JASDL: A Practical Programming Approach Combining Agent and Semantic Web Technologies

Thomas Klapiscak and Rafael H. Bordini

Department of Computer Science
University of Durham, U.K.
{T.G.Klapiscak,R.Bordini}@durham.ac.uk

Abstract. Although various ideas for integrating Semantic Web and Agent Programming techniques have appeared in the literature, as yet no practical programming approach has managed to harness the full potential for declarative agent-oriented programming of currently widely used Semantic Web technologies. When agent programmers are familiar with existing ontologies for the domain of interest, they can take advantage of the knowledge already represented there to make their programs much more compact and elegant, besides the obvious interoperability and reuse advantages. This paper describes JASDL: an extension of the Jason agent platform which makes use of OWL-API to provide features such as plan trigger generalisation based on ontological knowledge and the use of such knowledge in querying the belief base. The paper also includes a running example which clearly illustrates the features and advantages of our approach.

1 Introduction

The idea of agent-oriented programming with underlying ontological reasoning was first put forward in [17]. That paper showed the changes in the operational semantics of AgentSpeak that were required for combining agent-oriented programming with ontological reasoning. The Semantic Web vision depends on the availability of ontologies [22] so that web resources are semantically annotated, but depends also on the availability of *agents* that will be able to make use of such semantically enriched web resources. For this to be possible in practice, a well devised combination of autonomous agents and semantic web technologies is essential. This paper aims to contribute towards addressing this problem.

The main advantages for agent programming that were claimed in [17] to result from the work on that variant of AgentSpeak [20] (called AgentSpeak-DL) based on a Description Logic (DL) [2] are:

(*i*) queries to the belief base are more expressive as their results do not depend only on explicit knowledge but can be inferred from the ontology;
(*ii*) the notion of belief update is refined so that a property about an individual can only be added if the resulting ontology-based belief base would preserve consistency (i.e., if the ABox assertion is consistent with the concept descriptions in the TBox);

(*iii*) retrieving a plan (from the agent's plan library) that is relevant for dealing with a particular event is more flexible as this is not based solely on unification, but also on the subsumption relation between concepts; and
(*iv*) agents may share knowledge by using web ontology languages such as OWL [15].

(NB: The four points above are a direct quotation from [17].)

However, that was a formal paper, which set the grounds for this work, but was far removed from the actual technologies — such as an AgentSpeak interpreter and ontological reasoners such as [21, 13]. As anyone with experience in applied work in multi-agent systems will agree, there are major research questions to solve and technical obstacles to overcome before a theoretical contribution becomes useful for practical work. That is probably the reason why, so far, only one attempt has been made to implement the ideas in [17], at least to our knowledge. The first initial contribution towards implementing those ideas appeared in [8]; there are, however, limitations to that approach which our approach circumvents (as discussed in Section 3).

This paper describes JASDL (***Jason*** AgentSpeak–DescriptionLogic), which uses ***Jason*** [5] customisation techniques (as well as some language constructs such as annotations) and the OWL-API [12] in order to provide all the features of agent programming combined with ontological reasoning mentioned above. To our knowledge, JASDL is the first full implementation of an agent-oriented programming language with transparent use of ontologies and underlying ontological reasoning within a declarative setting.

However, while [17] suggested changes in the operational semantics to achieve this, we did not need to change any of the core classes of ***Jason*** in order to implement AgentSpeak-DL (note that JASDL is a ***Jason***-based implementation of AgentSpeak-DL). This is due to the various customisation and extension techniques which have been built into ***Jason*** [5]. This paper shows how such mechanisms were used, the various choices that had to be addressed in order to make concrete the formal proposal for combining agent-oriented programming and ontologies, and exemplifies how each of the four features of such combination (as discussed above) can be obtained in software development using JASDL.

The remainder of this paper is organised as follows. In Section 2 we describe JASDL in detail, including a running example which helps illustrate the main features of agent programming in JASDL. In Section 3, we discuss related work and then conclude the paper and mention future work in Section 4.

2 JASDL

2.1 Some Essentials

Before we delve into the workings of JASDL, it is necessary to introduce a number of supporting concepts.

We distinguish between ontology *schema* and ontology *instance*. The former corresponds exactly to the contents of an OWL file and is read-only, while the latter is an in-memory instantiation of an OWL file that is read/write. Each agent has its own

instance of each ontology schema it is aware of, and modifications to an instance are local to the agent (unless explicitly shared using inter-agent communication).

We often refer to a "semantically-enriched" literal or *SE-literal* for short. This is a ***Jason*** literal that corresponds exactly to an axiom within an OWL ontology. A literal is marked as semantically-enriched by annotating it with the *ontology annotation* (see below), in which case this assertion belongs to the ABox of the ontology instance indicated by the label within the ontology annotation, rather than the agent's own belief base. For emphasis, a normal ***Jason*** literal that does not have an ontology annotation is referred to as "semantically-naive".

A *unary* SE-literal asserts that its term is an individual that is a member of the class given by its functor. For example, `hotel(hilton)[o(travel)]` asserts that the individual *hilton* is a member of the class *hotel* in (the local instance of) the ABox given by the label "travel".

A *binary* SE-literal asserts one of two things dependent on whether its functor refers to an *object* or a *datatype* property. It asserts that (the individual referred to by) its first term is related to (the individual/datatype literal referred to by) its second term by (the object/data property referred to by) its functor. For example, `hasRating(hilton, threeStarRating)[o(travel)]` asserts that the individual `hilton` is related to the individual `threeStarRating` by the object property `hasRating`, while `hasPricePerNight(hilton,22.0)[o(travel)]` asserts that the individual `hilton` is related to the datatype literal `22.0` by the datatype property `hasPricePerNight`.

