraints, it may be that relaxing some of them will yield an acceptable solution. For example, a three star hotel might have to be used to keep the cost within the client's budget.
- choosing when to ask the client to resolve choices or provide more preference constraints – this involves managing the dialogue with the client to neither require them to 'brain-dump' their entire preference set at the beginning, nor to be barraged with low-level questions.

Not all of these issues are addressed by the use of ontologies, though it would seem that the use of an ontology representation has some impact on the solutions to most, if not all, of them.

3. OVERVIEW OF THE NUIN PLATFORM

Nuin [9] is an agent platform we have created to assist agent designers to program deliberative agents, with a particular emphasis on BDI [16] agents. Nuin is founded on Rao's AgentSpeak(L) [15], and extended to make a practical, Java™-based programming tool. In this section, we briefly introduce some of the key features of Nuin, in order to provide some background for the solution vignettes in section 4.

A key objective in developing Nuin was to create a flexible platform for building practical agents from high-level abstractions, such as beliefs, desires, intentions and plans. The emphasis has not been on building agent infrastructure services: we assume the existence of an underlying services architecture, and Nuin provides a services abstraction layer that allows to bind to a particular service fabric, for example the Jade agent platform [5]. Nuin's architecture is influenced by the FIPA abstract architecture [12], in order to better utilise existing agent infrastructure projects. We do not, however, assume that Nuin will operate only in an FIPA environment.

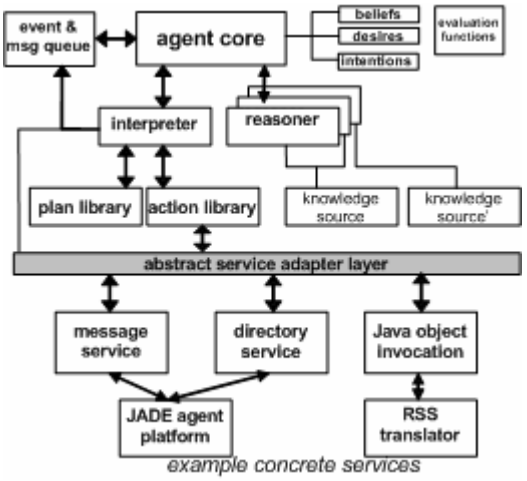

Figure 1: Outline Nuin architecture

As figure 1 shows, each Nuin agent has an interpreter, which runs one or more scripts to provide the agent behaviour. An agent also has a set of beliefs, as first-order sentences, and any number of other knowledge sources, each of which is labelled by a distinct symbol. A knowledge source can be wrapped by one or more reasoners, which provide additional services over the storage and retrieval of asserted sentences. Backwards-chaining reasoners attempt to solve queries that they are given, in essence by building a proof tree. Forwards-chaining reasoners opportunistically assert additional entailments when formulae are asserted into the knowledge store.

The key abstraction in defining an agent's behaviour is the *plan*. Following AgentSpeak(L), a plan has one or both of: a triggering event condition or a logical postcondition. Internal control flow within the agent is managed by a queue of events, which can be exogenous or endogenous, and include messages from other agents as a sub-type. A triggering condition is a Boolean expression formed from two predicates over events: `on( E )` is true when E unifies with the current event at the head of the event queue, whereas `after( E )` is true when E unifies with an event from the agent's memory of past events. The body of a plan is a set of individual actions, composed with either a sequence operator (`;`) or a non-deterministic choice (`|`). Plans may invoke other sub-plans directly or by post-condition, and may recurse.

Currently, a plan library is supplied to the agent as part of its script. However, there is no *a priori* reason why the plans could not be dynamically generated by an online planner, and this is a capability we intend to add in the future.

All of the abstractions shown in figure 1 are specified using Java interfaces, and created using the design pattern Factory Pattern. This makes it very easy for a programmer to provide a customised variant of a particular part of the system. This flexibility and extensibility was a key design goal for the platform. The configuration of the agent is specified as an RDF document, the URL of which is passed to the agent as a start-up parameter.

Given that we want to develop agents for the Semantic Web, we allow RDF stores as knowledge sources, using Jena [13]. In addition, all internal symbols are URI's. Jena's ontology reasoners are used to extend the entailments in the RDF stores, where OWL or DAML+OIL sources are available.

