Greeks and Trojans Together

Luís Botelho Nelson Antunes Mohmed Ebrahim Pedro Ramos
Luís.Botelho@iscte.pt Nelson.Antunes@iscte.pt Mohmed.Ebrahim@iscte.pt Pedro.Ramos@iscte.pt
Communicating Intelligent Systems Group of ADETTI
Av. das Forças Armadas, Edifício ISCTE, 1600 Lisboa, Portugal

ABSTRACT

This paper describes a comprehensive solution for the integration of object oriented ontology representation frameworks with logic-based agent communication frameworks. The proposed solution addresses the problem at both the agent communication level and the agent implementation level. At the agent communication level, we propose to extend logic content languages with some domain independent operators that allow building logic constructs as propositions from domain dependent entities defined in an object oriented ontology. At the implementation level, we propose to use object-oriented databases as the support for the agent information. Finally, we propose an automatic mechanism for translating agent messages using the extended content language into ODMG OQL commands, which are then used to interact with the object-oriented database. This binding mechanism relies on a special purpose data dictionary representing the mapping between the domain ontology and the agent internal database.

Keywords

Ontology, Agent Communication, Content Language

1. INTRODUCTION

In agent communication, a message can be understood only if the vocabulary used in the message content belongs to the ontology used by the agent. If the vocabulary contained in the ontology is described in terms of classes, objects, attributes and methods, but the message content uses predicates, functions, constants, and actions, the agent becomes helpless unless it can use some way of integrating the two different representation frameworks.

The paper presents an approach to integrate object-oriented ontologies with logic-based communication. In the scope of this paper, the expression "object-oriented ontology" refers to ontologies in which the domain is represented in terms of classes, objects, attributes, and methods. The expression "logic-based communication" refers to communication frameworks in which the contents of the messages are built form predicates, functions, constants, variables and actions.

In the Agentcities project [1], FIPA ACL [5] is used as the agent communication language; FIPA SL [6] and KIF [8] may be used as message content languages; and DAML+OIL [11] is used to represent ontologies. These choices were driven by a set of well founded reasons including project management reasons, current industrial and standardisation trends, and existing technological support. In spite of being well justified, these choices are not free of problems. Namely, they imply the harmonisation of the logic-based agent communication framework and the object oriented ontology representation framework. Whereas FIPA ACL, FIPA SL and KIF fit into a logic-based framework, DAML+OIL fits into an object-oriented framework.

The choice of FIPA ACL (and, in general, any other agent communication language based on speech acts [10]) entails the use of a logic-based message content language because the contents of ACL messages must be combinations of propositions and terms. Information and closed query messages take propositions as content; open query messages take referential expressions as content; and request, negotiation and error messages take combinations of actions and propositions as content. FIPA SL and KIF are logic-based languages capable of representing propositions and terms, as implied by the choice of FIPA ACL language.

The paper presents an approach for the integration of object-oriented and logic-based frameworks both at the communication level and at the implementation level. At the communication level (section 2), the paper proposes four new relational, functional and action operators that may be used to build logic-based communication constructs such as terms and propositions from classes, objects, attributes and methods. At the implementation level (section 3), information agents are implemented on top of object databases, which are well suited to directly maintain the information of the domain, which is formed by instances of the object-oriented domain ontology. Received messages are translated to commands of the object-oriented database query language.

Section 4 presents three approaches that can be considered alternatives to the current proposal. In one, [2] proposes the extension of FIPA ACL with new performatives taking objects as contents. In another alternative [4], we discuss the possibility of using classes of the domain ontology to represent propositions and other entities used in communication. Finally [9], we analyse the use of transposition rules to convert the object-oriented ontology into a relational model.

2. AGENT-COMMUNICATION LEVEL

This section describes four general-purpose operators that can be used in the communication for creating propositions and terms (including action propositions and action terms) from domain classes, objects, properties and methods.

The explanation considers two possible modelling scenarios. In one scenario, domain predicates are represented by methods; domain actions are represented by methods; and domain functions are represented by methods. In the other scenario, domain predicates are represented by classes; domain actions are represented by classes; and domain functions are represented by methods. The system designer is free to choose his/her preferred modelling approach.