In order for us to refer to an ontological resource in the functor of an SE-literal, its identifier must be compatible with AgentSpeak syntax. For example, it cannot be a reserved keyword (e.g., **not**) and must begin with a lowercase alphabetical character. However, there are no such restrictions placed upon the naming of objects (classes, properties, individuals, etc.) within an OWL ontology. To solve this problem, we make use of *alias to ontology object mapping*. For example, we might map the object identified by `http://www.owl.com/travel.owl\#Hotel` to the alias "hotel". For the sake of succinctness we assume this transformation is implicit from herein (as we did in the examples seen above). As an aside, in addition to allowing manual definition of mappings, JASDL provides an extensible and configurable "automapping" mechanism to perform common operations (*mapping strategies*), such as decapitalising the first letter of an object's local name, across an entire ontology. Moreover, ambiguity in this mapping is precluded in a JASDL agent by ensuring that no two identical aliases can refer to different objects in the ontology.

Each ontology instance an agent is aware of is associated with an "ontology label". This is used in the AgentSpeak **ontology annotation** (a term `o/1` used within a ***Jason*** belief annotation as in the examples above) to uniquely identify this ontology. An atomic label was chosen rather than full physical/logical namespaces purely for succinctness; JASDL code would soon become unwieldy with 50-character URIs following every SE-literal.

2.2 The General Architecture

We now explain the main components of JASDL, how they fit together, and how each corresponds to the enhancements claimed in this paper. A simplified view of the general architecture can be seen in Figure 1. Note that, throughout this paper we will be referring to points (i)–(iv) of the advantages of combining agent programming with ontologies as quoted in Section 1.

The extensibility mechanisms of ***Jason*** allow the functioning of certain steps in the agent reasoning cycle to be modified by extending core ***Jason*** classes and overriding methods as required. This technique is adopted to integrate the various mechanisms of JASDL with ***Jason***; this is to ensure that JASDL will work with future releases of ***Jason***. We now describe the three main ***Jason*** components that JASDL overrides and to what end it does so.

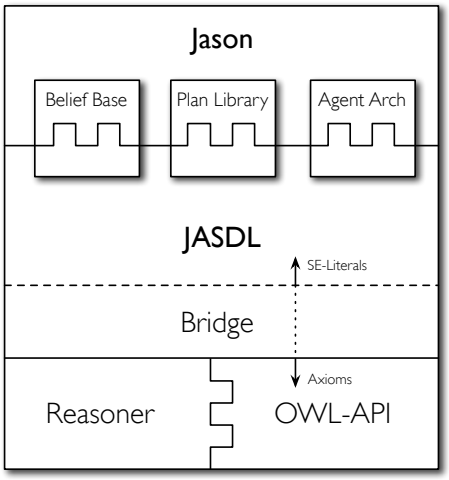

Fig. 1. The Architecture of JASDL.

Belief Base. A ***Jason*** agent stores information about the world within a data structure known as the *belief base*. Initially, the ***Jason*** belief base was simply a list of ground literals, although this has recently been extended to allow the use of Prolog-like rules [5]. JASDL extends this in such a way that the belief base now partly resides within the ABox of an ontology instance. This, in combination with a DL reasoner, facilitates the use of publicly available knowledge (in web ontologies) to increase the extent of inferences an agent can make based on its beliefs *(point i)*, and an enhanced assurance of knowledge consistency *(point ii)*.

Plan Library. The plan library of a ***Jason*** agent combines a data structure for storing AgentSpeak plans with various modification and retrieval operations. JASDL overrides these operations to facilitate enhanced plan searching *(point iii)* to include additional, more *general* (according to ontological knowledge) plans for dealing with an event.

Agent Architecture. The "overall agent architecture" provides an interface between the reasoning module of a ***Jason*** agent and the multi-agent infrastructure it uses for communication, perception, and action. ***Jason***'s extensibility mechanisms allow us to override certain aspects of this interfacing independently of implementation specifics [5]. In the case of JASDL, we augment the default architecture with message processing to facilitate semantically-enriched inter-agent communication *(point iv)*.

Our use of the OWL-API [12] provides us with a high-level means of interacting with OWL ontologies. It allows us to abstract away from the intricacies involved, for

example, in the parsing of concrete ontology syntax and the provision of reasoning services. Regarding the latter, the OWL-API provides a general reasoner interface of which there are well established implementations for the widely known DL reasoners Pellet [21] and FaCT++ [13]. The OWL-API takes an *axiomatic* approach to representing an OWL ontology, which, as the authors point out can lead to more elegant implementations than those possible using other approaches (such as the RDF-triple data model adopted by the Jena API [14]). Our experience provides further evidence to that, as a previous JASDL implementation used Jena and it seems to us that the new implementation is significantly clearer. Additionally, the OWL-API exposes various black-box debugging features (presently not supported by Jena to the best of our knowledge) which are necessary for two of JASDL's features (currently under development), which will be discussed later in this paper.

Finally, the *bridge* sub-component of JASDL encapsulates the interfacing between JASDL and the OWL-API. Its primary purpose is to provide various factory classes to conveniently allow the creation of ***Jason*** constructs (such as SE-literals) from the constructs of the OWL-API (axioms), and vice-versa.

2.3 Updating Beliefs

Point (ii) implies that a JASDL agent must be capable of adding and deleting assertions to the ABox of its ontology instance using standard AgentSpeak syntax. Furthermore, consistency of these modifications according to the constraints imposed by the associated TBox must be automatically ensured.

In JASDL, an agent belief base should be seen as being the result of the combination of two reasoning engines and knowledge representation languages. The first is identical to the default ***Jason*** belief base and is used to store semantically-naive beliefs as well as Prolog-like rules that can be used for inferring beliefs from the semantically naive ones. The second wraps around an ontology instance and is used for storing SE-literals as assertions in the ABox.

A belief addition/deletion comes in the form of a literal, l to be added to or deleted from the agent belief base. Processing of l can follow one of two flows of execution dependent on whether or not it is a SE-literal. If l is semantically-naive, it is simply passed to the default ***Jason*** belief base and dealt with in the usual way. Otherwise, l is passed to JASDL's extended belief base and dealt with as follows. Note that this process may only be applied to ground SE-literals.