The next section illustrates the use of Nuin in a series of vignettes addressing some of the challenges outlined in section 2.1. Note that the script syntax illustrated is also a configurable aspect of our platform. The encoding illustrated is NuinScript,

but this is only one possible syntax that can parse into the internal abstract syntax form. An XML encoding is also planned.

4. SOLUTION EXAMPLES USING NUIN

4.1 Preamble: use of ontologies

In the challenge problem description, a sample ontology for this domain is provided by Corcho et al [7]. We decided to create our own ontology, although it shares some characteristics with that of Corcho and colleagues. Our ontology is written in DAML+OIL [2], which allows us to use richer constructs than that in the sample ontology. For example, figure 2 shows a 5-star hotel in our formulation:

```
<daml:Class rdf:ID="QualityRating">
 <daml:oneOf rdf:parseType="daml:collection">
  <travel1:QualityRating rdf:about="#OneStar"/>
  <travel1:QualityRating rdf:about="#TwoStars"/>
  <travel1:QualityRating
rdf:about="#ThreeStars"/>
  <travel1:QualityRating
rdf:about="#FourStars"/>
  <travel1:QualityRating
rdf:about="#FiveStars"/>
 </daml:oneOf>
</daml:Class>

<daml:Class rdf:ID="FiveStarHotel">
 <rdfs:subClassOf rdf:resource="#Hotel"/>
 <rdfs:subClassOf>
  <daml:Restriction>
   <daml:onProperty rdf:resource="#rating"/>
   <daml:hasValue rdf:resource="#FiveStars"/>
  </daml:Restriction>
 </rdfs:subClassOf>
</daml:Class>
```

Figure 2: DAML+OIL ontology fragment for five-star hotel

Compare this with the definition from the sample ontology (slightly abbreviated):

```
<rdfs:Class rdf:ID="hotel5star">
 <rdfs:comment>First class hotel</rdfs:comment>
 <rdfs:subClassOf rdf:ID="#hotel" />
 <NS0:numberOfStars>5</NS0:numberOfStars>
</rdfs:Class>
```

Figure 3: RDFS fragment from OAS'03 call for papers

RDFS does not have the machinery to declare that quality ratings may have exactly one of one, two, three, four or five as values. Nor is it possible to infer in RDFS that having a five-star rating and being a hotel *entails* being in the class FiveStarHotel. In the sample RDFS ontology, membership of this class must be stated explicitly. Finally, we note that the RDFS ontology requires class 'hotel5star' to be treated as an instance, since the class itself is the subject of the statement '`numberOfStars 5`'. Looking ahead, we intend to switch to using OWL [8] as our ontology language [1]. The use of classes as instances necessarily places the `hotel5star` construct in the OWL Full variant of that language, for which it is known that inferencing is expensive and incomplete. Note that we have chosen not to use cardinality restrictions to define hotel star-classes. Consider this definition

[1] The only reason we have not yet adopted OWL is that tool support is limited, partly because, at the time of writing, the OWL specification is not yet complete. This situation is improving rapidly, however, and we anticipate switching to OWL very soon.

```
<daml:Class rdf:ID="FourStarHotel">
 <rdfs:subClassOf rdf:resource="#Hotel" />
 <rdfs:subClassOf>
  <daml:Restriction>
   <daml:onProperty rdf:resource="#hasStar" />
   <daml:cardinality rdf:value="4" />
  </daml:Restriction>
 </rdfs:subClassOf>
</daml:Class>

<Hotel rdf:about="http://quite-nice.com">
 <hasStar rdf:value="*" />
 <hasStar rdf:value="*" />
 <hasStar rdf:value="*" />
 <hasStar rdf:value="*" />
</Hotel>
```

Figure 4 Using cardinality restrictions as for hotel classes

Figure 4 suggests that the Quite-Nice hotel is a four star hotel, but we cannot know this for certain. Given the open world nature of the semantic web, we cannot be sure that we have collected all of the relevant statements about URI `http://www.quite-nice.com` – we may yet discover an additional `hasStar` statement. We can only rule out the Three Star and below classes.

Space does not permit a full explanation of our sample ontology in this paper. Elements of the ontology will be introduced below as needed. The full ontology is, however, available online at: `http://jena.hpl.hp.com/ontologies/travel1`.