2.1 Extending Logic-Based Content Languages

Some of the proposed new operators are relational operators (i.e., predicates), others are functional operators, and others are action operators.

The proposed extension assumes objects may be represented in a logic-based content language as proposed in [2]. That is, an object is a functional expression in which the functor is the name of the object class, playing the role of a class constructor. The functional expression arguments are the attributes of the object to be created. Using this convention, the following functional expression represents a restaurant named "Encher a Mula" with several other attributes including *restaurantAddress* whose value is an object of class *Address* having attributes *publicPlace*, *number*, *city*, and *zone*.

```
(Restaurant
   :name "Encher a Mula"
   :phone 219999999
   :restaurantAddress (Address
      :publicPlace "Rua Associado Dias"
      :number 1
      :city Lisboa
      :zone downtown)
   :type traditional)
```

Figure 1. Object instance representation

The new added operators enable the manipulation of classes, objects, attributes and methods.

New relational operator

instance/2 is a new relational operator used to access instances of specified classes. (`instance` *`Object ClassName`*) means that *Object* is an instance of the class named *ClassName*. Operationally, *instance/2* can be used to check whether an object is an instance of a class and also to access the different instances of a class. *instance/2* was originally proposed in [2].

New functional operators

value/2 is a new functional operator used to access the value of an attribute of an object. (`value` *`Object AttributeName`*) is the value of the attribute named *AttributeName* of *Object*.

value/3 is used to apply a certain method with the specified set of parameters to an object. (`value` *`Object MethodName ArgumentSequence`*) is the value returned by the application of method named *MethodName* with the arguments specified by *ArgumentSequence* to *Object*. *value/3* can be used only with methods that do not return void.

If the attribute specified in *value/2* or the method specified in *value/3* has multiple values, then value-expressions will represent sets.

Value expressions may also be used with class methods and class attributes instead of object methods and attributes. In those cases, the first argument of the operator must be a class name.

value/2 was originally proposed in [2] as a relational operator.

New action operators

Action operators represent the execution of the specified action. Our proposal has two action operators: *apply/3* and *execute/1*.

apply/3 is an action operator used to represent the application of a certain method with the specified set of parameters to an object. (`apply` *`Object MethodName Arguments`*) represents the application of method named *MethodName* with the arguments specified by *Arguments* to *Object*. Arguments may be either the sequence containing the values of the method arguments or a set of named arguments. *apply/3* should be used with methods that change the sate of the world.

execute/1 is an action operator to be used to represent the execution of an action represented by an object expression. (`execute` *`ActionDesignator`*) represents the execution of the action represented by *ActionDesignator*. *ActionDesignator* must be an object expression. That is, it must be an expression whose evaluation returns an object, for instance a referential expression or a functional expression representing an object.

2.2 Information Message

This section shows an interaction in which a restaurant agent receives the information that there is a traditional food restaurant named "Encher a Mula" with phone number 219999999, located in Lisbon, downtown, street "Associado Dias", number 1. In FIPA ACL, information messages use the *inform* performative.

```
(inform
  :sender Some restaurant SME access agent
  :receiver Lisbon restaurant agent
  :content "(
    (instance
      (Restaurant
        :name \"Encher a Mula\"
        :phone 219999999
        :restaurantAddress (Address
           :publicPlace \"Rua Assoc. Dias\"
           :number 1
           :city Lisboa
           :zone downtown)
        :type traditional)
      Restaurant)
   )"
   :language extended-FIPA-SL
)
```

Figure 2. Information Message

The message in figure 3 is expressed in the FIPA ACL communication language and FIPA SL content language.

2.3 Closed Query Example

This section presents an example in which an agent receives a closed query. In FIPA ACL, closed queries are expressed by the *query-if* performative.