1 We encode the precise meaning of l as an axiom asserting information about individuals. Table 1 gives some examples of this SE-literal to axiom translation and the intended meaning of the resulting assertion.
2 The statement is *asserted* either by adding it to or removing it from in the ABox (using the standard OWL-API mechanisms).
3 The validity of this assertion is ensured by checking the consistency of the resulting ABox. If a contradiction is inferred by the DL reasoner, the assertion is "rolled back" by using the converse of the applied operation.

Literal	Axiom	Meaning
`+hotel(hilton)[o(travel)]`	ClassAssertion Hotel hilton	`hilton` is a type of `hotel`
`+isPartOf(wembley, london)[o(travel)]`	ObjectPropertyAssertion isPartOf wembley london	`wembley` is part of `london`
`+hasPricePerNight(hilton, 22.0)[o(travel)]`	DataPropertyAssertion hasPricePerNight hilton "22"^^double	`hilton` costs 22.0 per night

Table 1. Ground SE-literal to axiom translation and associated meaning.

JASDL's notion of belief base consistency is currently based only on temporal precedence: earlier assertions precede over later ones. Future work will explore how this process can be made more elaborate. One promising approach is inspired by, and similar in operation to, the work on theoretically well motivated belief revision undertaken by Alechina et al. in [1]. Briefly, given an inconsistent ABox, we will make use of the functionality exposed by the OWL-API that allows us to generate assertion sets, each representing one possible justification for the inconsistency. We can then employ some trust rating to identify which of these explanations sets are *least preferred* (possibly derived using ***Jason***'s `source` annotation), which we then *contract*. The contraction operation can be implemented in JASDL as shown in [1]: we simply replace the mechanisms used to identify "support lists" with the functionality exposed by the OWL-API allowing us to pinpoint all possible alternative justifications for an entailment. Recursively contracting the least preferred member of each of these sets has the result of undermining our conflicting explanation set and restoring ABox consistency. Note that the property of reason-maintenance is inherent in this approach since inferences require explicit, asserted support to persist.

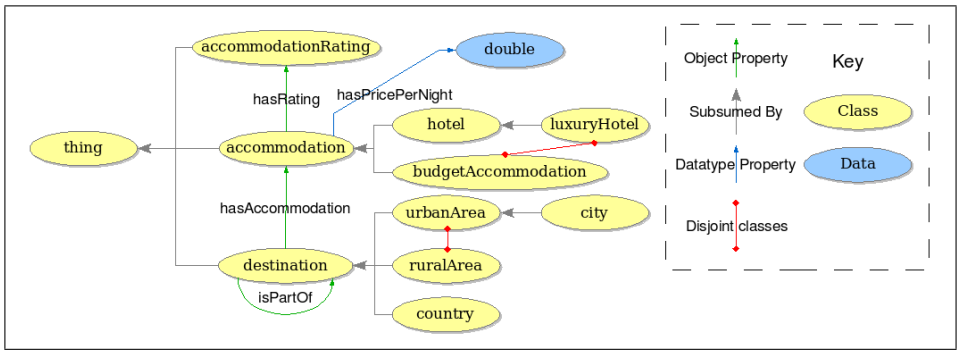

Fig. 2. Class hierarchy and property relations of travel.owl.

We will use an agent called *travel_agent* as a running example throughout the remainder this paper. *travel_agent* is a fully configured JASDL agent that is aware of a single OWL-DL ontology assigned the label "travel", the TBox state of which is given in Figure 2. The *travel* ontology is adapted from an established ontology by Holger Knublauch[1]. Our (cut-down) version describes various types of `destinations` and things that can be located at these `destinations` such as `activities` and `accommodation`. Properties are used to assign certain attributes to these things and to express relationships between them. Also asserted by the TBox, but not shown in Figure 2, is the following information:

- `hasRating` is *functional*.
- `isPartOf` is *transitive*.
- the class `luxuryHotel` is equivalent to the intersection of two classes: the (named) class `hotel` **and** the (anonymous) class formed by those individuals that are related to the individual `threeStarRating` (see below) by `hasRating`.

```
hotel(hilton)[o(travel)].                          // hilton is a hotel
hasRating(hilton, threeStarRating)[o(travel)].     // hilton has three-star rating
city(london)[o(travel)].                           // london is a city
hasAccommodation(london, hilton)[o(travel)].       // hilton is in london
country(england)[o(travel)].                       // england is a country
urbanArea(wembley)[o(travel)].                     // wembley is an urban area
isPartOf(wembley, london)[o(travel)].              // wembley is a part of london
isPartOf(london, england)[o(travel)].              // london is a part of england
hasPricePerNight(hilton, 22.0)[o(travel)].         // hilton costs £22 a night
```

Fig. 3. AgentSpeak Example: initial belief additions (comments denote meaning).

The initial ABox state of *travel_agent*'s ontology instance is given by the (initial) belief additions shown in Figure 3. Also present, but unseen here, are the individuals `oneStarRating`, `twoStarRating` and `threeStarRating` that are predefined (in the ontology *schema*) members of `accommodationRating`.

All of the belief additions seen in Figure 3 complete successfully, since no contradictory information is supplied. The plan `+!example1` seen in Figure 4 shows two belief additions which will fail, since they attempt to make assertions that are contradictory with already known information. Line 1 fails since `hasRating` is specified to be a *functional* property and therefore cannot simultaneously relate `hilton` to both `threeStarRating` and `twoStarRating`. Line 2 fails because `london` is (by inference) an `urbanArea` and the classes `urbanArea` and `ruralArea` are disjoint.

2.4 Querying the Belief Base

JASDL augments belief base querying in ***Jason*** with the combined expressive power and tractability of description logic. As mentioned earlier, the results of a query are

[1] Available at `http://protege.cim3.net/file/pub/ontologies/travel/travel.owl`.

```
+!example1
    <-
    /* 1 */ +hasRating(hilton, twoStarRating)[o(travel)];
    /* 2 */ +ruralArea(london)[o(travel)].
```

Fig. 4. AgentSpeak Example: contradictory belief additions.

no longer formed only of the knowledge available in the belief base maintained by ***Jason***, but also of that *inferred by a DL reasoner* operating over an OWL ontology external to the agent. The following section details how this is achieved and the practical implications it has for the agent designer.