4.2 Initial client to agent communication

At the beginning of the process, the client's basic goal to take a trip of a certain form must be communicated to the agent. We leave aside the machinery of the human-computer interface (important though it is), to consider the process. The agent must have access to two kinds of knowledge:

- the primary goal that initiated the travel request, and
- the client's travel preferences

We assume that a message is delivered to the agent with the first of these, and that the second can be queried from a general database of known preferences. Since we are interested in Semantic Web agents, we assume that the client preference information is available in at least RDF (if not DAML+OIL or OWL).

What should the message contain? An important choice is whether the agent is seen as a collaborative partner, or a subordinate. In the second case, the message might be a FIPA `request` message, which takes an action as parameter. The action is essentially an encoding of "book a trip respecting these constraints". The agent would directly adopt an intention to carry out the action. The first case would correspond to sending a FIPA `inform` message [2] saying "the client has a goal to go on a trip, with these constraints". We would then rely on the agent being programmed with social or behavioural rules that would translate this recognition of the user's goal into an intent of its own to assist with the development of the travel plan. For the scenario of a single client walking into the travel agency's office seeking to make a booking, the difference between these two approaches is slim. Indeed, the collaborative approach adds extra complexity that the directive approach avoids. However, consider the often quoted desire for proactive behaviour in agents. The recognition of the user's goal may arise by inference, rather than by a directive from the user. If the agent is

[2] Note that the FIPA ACL specification [11] does not include a performative that directly delegates a goal to another agent.

able to infer that the user has a goal to make a trip (e.g. by having a paper accepted at a conference), it can proactively instigate the travel planning process.

Both approaches are supported by Nuin. Figure 5 shows a plan fragment[3] that reacts to an incoming message that the user has a goal to make a booking, and creates a suitable intent.

```
plan
 on message()
     {fipa:performative ~ fipa:inform,
      fipa:content ~ goal() {
          user ~ ?u,
          makeTrip ~ ?t,
          constraints ~ ?c
      }
     }
 do
   holds desire( cooperate, ?u ) ;
   intend-that
     finalised( trip( ?t, ?proposal ), ?c )
end.
```

Figure 5: plan to adopt user goal as agent intention

Thus: if a message is received informing the agent that the user has a goal to make a given trip, and the agent desires to be cooperative with that user (it may, of course, be predisposed to be generally cooperative), then adopt an intention to achieve a finalised proposal starting from the initial conditions `?t` and respecting constraints `?c`. We can make use of an ontology of different booking types to generalise this condition slightly:

```
plan
 on message()
     {fipa:performative ~ fipa:inform,
      fipa:content ~ goal() {
          user ~ ?u,
          makeBooking ~ ?b,
          constraints ~ ?c
      }
     }
 do
   holds desire( cooperate, ?u ) ;
   holds rdf:type( ?b, makeTrip ) ;
   intend-that
     finalised( trip( ?b, ?proposal ), ?c )
end.
```

Figure 6: plan to detect a trip booking and adopt an intention

Figure 6 shows a plan that reacts to any booking request, but checks that it can infer a trip booking before proceeding. The `rdf:type makeTrip` may be stated directly, or it may be an entailment from the ontology class hierarchy, or rely on other semantic entailments from the ontology language definition.

4.3 Interactions with suppliers

The travel agency's agent does not handle provisioning of the various elements of the trip itself. It will therefore need to communicate with the various suppliers in order to decide on flights, rail journeys, hotels and so on. It could be the case that the interface to each supplier is a web service, and the agent's job would then be to invoke the web service by fashioning a suitable SOAP [17] call, or whatever the appropriate mechanism is for that service. This can be accommodated in Nuin by either designing a custom web-service action that gets invoked from the script, or by registering a Java object binding that gets invoked by the built-in `invoke service` action. However, for the purpose of this exercise, we assume that the suppliers are also agent-based, and that collaboration becomes a problem of inter-agent communication.