In this example, the restaurant information agent receives a query representing the question *"Is there a traditional restaurant located downtown?"*.

```
(query-if
  :sender Some Personal Assistant
  :receiver Lisbon Restaurant Agent
  :content "(
    (exists ?r (exists ?a
      (and
        (instance ?r Restaurant)
        (= (value ?r type) traditional)
        (= (value ?r restaurantAddress) ?a)
        (= (value ?a zone) downtown))))
   )"
   :language extended-FIPA-SL
   :ontology AgentcitiesRestaurantOntology
   :reply-with query01
)
```

Figure 3. Closed Query Message

2.4 Opened Query Using a Relation

This section describes two approaches to represent relations between objects. In the first case, the relation is represented by a class of the domain ontology. In the second case, the relation is represented by a method.

Figures 4 and 5 show the two ways of representing the query "What are the names and addresses of downtown traditional restaurants that are better than 'Encher a Mula'?".

In the first case, we assume the existence of a domain class named Better with two attributes: worse and best, which are restaurants. Actually, the class should also take another argument specifying the comparison criterion but, for the sake of simplicity, we omit such details here. The instances of this class represent pairs of restaurants in which one is better than the other (according to some criterion).

```
(query-ref
  :sender Some personal assistant
  :receiver Lisbon Restaurant Agent
  :content "(
   (all (sequence (value ?r1 name) ?address)
     (exists ?r1 (exists ?r2
     (and
        (instance
           (Better :best ?r1 :worse ?r2)
           Better)
        (=(value ?r2 name)\"Encher a Mula\")
        (=
           (value ?r1 restaurantAddress)
           ?address)))))
    )"
  :language extended-FIPA-SL
  :reply-with query02
)
```

Figure 4. Opened Query Message

```
(query-ref
  :sender Some personal assistant
  :receiver Lisbon Restaurant Agent
  :content "(
    (all
     (sequence
        (value ?r1 name)
        (value ?r1 restaurantAddress))
     (exists ?r1 (exists ?r2
        (and
           (instance?r2 Restaurant)
           (=
            (value ?r2 name)
            \"Encher a Mula\")
           (=
            (value
              ?r2
              betterThan
              (sequence ?r1))
            true)))))
    )"
  :language extended-FIPA-SL
  :reply-with query03
)
```

Figure 5. Opened Query Message

In the second example (figure 5), the Restaurant class has a method called *betterThan* that is used to check whether or not the object to which it is applied is better than the method single argument. *betterThan* returns true or false. Actually, betterThan should receive a second argument specifying the comparison criterion but, for the sake of simplicity, we won't consider it here.

2.5 Request to Perform an Action

The FIPA ACL *request* performative is used for an agent to ask another one to perform a given action. In this section we consider two scenarios. In the first scenario, the requested action is represented in the receiver's ontology by a class. In the second scenario, the requested action is represented by a method.

In both cases, some personal assistant asks a restaurant representative agent to book a table for 10 people to have dinner at 8 PM.

```
(request
  :sender Some personal assistant
  :receiver The restaurant representative
  :content "(
    (action
      The restaurant representative
     (execute
          (BookTable
               :number_of_people 10
               :dinner_time 8PM)))
   )"
  :language extended-FIPA-SL)
```

Figure 6. Request to perform an action.

In the following case, we assume that the booking action is represented by the class called *BookTable*, which has two attributes: number of people and dinner starting hour.

In the second example, we assume the ontology used by the restaurant representative agent has the class *Table*, which contains the method *book* taking two arguments: the number of people and the dinner starting time.

```
(request
  :sender Some personal assistant
  :receiver The restaurant representative
  :content "(
    (action
     The restaurant representative
      (apply
       (any ?table
         (and
            (instance ?table Table)
            (>=
               (value ?table num-of-seats)
               10
            )
            (member
               8PM
               (value ?table free-slots))))
       book
      (sequence 10 8PM))
   )"
  :language extended-FIPA-SL
)
```

Figure 7. Request to perform an action.

In the above message, the first argument of the *apply/3* operator is a referential expression that represents any table with more than 10 seats that is not reserved for 8 PM. The second argument is the name of the method used to book the selected table. The third argument is the sequence of arguments of the book method: number of people and dinner starting time.

3. FROM AGENT MESSAGES TO OBJECT DATABASE INTERACTIONS

In our proposal, agents are built on top of an object database. The object database management system has an interface compatible with the ODMG 3.0 object model therefore the best way to interact with it is through OQL, the ODMG Object Query Language [3].