Like the ABox modification operations seen in Section 2.3, processing of a literal l is passed either to JASDL's extended belief base if it is semantically-enriched, or ***Jason***'s default belief base otherwise. In the former case, however, l can be thought of as a kind of "query" indicating what is to be matched against. As before, l is translated into its OWL-API axiomatic equivalent. This process is identical to that seen in Table 1, although in this case we permit l to be either grounded or ungrounded. If l is grounded, it is simply returned as-is if its axiomatic translation can be inferred from the ontology. If, on the other hand, l is ungrounded, we return a set of grounded SE-literals corresponding to all possible groundings of l's axiomatic translation that can be inferred from the ontology.

We have not yet discussed how non-JASDL annotations (e.g., the ***Jason*** `source` annotations that denote from whom the belief originated) are dealt with. ***Jason***'s default belief base, which stores hash-table references to (semantically naive) literals themselves, handles this implicitly since annotations form part of the literal description. For JASDL it is not so straightforward since our literals are deconstructed for ontological storage and reconstructed upon retrieval. Additionally, annotations of assertions also apply to the axioms that they entail.

A possible solution to this issue, which will be considered further in future work, is to first associate SE-literals corresponding to assertions with the set of annotations applied to them (for example, in a hash-table). Then, through use of the black-box debugging features exposed by the OWL-API, we can pinpoint the set of assertions that entail an inference. The union of all annotations applied to this set then form the set of annotations that should be associated with the SE-literal representation of this inference.

By extending ***Jason***'s belief base, we ensure that ABox reasoning is applied ubiquitously across all relevant AgentSpeak syntactic operators. Hence, there are many situations under which such operations result directly from the execution of an agent's intentions. Here, we illustrate the key benefits as AgentSpeak code for our running example agent *travel_agent*. We now discuss how JASDL applies description logic inferencing in each line of the plan `+!example2` given in Figure 5.

1 This test goal will succeed since `accommodation` subsumes `hotel` and it has been asserted that `hilton` is a member of `hotel`.

```
+!example_2
    <-
    /* 1 */ ?accommodation(hilton)[o(travel)];
    /* 2 */ ?luxuryHotel(LuxuryHotel)[o(travel)];
    /* 3 */ ?~budgetAccommodation(hilton)[o(travel)];
    /* 4 */ .findall(Thing, thing(Thing)[o(Ontology)], E);
    /* 5 */ ?countryOf(wembley, Country);
    /* 6 */ ?hasPricePerNight(hilton, Price)[o(travel)];

+?countryOf(Destination, Country) :
    isPartOf(Destination, Country)[o(travel)] &
    country(Country)[o(travel)].
```

Fig. 5. AgentSpeak Example: DL belief base queries.

2 This test goal will result in the variable `LuxuryHotel` being unified with `hilton`, since a `luxuryHotel` is defined as any `hotel` that has been given a `threeStarRating`, as has `hilton`.

3 This strongly-negated test goal will succeed since `hilton` is (by inference) a `luxuryHotel` and the classes `luxuryHotel` and `budgetAccommodation` are disjoint. In other words, `hilton` *cannot possibly* be a member of `budgetAccommodation`.

4 The `.findall` internal action unifies its third argument with a list of groundings of its first argument that renders its second argument a logical consequence of the belief base [5]. Consequently, since `thing` is a super-class of all others, this line results in the unification of the variable `E` with a list of all individuals known to *travel_agent* (i.e., `[london,threeStarRating,hilton,england, ...]`). Note also the ungrounded variable argument to its ontology annotation, which results in the relevancy being checked across *all* ontology instances known to this agent.

5 Since this binary literal is not a logical consequence of the *travel_agent*'s belief base (note that it is not a SE-literal), the complex test-goal plan (`+?countryOf`) is executed. The context of this plan checks that its second argument is a `country` and its first argument is a part of this `country`. In this case, the test-goal succeeds and results in the unification of the ungrounded variable `Country` with `england`. This is because of the assertions that `wembley` is a part of `london` and `london` is a part of `england`. Consequently, by the transitivity of `isPartOf`, we can infer that `wembley` is a part of `england`.

6 This test goal will result in the variable `Price` being unified with the datatype literal 22.0 (of type double), since `hilton` is related to this value by this datatype property.

2.5 Retrieving Relevant Plans

The ***Jason*** reasoning cycle involves, in part, summating an agent's perceived environmental or circumstantial change as a set of *events*. In a given reasoning cycle, a single

event must be selected from this set. Subsequently, the agent must perform a plan search to (possibly) identify a set of plans that are *relevant* for this event. Both events and plans are associated with *triggers*, which are themselves in part composed of a literal. A plan is considered relevant to an event if these triggers match, which involves first-order unification of the associated literals [5]. JASDL extends ***Jason***'s definition of plan relevancy by, in effect, allowing this unification to be achieved using *description logic* inferencing. Finally, before a plan can be used, it must also be deemed *applicable* by ***Jason***, which is dependant upon the additional checking of the plan's context.

Consider that SE-literals now correspond to classes or properties from a TBox. These themselves form part of a taxonomy constructed from the *subsumption* relationship. An implication of this is that plans whose triggers contain SE-literals also belong to a hierarchy of identical structure. Accordingly, for an event associated with an SE-literal, we can extend ***Jason***'s definition of plan relevancy. In particular, for such an event, it may now include not only specifically designated plans, but also those that can deal with it in a more general sense as indicated by this hierarchy. This grants us a much more elaborate notion of plan generality than the use of higher-order variables in plan triggers allows (the only currently available means for this in ***Jason*** [17]). The resulting process and mechanism, corresponding to *point (iii)*, is referred to as *trigger generalisation*.