First we note that a similar problem arises between agents as between the client and the travel agency agent. Should the agents invoke actions on the other agents, or delegate an intention or goal? One determining factor may be the need to build a coherent and optimal solution according to the client's preferences. The travel agency agent could determine which of the customer's preferences were relevant to a given subgoal, and pass these to the supplier agent. Indeed, if the client's preferences are available as a Semantic Web source, then (ignoring the important details of security and privacy) the supplier agents could access the client's preferences directly. The potential difficulty here, though, is building a globally optimal solution. Having each supplier agent construct an optimal segment of the journey does not guarantee that the overall solution is optimal. It may well be possible to use inter-agent negotiations among the whole community of stakeholder agents to build a globally optimal solution, but that is not the focus of our current research. Therefore we assume that the travel agency agent sends queries to the supplier agents, and assembles the solution pieces into an overall trip proposal. All negotiations are then pair-wise, with one of the parties always being the travel agency agent. The travel agency agent is solely responsible for optimising the solution.

The FIPA `query-ref` performative seems appropriate for the task of seeking solution elements from the suppliers. But what should the content of the message be? At the beginning of the process, we may know that John wants to travel from Madrid to Washington. We could query all known transportation services providers for routes that originate in Madrid. This, however, would generate many air-routes from Madrid, including those taking John away from the USA, plus road and train journeys to France and Portugal. We could ask for routes starting in Madrid and terminating in Washington DC, which would allow airlines to report their suggested routes (via Paris Charles de Gaulle, for example). Another tactic would be to use the geographic elements of the ontology to test whether a supplier is able to provide a single journey to a given region (e.g. Madrid to the Eastern USA) and use this to prune the search space by querying in more detail only those agents that can provide suitable routes in principle. This tactic may be invoked directly from the agent's script; it may also be invoked by the agent monitoring the responses to queries, noticing a high branching factor in the search space, and adopting an improved strategy. The current version of Nuin does not support this meta-monitoring directly. We will investigate convenient mechanisms for doing so in future versions.

We make the distinction in our ontology between journeys, routes and bookings. Initially, we query the supplier agent for information on routes. A route has a start and end location, distance and vehicle. A given instance of a route may start at Madrid airport and end in Paris Charles de Gaulle, and use an Airbus A320. We can infer that an A320 is an `AirbusPlane` which is an `Airplane`, thus this trip is also in the class `AirTravel` because `AirTravel` is defined as the class that

[3] *Syntax note*: terms with fixed arity are encoded like Prolog terms, with a functor and fixed argument list between parentheses. However, many structured terms have variable numbers of arguments (consider the FIPA message structure). Nuin supports both constructions: a Prolog-like term may be decorated with an additional list of named parameter-value pairs, of the form `functor() {key ~ value, … }`. Unification is extended to unify named arguments as well as positional arguments.

has `vehicleType` of class Airplane. Figure 7 shows a fragment of our ontology class hierarchy (using Protégé [14]):