Since our agents communicate through the exchange of FIPA ACL messages with extended FIPA SL contents, ACL/SL messages are translated to OQL commands. This section describes the translation of ACL/SL messages to OQL commands.

The same approach could easily be used for other content and query languages, such as KIF and XQuery.

As the agent internal database doesn't have exactly the same model as the domain ontology, the translation process uses a Data Dictionary that maps from the domain ontology classes and attributes to the internal database classes and attributes. This way the generated OQL command uses the internal database classes, attributes and data types. Besides mapping from the domain ontology to the internal data model, the Data Dictionary allows the agent designer to define his or her own database actions, such as *create_new_restaurant*. Finally, the Data Dictionary is used to check attribute and method types. This is important, for instance, to know when to use commas around string values.

In order to facilitate the creation of the Data Dictionary for an agent, we have developed a Graphical User Interface that looks into the database, and automatically creates part of the dictionary. However, it is not possible to automatically create the whole dictionary content, such as database actions and pre-defined queries. Those are easily introduced by the agent designer through the GUI.

The translation of ACL/SL messages into OQL commands is a four-step process. In the first step, the message string, which is an S-Expression, is converted into a parse tree representing the structure of the S-Expression. In the second step, the parse tree representing the S-Expression is converted into an object structure representing the ACL message and its extended SL content. SL expressions are validated during the first two steps.

In the third step, the object structure representing the ACL/SL message is converted into another object structure, representing the OQL query. Finally, in the fourth step, the method `toString()` of the created objects is applied to produce the final string format of the OQL command, which is passed to the object database.

One of the advantages of having a stepwise translation process is that it allows the agent designer to use only the steps found appropriate in each situation. In some circumstances it may be useful to be able of using only the third step of the translation. There are messages, such as `request-when`, `request-whenever`, `cfp`, and `propose` that cannot be translated to OQL because they are not queries. However, some of the translation steps may be used to process parts of the message. The first and second steps can always be used to decompose the message and its contents in its several parts. For instance, steps 1 and 2 can be used to isolate the condition part of action-condition expressions. Once the condition part has been isolated, the agent can then use steps 3 and 4 to determine whether or not the condition is true. It may also be possible, at least in the cases of database actions, to use the translator to create the database command that implements the action.

Another advantage of the stepwise translation is that it is easier to trace the origin of possible errors and hence to determine reason part of the contents of failure messages.

Usually, the translation from ACL/SL to OQL is straightforward. Domain independent constructs such as conjunction, negation and disjunction are directly mapped onto OQL conjunction, negation and disjunction. Domain entities such as class names and attributes are easily converted using the Data Dictionary, which contains a one-to-one mapping from the ontology to the internal database. However, some aspects of the translation are not as straightforward as applying the mapping represented by the Data Dictionary.

1. Extended SL structures with format "(`instance` *Object Class*)" are converted into OQL from-clauses with pattern "`from` *Object in i(Class)*", in which *i(Class)* is the internal representation of the class *Class* of the domain ontology.

2. Extended SL structures with format "(`value` *Object Attribute*)" are converted into OQL object expressions with pattern "*Object*.i(*Attribute*)", in which *i(Attribute)* is the internal representation of the attribute *Attribute* of the domain ontology.

3. Extended SL structures with format "(`value` *Object MethodName Arguments*)" are converted into OQL method invocation expressions with pattern "*Object.i(Attribute)***(***i(Arguments)***)**", in which *i(MethodName)* and *i(Arguments)* are the internal representation of *MethodName* and

Arguments of the domain ontology. In these mappings, some of the arguments such as constants are not subject to translation.

4. Extended SL structures with format "(`apply` *Object MethodName Arguments*)" are also translated to method invocation expressions exactly as *value/3* expressions.

5. The translation of extended SL structures with format (`execute` *ActionDesignator)* makes intensive use of the Data Dictionary, which contains the explicit parameterised templates of this kind of action-expressions. The translator just has to use an instantiation of the translated parameterised template stored in the Data Dictionary.