In order to override ***Jason***'s default definition of plan relevancy, first we must override the unification algorithm used to unify the trigger of a plan with the trigger of an incoming event. This is achieved by intercepting plans as they are added to the plan library and, if their triggers are associated with a SE-literal, substituting them with modified versions of their representation as a Java object. We then use Algorithm 1 to allow the matching of a trigger with all relevant plans, including those that are relevant because their triggers generalise (according to the ontology) the event's trigger.

We define the following functions and example executions within the context of our running example:

- $unify(t_1, t_2)$ performs standard ***Jason*** unification on the triggers t_1 and t_2 (and their associated literals). If they are unifiable the resultant unifier is returned, otherwise it returns $failure$.
  - $unify(+hotel(hilton)[o(travel)], +hotel(X)[o(travel)]) = [hilton/X]$
  - $unify(+thing(wembley)[o(travel)], +thing(england)[o(travel)]) = failure$
  - $unify(+!hotel(hilton)[o(travel)], +!thing(hilton)[o(travel)]) = failure.$
- $generalise(t)$ returns a set of triggers containing all triggers that are *more general* than t. A more general trigger is identical to t except the functor of the associated SE-literal is replaced with (the alias of) a subsuming ontological term.
  - $generalise(+!hotel(hilton)[o(travel)]) =$
    $\{+!accommodation(hilton)[o(travel)], +!thing(hilton)[o(travel)]\}$

Since this mechanism for DL-based generalisation of plan triggers is in place, one might expect implementation to be complete. In practice, however, this is not the case since to improve plan search efficiency, ***Jason*** does not perform relevancy checks across the entire plan library, but only a "candidate" subset obtained through an (efficient) hash-table look up. To work around this issue we apply an extension, shown by Algorithm 2, to the code responsible for generating these candidates within ***Jason***'s plan

Algorithm 1 Extended Plan-Trigger Unification

1: $t_{plan} \Leftarrow$ trigger of the plan we are testing
2: $t_{inc} \Leftarrow$ the incoming trigger
3: $\phi \Leftarrow unify(t_{plan}, t_{inc})$
4: **if** $\phi \neq failure$ **then**
5: **return** ϕ
6: **end if**
7: $T_{general} \Leftarrow generalise(t_{inc})$
8: **for all** $t_{general} \in T_{general}$ **do**
9: $\phi \Leftarrow unify(t_{plan}, t_{general})$
10: **if** $\phi \neq failure$ **then**
11: **return** ϕ
12: **end if**
13: **end for**
14: **return** $failure$

Algorithm 2 Extended Plan Candidate Generation

1: $t_{inc} \Leftarrow$ the incoming trigger
2: $r \Leftarrow relevant(t_{inc})$
3: $T_{general} \Leftarrow generalise(t_{inc})$
4: **for all** $t_{general} \in T_{general}$ **do**
5: $r \Leftarrow r \cup relevant(t_{general})$
6: **end for**
7: **return** r

library. We define the following function and example executions given a plan library containing the plans: $\{+a(X) \leftarrow \ldots, +!b(X,j) \leftarrow \ldots\}$

– $relevant(t)$ returns a set of candidate relevant plans according to a hash-table lookup by literal functor and arity only. The condition that the two triggers unify is not enforced and no unification is performed at this stage.
  - $relevant(+a(i)) = \{+a(X) \leftarrow \ldots\}$
  - $relevant(+!b(i)) = \{\}$
  - $relevant(+!b(i,k)) = \{+!b(X,j) \leftarrow \ldots\}$

The only mandatory condition for the invocation of the trigger generalisation mechanism on an incoming event is that this event be associated with a SE-literal, since otherwise we would have no taxonomic information to use for generalisation[2]. This may present a problem in larger applications using JASDL since we must incur a computational cost on all incoming events, many of which (particularly those from belief update) may be of no interest to the agent. Future work will explore possible *optional* conditions we can impose to restrict trigger generalisation invocation. For example, we may prevent it when more specific options are available (perhaps discovered using ***Jason***'s standard mechanisms alone). Additionally, we may introduce some means for the

[2] Note that for test goals an event to be generalised will only be generated if a simple unification against the belief base fails (i.e., only for *complex* test goals) [5].

agent designer to preclude certain events or *kinds* of event from invoking trigger generalisation. Additionally, because of complications due to ***Jason***'s plan failure handling mechanism, how trigger generalisation fits in with *goal-deletion* events is left to further research.

```
!luxuryHotel(fourSeasons)[o(travel)].
/* 1 */ +!hotel(hilton)[o(travel)]        <- +hotel(hilton)[o(travel)].
/* 2 */ +!hotel(H)[o(travel)] : false     <- +hotel(H)[o(travel)].
/* 3 */ +!accommodation(A)[o(travel)]     <- +accommodation(A)[o(travel)].
```

Fig. 6. AgentSpeak Example: trigger generalisation.

We now illustrate the operation of the trigger generalisation mechanism by way of a simple example appropriate to our agent *travel_agent*. Figure 6 shows the instantiation of the declarative achievement goal whose effect should be to achieve the state of affairs such that *travel_agent* is aware of a `luxuryHotel` called `fourSeasons`. In this case, we have no plan specifically designed to accomplish this. We do however have some plans that are relevant and (potentially) applicable in a more general sense. The first to be considered is that in line 1. JASDL may consider this relevant (since `hotel` subsumes `luxuryHotel`). However, the plan trigger literal is grounded (with term `hilton`) and so cannot be unified with `fourSeasons`; this plan is only relevant for the `hilton` hotel specifically. The plan in line 2 is considered relevant, however it is not *applicable* (which we forced to be the case with the context *false*). Consequently, we attempt to generalise to the plan found in line 3. This is both relevant (since `accommodation` subsumes `hotel` and thus `luxuryHotel`) and applicable (since an empty context is always true) and is therefore selected for execution by JASDL. The overall effect of this is that although we were unable to achieve the original goal in its most specific sense, we have at least achieved a generalisation of it (or possibly achieved it because a more general plan was meant to be applicable in this case).