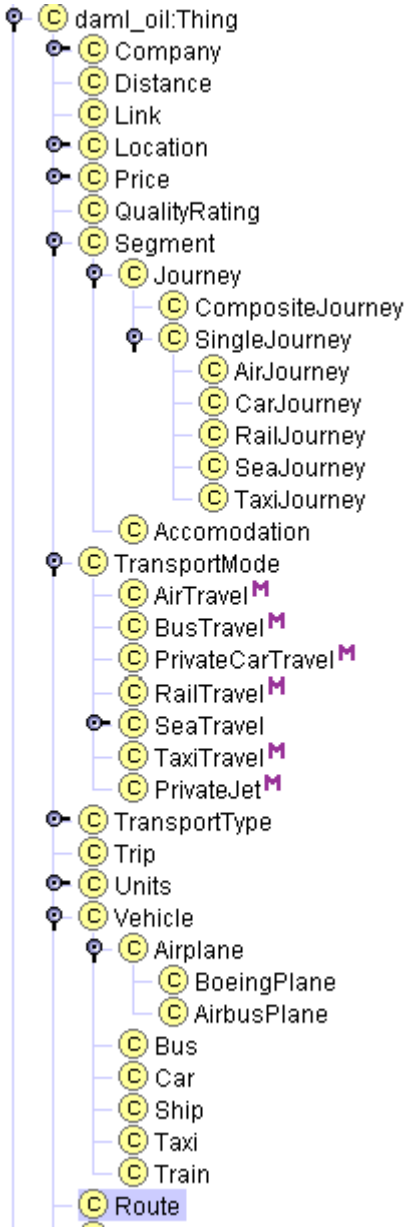

Figure 7: section of ontology class hierarchy

This approach highlights a particular difficulty with ontology development: when to uses classes vs. instances. We can define `A320` as an instance of the class `AirbusPlane`, and for many applications it is sufficient to know that a given route uses an (i.e. some unknown) A320. But for other applications, such as aircraft maintenance or scheduling, we need to know which individual aircraft, so `A320` should be a class, and instances of it would be named by the individual aircraft identifiers. But to define the route, naming the individual plane is incorrect, since different actual planes will fly the route on different days. Using DAML+OIL (or OWL), we can define an auxiliary Route subclass using a restriction:

```
<daml:Class rdf:ID="A320Route">
 <daml:subClassOf rdf:resource="#Route" />
 <daml:subClassOf>
  <daml:Restriction>
   <daml:onProperty
rdf:resource="#vehicleType"/>
   <daml:toClass rdf:resource="#AirbusA320" />
  </daml:Restriction>
 </daml:subClassOf>
</daml:Class>
```

Figure 8: class description for routes that fly A320's

For any `A320Route`, we can infer that the transporting vehicle is an Airbus A320, even if we don't know which one. It is an open question, however, whether the extra complexity introduced by this definition is worthwhile, or whether we should have multiple ontologies (e.g. one for travel and one for maintenance) and a process for translating between them when necessary.

In Nuin, we implement the process of sending the `query-ref` as a `message send` action, followed by a `suspend until` the reply is received. This works for a single communication with another agent. If, however, there are multiple agents involved, a better alternative would be to send a series of messages out, and have plans that trigger on the incoming reply messages. There are two difficulties with the second approach: firstly, enough state has to be asserted into the agent's beliefs (or other KS) to allow the agent to continue developing the plan from that point, and secondly it is harder for the agent to monitor a lack of response from the remote agents and adapt accordingly. We solve the first by assigning each partial trip its own unique identifier, and use the `reply-with` field to relate incoming answers to the results of previous planning. This then generates a set of new, extended partial plans that get new identifiers. For the second problem we do not have a convenient solution. A possible future extension to the Nuin platform will be to include first-class support for the FIPA interaction protocols [3]. Either directly as a result of supporting interaction protocols, or as a result of implementing the necessary supporting code, we hope that a clear and practical solution to the meta-monitoring problem will emerge. Note that, in our opinion, it remains an open question as to whether the ability of PRS-based agent architectures to recurse to meta-level planning is a viable solution to this problem (without creating enough complexity in the agent plan to make it difficult to perform software design and maintenance).

4.4 Reconciling vocabularies

In general, determining the correspondences between two (or more) ontologies is a very difficult task, requiring extensive human intervention [10]. Once the mapping between two ontologies is defined, it is possible that translations between a value expressed in one ontology and a value expressed in another can be automated. Some transformations are fairly straightforward, such as the units conversion (e.g. from km to miles and vice versa).

In a multi-agent system, there is a open question about whose responsibility it is to do ontology conversion. One possibility is for each agent to have a normal form that it uses for its own knowledge representation. Each received sentence would then normalised, using the information from ontology mappings where necessary. This would cope well with allowing communications from agents that used different measurement units, for example, providing that the units themselves are explicit in the ontology. An alternative is that the ontology used by the receiving agent is advertised in a public directory, and it is the originating agent's responsibility to do any necessary

translations before sending a sentence as part of a message. A further alternative is an intermediate position between these two, where the agent community includes translator agents that can handle two-way translations between agents using different ontologies. A version of the intermediary architecture may be needed when providing large semantic web or other legacy information sources into the agent community. It is often impractical to translate the entire information source to a different ontology, but it may well be possible to wrap the information source with a mediating agent that dynamically performs the necessary ontology-based transformations on queries and results. We used this strategy effectively in a project that used DMOZ [1] information in a distributed knowledge-sharing application [4]. Rather than convert the very large DMOZ data set to RDF, it was stored in a custom database layout and queries and query results were dynamically translated to RDF as needed.

Using Nuin, we can define a plan that triggers when incoming messages are received, and use this to check that the message content is in a suitable ontology. If not, it may be a simple action to do the translation locally if the agent is capable of doing so, or the agent may adopt an intention to translate the message content to a suitable ontology. This intention may then be discharged in different ways, for example by sending a request to the translator agent. Once the message is expressed in a known ontology, an event is raised to trigger further processing on the message content.