Some SL content expressions are not directly translated to OQL. First they are converted to an equivalent format, which is more easily translated to OQL. Universally quantified questions are converted to equivalent existentially quantified expressions, using the definitional equivalence $\forall_x P \equiv \neg\exists_x\neg P$. Implications are converted to equivalent disjunctions by the definitional equivalence $P \Rightarrow Q \equiv \neg P \vee Q$. Equivalence is also converted to another format using the relation $P \Leftrightarrow Q \equiv (P \Rightarrow Q) \wedge (Q \Rightarrow P) \equiv (\neg P \wedge \neg Q) \vee (P \wedge Q)$. Double negations are also simplified before translation to OQL.

In FIPA ACL there are two kinds of querying messages: *query-if* and *query-ref*, used for closed and open queries respectively. The content of a *query-if* message is a proposition. The equivalent OQL command must determine whether or not that proposition is true. The content of a *query-ref* message is an identifying referential expression. The equivalent OQL command must return the set of values that satisfy the specified condition.

6. SL structures with format "(*ReferentialOperator* ?var Proposition)" are converted into OQL commands with pattern "select OQLVar[.AttributeName] from OQLVar in VarDomain where OQLConditions", in which VarDomain can be a class name, or an attribute represented by a nested select.

This rule is just a simplification of the actual rule, since the first argument of a referential operator can be any term, for instance a sequence of variables.

The translation of referential expressions is independent of the specific referential operator. The differences between referential operators are handled when the result set returned by the database is converted into a message. If the used referential operator is `iota`, the result set must contain exactly one record. If it contains more or less than one record, a failure message is generated. If the used referential operator is `any`, the result set must contain at least one record. If the result set is empty, a failure message is generated. Otherwise, the reply will contain the first record contained in the result set. If the used referential operator is `all`, the result set is sent in the reply message, even if it is the empty set.

When the received query is successfully processed, the receiver replies with an *inform* message. In the case of open queries <s, query-ref(r, *ReferentialExpression*)>, the reply is <r, inform(s, (= *ReferentialExpression* i^{-1}*(ResultSet)*))>, in which i^{-1}*(ResultSet)* is the domain ontology representation of the internal result set. In the case of closed queries <s, query-if(r, *Proposition*)>, the reply is <s, inform(r, *Proposition*)> if the result set is not empty and <s, inform(r, (not *Proposition*))> if the result set is empty.

Unsuccessfully processed queries are replied with failure messages.

In the following example, it is assumed that the domain ontology and the internal data model are identical and include the class `Restaurant` with attributes `name`, `restaurantMenu` (set of instances of the class `MenuItem`) and `restaurantAddress`; and the class `MenuItem` with attributes `dish` and `price`.

The message that is translated in the example (see Figure 8) is the query "I want the names and the prices of the dishes cheaper than 10 Euro available in the restaurant named `Encher a Mula`".

```
(query-ref
   :sender Some personal assistant
   :receiver Lisbon restaurant agent
   :content "(
     (all
       (sequence (value ?item dish) ?price)
       (exists ?restaurant (exists ?item
          (and
             (instance ?restaurant Restaurant)
             (=
                \"Encher a Mula\"
                (value ?restaurant name)
             )
             (member ?item (value
                ?restaurant restaurantMenu)
             )
             (= ?price (value ?item price))
             (< ?price 10)))))
   )"
   :language extended-FIPA-SL
)
```

Figure 8. Query-ref message

The result of translating the above message to an OQL command is shown bellow.

```
select item.dish item.price
   from item in select r.restaurantMenu
               from r in Restaurant where r.name =
               "Encher a Mula"
   where item.price < 10
```

Figure 9. OQL command

It has been suggested that object oriented databases are not as popular as they used to be, hence an XML query language should be used instead. XQuery could certainly be an alternative. In that case, we would have to translate the received messages to XQuery instead of OQL.

4. ALTERNATIVE APPROACHES

In this section we describe three alternative approaches to the proposal presented in this paper. In the first alternative, the communication language is extended with new performatives so that the agent can send information messages whose content is an object.