2.6 Knowledge Sharing Among Agents

As indicated by point (`iv`), the use of ontological reasoning facilitates the sharing of knowledge among agents through the use of standard ontologies for many domains that are increasingly available on the Web. An ontology provides a shared vocabulary between two communicating agents. Moreover, a recipient agent can easily be introduced to novel ontologies, thus extending its knowledge of the world as required *at run-time*. The implications of this for multi-agent systems are huge, since the meaning of inter-agent communication can potentially be understood even with no prior agreement on terminology. As contributions from the community doing research on ontological alignment (such as [6]) fully mature, the benefits will be further increased, since two agents will be able to communicate even when using disparate ontologies.

In JASDL, the propositional content of incoming and outgoing messages can be composed of SE-literals. If this is the case, they must be intercepted (in JASDL's

customised agent architecture, see Section 2.2) and processed. In brief, for outgoing messages this is composed of two steps. First, the atomic ontology annotation label is replaced by a full physical namespace URI, thus allowing an ontology to be unambiguously referred to and possibly instantiated if novel to the recipient. Notice that simply being informed of the (accessible) physical namespace URI of an ontology immediately extends the vocabulary an agent can use to reason and communicate. Second, the `expr` (read "expression") annotation is added. This unambiguously describes the resource referred to by the SE-literal (recall that SE-literal functors may be aliased arbitrarily). If this class has been defined locally at run time, the defining class expression is placed here (see Section 2.7 for more details on the `jasdl.ia.define_class` internal action). For incoming messages, this process is reversed: uninstantiated ontologies are instantiated and run-time defined class expressions are compiled.

```
!example3.
+!example3
    <-
    .send(customer, tell, luxuryHotel(hilton)[o(travel)]).
```

Fig. 7. AgentSpeak Example: knowledge sharing among agents (*travel_agent*).

In order to exemplify the semantic inter-operability of JASDL agents we must introduce a new agent for *travel_agent* to communicate with, namely *customer*. *customer* is also aware of the "travel" ontology, but it has assigned to it the label "holiday". This discrepancy does not pose a problem given the message processing described previously. The execution of the plan `!example3` seen in Figure 7 results in a SE-literal being sent to *customer* and added to the ABox of its belief base; it also results in the instantiation of the `+luxuryHotel` plan as seen in Figure 8. We now describe, line by line, how this plan is processed.

```
+luxuryHotel(L)[o(holidays), source(Source)]
    <-
    /* 1 */ .print("The ", L, " luxury hotel is available");
    /* 2 */ jasdl.ia.define_class(query, "city and hasAccommodation value ", L, holidays);
    /* 3 */ .send(Source, askOne, query(City)[o(holidays)], query(City)[o(holidays)]);
    /* 4 */ .print(L, " is located in the city ", City).
```

Fig. 8. AgentSpeak Example: knowledge sharing among agents (*customer*).

1 The message "The hilton luxury hotel is available" is displayed to the customer on-screen. This is because the variable `L` has been unified with `hilton`.
2 The class described by the concatenated string "city and hasAccommodation value hilton" is added to the TBox of *customer*'s "holidays" ontology instance and assigned the local alias `query`.

3 The use of the `askOne` KQML performative results in the generation of a complex test-goal event by the sender of the original message (*travel_agent* in this case). This event is associated with a SE-literal corresponding to *customer*'s definition of `query`. Although this definition is local to *customer*, its meaning is shared with *travel_agent* through JASDL's use of the (predefined) `expr` annotation as discussed previously. Consequently, the variable `City` is unified with `london`. As an aside, our query is subtly different to simply asking `hasAccommodation(City, hilton)`; we are requesting something more specific since we are imposing the additional restriction that not only must the result have this object property relationship, but it must also belong to the `city` class.
4 The message "hilton is located in the city london" is displayed to the customer on-screen.

2.7 Some Final Remarks on JASDL

No feature of JASDL seen thus far allows modification of the TBox component of an ontology instance. Hence, our ABox individual retrieval capabilities are restricted to querying the ABox in terms of existing classes and properties. The only exception to this is when using strongly-negated unary SE-literals, which essentially provide a shortcut to defining the complement of a class at run-time (if not already explicitly defined in the ontology schema). Beyond this, however, it would be useful if an agent could, at run-time, arbitrarily combine classes and properties using logical operators to produce new classes for interaction with the ABox.

We have implemented an internal action `jasdl.ia.define_class` which allows an agent to do just that (for classes only). It is supplied n parameters $p_1 \ldots p_n$ such that $n \geq 3$ where each represents the following: p_1 is an atomic term that gives the alias to be used for referring to the defined class; $p_2 \ldots p_{n-1}$ are strings that are concatenated to form a class-expression defined in terms of predefined classes and properties, other run-time defined classes and the DL operators, such as *and*, *or*, *some*, *value*, etc.; p_n is an ontology label used to identify the TBox instance within which the new class definition will reside. At the heart of this internal action is the *Clexer* (or *Class Expression Parser*) library. Put simply, this library can be used to construct an OWL-API axiomatic representation of a class expression string. Clexer forms a general, re-usable contribution of this work and will be released open-source in the near future.

During the implementation of JASDL, an issue became apparent with the extensibility mechanisms provided by ***Jason*** that allow an agent designer to override the methods that implement the various selection functions[3] of an agent [5]. Conceivably, selection function implementations may be provided as libraries for general use. However, it becomes difficult for an agent designer who wishes to simultaneously make use of multiple such libraries whose implementations overlap, since they are forced to combine code in an ad-hoc fashion (presuming the library source code is available in the first place).

[3] Due to shortness of space, we cannot introduce the AgentSpeak language in this paper. Readers unfamiliar with the language are referred to the cited AgentSpeak literature.