Our current experiments with the Challenge Problem make the simplifying assumption that the global ontology is shared. This assumption is only valid for such a self-contained exercise. Any realistic scale of application, especially one that uses open semantic web information sources, will be exposed to the ontology reconciliation problem.

4.5 Critiquing and ranking solutions

As the travel agency agent begins to assemble solutions to the client's requested travel goal, it will be faced with a rapidly expanding search space. In order to improve its chances of success, it should choose to pursue only those partial solutions that are promising. If the agent waits until solutions (i.e. travel plans) are complete to critique them, it is likely still to be processing long after the client's patience has run out and they have left the store. This implies that we must be able to critique partial solutions to the problem, and select which ones will be further expanded. We note that planning algorithms have been studied extensively for many years in AI, and it is not our intent in this short paper to revisit the many choices that a planning system can adopt to be able to plan effectively. Pending deeper investigation of this topic, our current design uses a simple forward-chaining means-end search algorithm. As mentioned above, we assign each partial solution a unique identifier. A solution is a series of segments, each of which is either a journey segment or an accommodation segment. The journey segment identifies the route, and may be composed of a series of individual journeys. A segment has an associated cost.

Reviewing the Challenge Problem text, we hypothesise that the following represent typical preferences a client may have over journey segments:

- type vehicle (e.g. Airbus A370)
- cost
- quality rating (first class, business class, economy, five star, etc)
- existence of facilities (TV, Internet connection, smoking rooms, pool)
- preference of mode of transport (fly vs. drive) – which may be conditional on other factors, such as accessibility of airport
- distance to local amenities (sightseeing, ski, beach, etc)

Some of these preferences will be fixed, some context dependent. On a business trip, customers might be less cost-sensitive than on a personal vacation (or vice versa!). In summer, distance from ski resorts is less important than distance from the beach.

We would like to explore making this preference information as widely available as possible, so encoding it as a semantic web resource seems plausible (we ignore for the time being important requirements to do with security and privacy).

One natural approach is to consider the various categories of alternatives that the client might prefer as ontological classes. Thus, a customer who prefers non-smoking hotel rooms has a preference for a room in the class `NonSmoking` over class `Smoking`. A simple way to encode this in the client's profile is shown in fig 9:

```
<Preference>
  <prefer rdf:resource="#NonSmoking" />
  <over rdf:resource="#Smoking" />
</Preference>
```

Figure 9: First attempt at encoding user preferences

```
<Preference>
  <prefer>
    <NonSmoking />
  </prefer>
  <over>
    <Smoking />
  </over>
</Preference>
```

Figure 10: alternative encoding for user preferences

This example uses classes as individuals, so again, exceeds the limitations of OWL DL and OWL Lite. An alternative approach would be to treat the preference arguments as expressions, using RDF blank nodes (bNodes) as existential variables (an interpretation sanctioned by RDF theory). This transforms the preference from fig 9 into fig 10:

The difference between these approaches may be subtle to readers unfamiliar with RDF. In the first encoding (fig 9), the arguments to the preference relation are the classes themselves. In the second encoding, the term `<NonSmoking />` is RDF shorthand for:

```
<rdf:Description>
  <rdf:type rdf:ID="NonSmoking" />
</rdf:Description>
```

that is, an anonymous node of type `NonSmoking`.

To use this second encoding, the agent must match the existential query implicit in the graph to the data at hand. This exploits a feature of RDF (not, it must be admitted, a universally loved feature) that meta-level information can be encoded in the same formalism as the object-level information. The preference query can be seen as expressing a predicate over the proposed solution classes, but is encoded in the same graph structure as the data itself.

By using pair-wise preferences of this kind, whichever approach is adopted, we obtain a partial ordering over sets of solutions. The reified Preference relationship is transitive, so a data source aware of this fact could pre-compute the transitive closure of preferences. Thus, if the client stated their preference was for 5-star hotels over 4-star, and 4-star over 3-star, the transitive closure would allow two proposed segments, one for a

5-star hotel and one for a 3-star hotel to be ranked correctly. Since the ordering would be partial, however, not all solutions could be ranked, so the solution evaluator would need to allow for sets of equally preferred candidates at any one time.