In the second alternative approach, it is assumed that certain ontology entities represent appropriate types of communication entities. For instance, class C of the ontology plays the role of a proposition in the communication.

Finally, the third alternative approach considers the implementation of the agent on top of a relational database instead

of an object database. Then it uses transposition rules to map from queries addressing an object database into queries addressing a relational database.

4.1 Extending ACL to Handle Objects

In [2], Botelho and Ramos present an extension of the FIPA ACL language with three new performatives to be used with objects: present-object, ask-object, and subscribe-object.

present-object takes an object as content. It is used when the receiver wants to present an object to the receiver. The sender may reasonably assume that, upon receiving the message, the sender will believe the message content to be an existing object of the specified class. *present-object* is similar to the *InformRef* performative proposed in [4].

ask-object is used when the sender wants the receiver to send it the object that satisfies a given condition. *ask-object* takes a referential expression as content, as is the case with *query-ref*. However, *query-ref* is used when the sender wants the reply to use the *inform* performative, whereas *ask-object* is used when the sender wants to receive a *present-object* message.

subscribe-object is the persistent version of *ask-object*.

According to [2], agents receiving information messages containing objects would create beliefs about the class and the attributes of the received objects. Those beliefs could be used to answer questions about the received objects.

The approach described in this section involves extending the FIPA ACL language with three new performatives. Additionally, it involves extending the first order predicate calculus with a set of operators and new inference rules directed at reasoning about objects.

The approach presented in this paper involves only the extension of the content language with new operators, preserving the general semantics and properties of the language. If simpler solutions are preferred to more complex ones, then the proposal presented in this paper would be preferred.

4.2 Representation Assumptions

A possible approach would be to assume that all classes in the ontology would represent predicates in the agent communication. If this assumption could be made, the name of an ontology class would be mapped into the name of a communication predicate. The attributes of an ontology class would be mapped into the arguments of a predicate used in the communication.

This approach has severe problems though. For instance, we may want to use objects (that is, class instances) as arguments of predicates therefore some ontology classes would have to be mapped into communication classes while some other ontology classes would have to be mapped into communication functions (e.g., class constructors).

Even if the above problem could be surpassed, what about the representation of functions and actions? If classes always represent predicates, then functions and actions must both be represented by methods. Therefore we must be able of saying which methods represent functions and which ones represent actions. The conclusion is that, in the general case, the harmonisation of object-oriented ontologies with logic-based communication cannot be handled by general implicit assumptions. We need the means to specify which communication entity types are played by each ontology entities.

In [4], Cranefield and Purvis present an approach for the integration of logic-based communication with object-oriented ontologies. The approach defines a meta-model of general content languages. This meta-model defines entities as propositions, definite descriptions and ground terms, and specifies several relations among them. The approach defines the communication role played by each class in the domain ontology. For instance, the instances of a certain class may play the role of propositions used in the content language while the instances of another class may play the role of definite descriptions used in the content language and so forth.

This is a very promising approach but it does not solve all problems yet. First of all, it does not handle all kinds of object-descriptions (it handles only definite descriptions and functional expressions representing value type terms). Second, it does not handle action terms. Finally, and possibly more importantly, it does not specify the semantics of propositions represented by objects. How does an agent know that such a proposition is true? However the authors are working on some of these issues.

At least while the approach described in [4] does not solve the referred problems, the proposal presented in this paper seems to be preferable since it does not exhibit any of the mentioned limitations.

McDermott and co-authors [7] present an object-oriented ontology (using DAML+OIL) of PDDL, a language of the first order predicate calculus[1].

That ontology defines entities such as propositions, functional expressions, predicates, and functions. This work suggests yet an alternative approach. This would use the ontology defined by McDermott and others as an ontology representation framework. Using the new framework, the domain could be modelled in terms of predicates, functions, functional expressions and propositions. We have two objections against this approach. First we would not be using an object-oriented framework for representing the domain ontology, as decided by the Agentcities project. The model of the domain would be treated as a second category component among agent technologies. Second, if we are modelling the domain using a logic-based framework, we should use an existing logic-based ontology representation framework such as Ontolingua, instead of creating a new one on top of DAML+OIL.