As a solution to this problem, a separate extension to ***Jason*** was devised, namely JMCA[4] (***Jason*** *Module Composition Architecture*). JMCA permits multiple selection function implementations to interact in a well-defined manner. This is achieved by providing a framework under which an agent designer can encapsulate the implementation of a selection function within a *selection strategy*. Subsequently, an agent making use of JMCA is capable of composing a chain of such selection strategies and specifying a *mediation strategy* to mediate between the choices they make, thus defining the overall behaviour produced by the composition chain and ultimately the choices made by the agent.

By default, OWL does not make the unique naming assumption [16]. This implies that, unless explicitly stated otherwise, individuals with different names can be treated as identical by a reasoner. It is clear that, when developing an agent using JASDL, it will often be the case that we wish to explicitly state at run-time that a set of individuals are mutually distinct. To this end, we have implemented the `jasdl.ia.all_different` internal action. This accepts an ontology label and a list of atoms representing individuals. It has the effect of declaring, in the referenced ontology, each listed individual as distinct from all others in this list.

3 Related Work

Theoretical foundations for the use of ontologies in agent-oriented programming languages first appeared in [17]. The paper used the formal semantics of AgentSpeak to show a number of features that would result from the use of ontological reasoning, and forms the basis of our work. Similar ideas, restricted to an agent's belief base, and using the Go! language, appeared independently in [7]. However, differently from the main idea in JASDL which is to refer to existing ontologies on the Web at run-time (in particular, using run time instances of such ontologies as extra information available to the agent on top of its current beliefs), and making use of existing (efficient) ontological reasoners, the work in [7] concentrates in showing how to translate an OWL-Lite ontology into Go! so as to use that knowledge as part of an agent's belief. Note, however, that without an explicit link to the ontology as a web resource, and agents being able to change their beliefs, the interoperability aspect that is an important advantage of standard Semantic Web technologies is (possibly) lost.

There are a variety of agent-oriented programming languages [4], some of which have working interpreters with a growing base of users in the multi-agent systems community, such as 3APL [9] (and its recent 2APL variant), Jadex [19], SPARK [18], and JIAC [11]; there are also commercial products such as JACK [23]. Some of these platforms allow a reference to an ontology used for agent communication to be given (such as JIAC and Jade [3], and the platforms that use Jade as middleware, such as Jadex, 2APL, and also ***Jason*** itself). However, this is a long way from the functionalities that a full integration (at the declarative level) of agent-oriented programming and ontological reasoning, as first suggested in [17], would provide. For example, relevant plans to handle an event can be inferred using ontological relations when an agent is not aware of a specific plan for an event.

[4] An initial release of JMCA is available at http://jason.sf.net.

An approach combining BDI-like agents and Semantic Web technologies was introduced in [10]. The paper presented an agent architecture called *Nuin* and a scripting language, whose interpreter is based on AgentSpeak, where XML namespace prefixes can be used as part of identifiers, thus allowing particular names to be linked to given ontologies. One disadvantage of the approach is that the BDI agent programming notions are only informally described; by using ***Jason***, in contrast, we are able to take advantage of the significant body of work formalising the core AgentSpeak constructs used as part of the language interpreted by ***Jason***. Besides, to our knowledge, no implementation of *Nuin* is publicly available.

As this paper is concerned with a practical implementation of a combination of an agent programming language and ontology techniques, the closest work to ours is that on Argonaut [8]. Argonaut demonstrates how ontological reasoning can be practically integrated with ***Jason*** to grant some degree of contextual awareness to an agent providing location-based services. The approach taken by Argonaut isolates all ontological interaction within highly specialised, predefined internal actions. JASDL, on the other hand, integrates ontological reasoning more tightly and transparently with the ***Jason*** AgentSpeak interpreter. This has several advantages, as discussed below.

To implement new kinds of ontological interaction, Argonaut requires new *Java* code to be written. JASDL however allows the implementation of new ontological processes at the *AgentSpeak* level.

Since JASDL augments the ***Jason*** belief base with the capacity for ontological reasoning, additional inferencing immediately becomes ubiquitous amongst the syntactic operators of AgentSpeak. For example, test goals, plan contexts, predefined internal actions that query the belief base (such as `.findall`), belief additions and deletions, etc., all leverage the combined expressive power and tractability of description logic. Contrast this to the Argonaut approach, under which queries can be made only by means of internal actions made available as part of Argonaut. Plan contexts, for example, cannot be so naturally expressed since they must contain such internal actions.

JASDL's description logic extension to ***Jason***'s plan retrieval mechanism overloads ***Jason***'s definition of plan relevancy. This means that more general plans can be found automatically by the agent, with no additional effort required on the part of the agent designer. Although not explored by Argonaut, it might be possible to implement an analog of the trigger generalisation mechanism using internal actions alone. However, this would likely prove cumbersome and inefficient when compared to JASDL's more elegant mechanism.

Furthermore, JASDL automatically checks ABox consistency after any modification and rolls back any contradictory changes. It is difficult to see how this could be elegantly implemented and enforced using internal actions alone.

4 Conclusion and Future Work

In this paper, we have described the implementation of an existing theoretical proposal for combining agent-oriented programming and ontological reasoning. The paper highlights various issues that need to be addressed in practice which are not apparent from the abstract point of view of formal work. The running example used in the paper made

reference to an existing ontology on the Web; an important aspect of our approach is precisely to allow programmers to make use of existing knowledge representation to facilitate programming as well as allowing agents to share ontological information.

An aside benefit of the use of our approach is that it allows us to, in some cases explicitly, distinguish between declarative and procedural achievement goals (AgentSpeak does not have different syntactic constructs for this). Any achievement goal whose trigger corresponds to a semantically enriched literal — hence a belief — is inherently a declarative goal. Note that this technique cannot explicitly identify procedural achievement goals since there can still exist declarative goals corresponding to semantically-naive beliefs. This presents an interesting opportunity, which remains future work, for checking the current set of intentions for consistency (using description-logic inferences) and perhaps even automatic conflict resolution.

We plan a number of other directions for future work, including incorporating means for ontological alignment when an agent detects any ontological mismatch, and facilitating the use of Web services in the context of JASDL by means of OWL-S [5]. Future work will also include mechanisms that permit persistency of the (at present volatile) ontology instances of JASDL agents.