The client should be able to order their preferences, so that the preference for a certain cost band is allowed to dominate over the preference for smoking rooms, or vice versa. This could be achieved by adding a weight to the each `Preference` instance, or allowing preferences that ranked other preferences recursively. It is not clear which, if either, of these choices would work better in practice, and more experimentation is needed.

Again, speculating about the design (we have not yet implemented the solution ranking mechanism), we could encode context-dependent preferences by adding a condition clause to the `Preference` instance. The problem we foresee here is that there is no standard mechanism, *de facto* or otherwise, for encoding general predicates in RDF. Thus any mechanism that allowed the encoding of "if summer-time" on a preference of `NearBeach` over `NearSkiRun` would be dependent on a processor being aware of the encoding scheme used. The choices presented above, assuming that the existence of `Preference` is recognised, stay closer to standard RDF interpretations.

Given that we can achieve a satisfactory encoding of user preferences, we must then incorporate them into the strategy for prioritising the search space. We envisage a plan that is triggered by the asserting of a partial solution into the agent's beliefs KS, and which would rank the new solution against the current unexpanded partial solutions. Thus each partial solution is in one of two states: either it has been selected for expansion, or it has not been expanded yet, but is sorted according to the partial order defined by the user's preferences. It would only be necessary to find the highest ranked plan that has not yet been expanded that is preferred over the new solution, so searching from the front of the candidates list will be effective.

A more open question, and one that we have not yet addressed, is to be able to critique full and partial solutions, rather than just rank them. For example, if the agent was able to determine that a client could save a substantial amount of money by accepting a certain hotel that meets all criteria except having in-room Internet connections, it may be able to propose this to the user. Alternatively, such deductions might form the basis for negotiation strategies that suggest which factors to yield on, and which to stand firm on. This seems to be a fruitful area for future investigation.

4.6 Determining acceptable solutions

Before proposing a solution to the client, the agent must be certain that it has met the client's expressed criteria for the trip. We have not yet stated in this paper how the client's constraints are to be specified. This is in part because we run into limitations of standard ontological languages, since we will need constraints on the literal values of instance properties, and this is not an area that current ontology languages address.

Assuming that we have an appropriate canonicalization of the string form of a date, we can test for equality between two departure dates. But if the client specifies a departure date of "10-July-2003", domain knowledge is needed to recognise that "10-July-2003 10:16" is acceptable. Moreover, the client may actually want specify a departure date of "around the 10th of July" or "between the 4th and the 10th of July".

We may also want to specify that the trip includes a visit to the Statue of Liberty. While we can – just – imagine the creation of a pseudo-class `VisitToStatueOfLiberty`, and subsequently a check that some segment of the trip is subsumed by this pseudo-class, it is hard to see what the definition of the class would be in practice.

We thus currently define the constraints as a list of logical predicates that are interpreted by problem solvers other than the ontology reasoners. However, it remains an interesting area for speculation and future research whether there is a reasonably simple constraint language, that could be combined with a description-logic–like reasoner to give a richer means of checking consistency in candidate solutions.

5. Evaluation and conclusions

We have presented some vignettes of parts of the solution to the OAS'03 Challenge Problem using our BDI agent platform, Nuin. The key goal in the Challenge call for papers is to explore how agents would actually use ontological information. Much of the foregoing discussion represents our design thinking, since we have only begun to build the complete solution.

Our agents are strongly knowledge-based, and use logical sentences and mental attitudes for their internal modelling. Ontological information is clearly useful compactly describing the domain of discourse (especially if the same ontology is shared with other agents), and allows the agent to use class and property hierarchies to generalise and specialise queries and results.

Given our interest in building agents for the semantic web, we have restricted ourselves to the common semantic web ontology languages: DAML+OIL and OWL. Both of these languages' designs are based on description logic (DL) reasoning. The use of description logic reasoners in practical agent applications is not a widely explored topic, due in part to a limited availability of DL reasoners. More such reasoners are now becoming available, and we can expect more research into this area in future. A key component of the description logic approach is *class description*, and we have shown above a few instances of using class descriptions in the agent's reasoning. Using class descriptions and a meta-level `prefers` predicate to encapsulate the client's preferences appears to be a useful way to make those preferences available to a wider range of semantic web services. The limitations of description logic sentences, however, suggest that richer representations will need to be developed to encode a broadly useful sub-set of the client's general preferences.