Although everything described in this paper has been implemented and the examples have been executed, it is still very early days for the development of JASDL. Whether its engineering principles will prove natural and useful in programming real-world applications remains to be seen, as does the analysis of performance issues. In the near future, JASDL will be made available *open source*, which should help in our aim to assess its usability and efficiency in practical applications. Nevertheless, our initial experiments show that there is great potential benefit to be obtained from the use of JASDL and performance does not seem to be an issue.

References

1. N. Alechina, R. H. Bordini, J. F. Hübner, M. Jago, and B. Logan. Automating belief revision for agentspeak. In M. Baldoni and U. Endriss, editors, *Proceedings of the Fourth International Workshop on Declarative Agent Languages and Technologies (DALT 2006), held with AAMAS 2006, 8th May, Hakodate, Japan – Selected, Revised and Invited Papers*, volume 4327 of *Lecture Notes in Computer Science*, pages 61–77. Springer, 2007.
2. F. Baader, D. Calvanese, D. N. D. McGuinness, and P. Patel-Schneider, editors. *Handbook of Description Logics*. Cambridge University Press, Cambridge, 2003.
3. F. Bellifemine, F. Bergenti, G. Caire, and A. Poggi. JADE — a java agent development framework. In Bordini et al. [4], chapter 5, pages 125–147.
4. R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors. *Multi-Agent Programming: Languages, Platforms and Applications*. Number 15 in Multiagent Systems, Artificial Societies, and Simulated Organizations. Springer-Verlag, 2005.
5. R. H. Bordini, J. F. Hübner, and M. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using **Jason***. Wiley Series in Agent Technology. John Wiley & Sons, 2007.
6. S. Castano, A. Ferrara, and G. Messa. Islab hmatch results for oaei 2006. In *Proc. of International Workshop on Ontology Matching, collocated with the 5th International Semantic Web Conference ISWC-2006*, Athens, Georgia, USA, November 2006.

[5] Refer to `http://www.w3.org/Submission/OWL-S/`.

7. K. L. Clark and F. G. McCabe. Ontology schema for an agent belief store. Unpublished manuscript available at `http://www.doc.ic.ac.uk/ klc/`, 2005.
8. D. M. da Silva and R. Vieira. Argonaut: Integrating jason and jena for context aware computing based on owl ontologies (short paper). In M. Baldoni, C. Baroglio, and V. Mascardi, editors, *Proceedings of the Workshop on Agents, Web Services, and Ontologies – Integrated Methodologies (AWESOME'007) held as part of MALLOW'007, Durham 3–7 September*, 2007.
9. M. Dastani, M. B. van Riemsdijk, and J.-J. C. Meyer. Programming multi-agent systems in 3APL. In Bordini et al. [4], chapter 2, pages 39–67.
10. I. Dickinson and M. Wooldridge. Towards practical reasoning agents for the semantic web. In *The Second International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2003, July 14-18, 2003, Melbourne, Victoria, Australia, Proceedings*, pages 827–834. ACM, 2003.
11. A. Heler, B. Hirsch, and J. Keiser. Collecting gold. jiac iv agents in multi-agent programming contest. In M. Dastani, A. El Fallah Seghrouchni, A. Ricci, and M. Winikoff, editors, *Fifth International Workshop on Programming Multi-Agent Systems (ProMAS'07) – Agent Contest paper. Held with AAMAS07, Honolulu*, 2007.
12. M. Horridge, S. Bechhofer, and O. Noppens. Igniting the owl 1.1 touch paper: The owl api. In C. Golbreich, A. Kalyanpur, and B. Parsia, editors, *OWLED 2007, 3rd OWL Experienced and Directions Workshop, Innsbruck, Austria, Proceedings*. CEUR-WS, 2007.
13. I. Horrocks. FaCT and iFaCT. In P. Lambrix, A. Borgida, M. Lenzerini, R. Möller, and P. Patel-Schneider, editors, *Proceedings of the International Workshop on Description Logics (DL'99)*, pages 133–135, 1999.
14. B. McBride. Jena: a semantic web toolkit. *IEEE Internet Computing*, 6(6):55– 59, 2002.
15. D. L. McGuinness and F. van Harmelen, editors. *OWL Web Ontology Language overview. W3C Recommendation*. Avalilable at http://www.w3.org/TR/owl-features/, February 2004.
16. D. L. McGuinness and F. van Harmelen, editors. *OWL Web Ontology Language Reference. W3C Recommendation*. Avalilable at http://www.w3.org/TR/owl-ref/, February 2004.
17. Á. F. Moreira, R. Vieira, R. H. Bordini, and J. F. Hübner. Agent-oriented programming with underlying ontological reasoning. In M. Baldoni, U. Endriss, A. Omicini, and P. Torroni, editors, *Proceedings of the Third International Workshop on Declarative Agent Languages and Technologies (DALT-05), held with AAMAS-05, 25th of July, Utrecht, Netherlands*, number 3904 in Lecture Notes in Computer Science, pages 155–170. Springer, 2006.
18. D. Morley and K. L. Myers. The spark agent framework. In *3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004), 19-23 August 2004, New York, NY, USA*, pages 714–721. IEEE Computer Society, 2004.
19. A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: A BDI reasoning engine. In Bordini et al. [4], chapter 6, pages 149–174.
20. A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. Van de Velde and J. Perram, editors, *Proceedings of the Seventh Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96), 22–25 January, Eindhoven, The Netherlands*, number 1038 in LNAI, pages 42–55, London, 1996. Springer-Verlag.
21. E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical owl-dl reasoner. *Web Semant.*, 5(2):51–53, 2007.
22. S. Staab and R. Studer, editors. *Handbook on Ontologies*. International Handbooks on Information Systems. Springer, 2004.
23. M. Winikoff. JACK$^{\mathrm{TM}}$ intelligent agents: An industrial strength platform. In Bordini et al. [4], chapter 7, pages 175–193.