While we have shown the use of ontology information by BDI agents, both as additional open knowledge sources for the agent to access, and as additional entailments that the agent reasoners can draw upon, we nevertheless feel that this is only a preliminary account of the integration of these two areas. Further practical experiences will help to resolve this, and we continue to develop a complete implementation of the Challenge Problem in the Nuin framework. We also look forward to the development of theoretical treatments of the interactions between the principles of deliberative agents and the principles of description logics.

6. ACKNOWLEDGEMENTS

We would like to thank the anonymous OAS'03 reviewers for their detailed comments on the original version of this paper. Due to a short deadline and a lack of space, we have not been able fully to address all of their comments, but we hope to do so in future publications. Thanks also to Dave Reynolds of HP Labs for his comments and suggestions.

7. REFERENCES

1. *ODP - The Open Directory Project.* http://www.dmoz.org
2. *The DARPA Agent Markup Language (DAML+OIL)*. 2001.
   Web site: http://www.daml.org
3. *FIPA Interaction Protocol Specifications*. 2003. http://www.fipa.org/repository/ips.php3
4. Banks, Dave, Cayzer, Steve, Dickinson, Ian, and Reynolds, Dave. *The ePerson Snippet Manager: a Semantic Web Application.* (HPL-2002-328) HP Labs Technical Report. 2002.
   Available from: http://www.hpl.hp.com/techreports/2002/HPL-2002-328.html
5. Bellifemine F., Poggi A. & Rimassa G. "Developing Multi Agent Systems With a FIPA-Compliant Agent Framework". *Software Practice and Experience.* Vol. 31:2. 2001. pp. 103–128.
6. Berners-Lee, Tim, Hendler, James, and Lassila, Ora "The Semantic Web". *Scientific American*. 2001.
7. Corcho, O., Fernandez-Lopez , M., & Gómez-Pérez , A. *An RDF Schema for the OAS Challenge Problem*. 2003. http://oas.otago.ac.nz/OAS2003/Challenge/MadridTravelOntology.rdfs
8. Dean, Mike, Schreiber, Guus, van Harmelen, Frank, Hendler, Jim, Horrocks, Ian, McGuinness, Deborah L., Patel-Schneider, Peter F., and Stein, Lynn Andrea . *OWL Web Ontology Language Reference*. 2003. http://www.w3.org/TR/owl-ref/
9. Dickinson, I. & Wooldridge, M. "Towards Practical Reasoning Agents for the Semantic Web". In: *Int. Conf. on Automomous Agents and Multi-Agent Systems (AAMAS'03)*. 2003. pp. to appear.
10. Dou, D., McDermott, D., & Qi, P. "Ontology Translation by Ontology Merging and Automated Reasoning". In: *Proc. EKAW Workshop on Ontologies for Agent Systems*. 2002. pp. 3-18. http://cs-www.cs.yale.edu/homes/dvm/papers/DouMcDermottQi02.ps
11. FIPA. *FIPA ACL Message Structure Specification.* (XC00061) 2000. http://www.fipa.org/specs/fipa00061/
12. FIPA. *Abstract Architecture Specifiation*. 2002. http://www.fipa.org/specs/fipa00001/
13. HP Labs. *The Jena Semantic Web Toolkit*. 2002. http://www.hpl.hp.com/semweb/jena-top.html
14. Noy N. F., Sintek M., Decker S., Crubezy M., Fergerson R. W. & Musen M. A. "Creating Semantic Web Contents With Protege-2000". *IEEE Intelligent Systems*. Vol. 16:2. 2001. pp. 60-71. http://www-smi.stanford.edu/pubs/SMI_Reports/SMI-2001-0872.pdf
15. Rao, A. "AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language". In: *Proc. 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW '96)*. Springer-Verlag, 1996. pp. 42–55.
16. Rao, A. & Georgeff, M. "BDI Agents: From Theory to Practice". In: *Proc. First Int. Conf on Multi-Agent Systems (ICMAS-95)*. 1995.
17. W3C. *Simple Object Access Protocol (SOAP) 1.1*. 2000. http://www.w3.org/TR/SOAP/