4.3 Transposition Rules

Sections 4.1 and 4.2 presented two alternative approaches at the level of the integration of object-oriented ontologies with logic based agent communication. This section presents an alternative at the implementation level. In our proposal, the agent domain information is stored in an object database. In the alternative approach described in this section, the agent information is stored in a relational database. This approach has the advantage of using a more stable and well-supported technology.

[1] www.cs.yale.edu/~dvm/daml/drsonto.daml

Given this alternative, received messages must be translated into SQL queries, instead of the OQL queries of our approach. Since the domain ontology is an object-oriented ontology, the communication will refer to the classes, objects, attributes and methods of the ontology (see section 2), while the agent database is organised around relations, fields and tuples. Therefore, translating messages into SQL commands is not a straightforward process since the models do not directly match. First, we have to understand the translation between the two models.

In order to automate the process of translating messages containing classes, objects and attributes, two kinds of rules are necessary:

1. Ontology Mapping Rules: map an object oriented ontology representation framework into a relational model; and

2. Query Mapping Rules: map queries containing classes, objects, attributes and methods into SQL queries. The Query Mapping Rules are derived from the Ontology Mapping Rules.

Some ontology mapping rules have already been defined using a computational formalism [9], in the context of the UML (Unified Modelling Language) object model [1]. Below, we informally present three examples.

a) Each class is mapped into a relational table;

b) If class C has been mapped into table T, a scalar attribute of C is mapped into an attribute of the relational table T;

c) If class C has been mapped into table T, a collection attribute of C is mapped into a new relational table that inherits, as foreign key, the primary key of table T.

From a complete set of ontology mapping rules like the above three examples, it would be possible to create a set of query mapping rules that could be used to automatically translate received query messages into SQL commands.

The alternative approach briefly sketched in this section could in principle be used, however only the model mapping rules for mapping object-oriented models into relational models have been formalised. To the best of our knowledge, there is no documented set the rules for mapping queries addressing object-oriented databases into queries addressing relational databases. Therefore, the approach presented in this paper seems to be more reliable than the hypothetical alternative approach briefly described in this section.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a comprehensive approach that allows the integration of object-oriented ontologies, logic-based agent communication, and object-oriented internal databases. The proposal comprises two levels. The first level handles the integration of object oriented domain ontologies with logic-based agent communication. The second level deals with the integration of communication with internal object-oriented databases.

The integration of object oriented domain ontologies with logic-based agent communication was achieved through extending logic content languages such as KIF and FIPA SL in order to enable expressing propositions from object-oriented domain ontologies. The key for this was to use only general-purpose predicates, functions and actions. All the domain dependent concepts are represented by classes, objects, attributes, and methods. Contrarily to the proposal described in [4], our approach does not require an explicit mapping between the domain ontology and the communication. Classes, objects, attributes and methods of the ontology are all handled as terms in the communication.

The integration of logic-based messages with internal object-oriented databases is achieved by a translation process, which relies in a Data Dictionary representing a one-to-one mapping from the domain ontology entities into the internal database entities. Since agent communication directly refers to object-oriented entities and the agent internal information is also stored in an object-oriented database, the translation is not difficult. The main difficulties are due to expressiveness differences between logic content languages and object oriented database query languages.

Our proposal grants agent designers considerable modelling freedom. They may chose to represent domain relations, functions and actions both by objects and by methods.

This paper contributes an elegant, comprehensive, robust, flexible, and simple approach to gather together the Greeks and Trojans of the modelling arena. Domain models do not loose their object-oriented lineage, agent communication also preserves its logic-based background, and none is treated as a second category component among agent technologies as it would be the case in approaches such as [7]. Our current proposal is simpler, less restricted, or fairer than other related approaches.

We envisage yet two ontology-related future problems. Sometimes the domain ontology is so large that it is not practical to have a single agent responsible for maintaining all the described information. Sometimes, it is covenant that the domain ontology is divided into smaller ontologies, each one maintained by a specific agent. In this case, we need to find a systematic way to provide an integrating interface between the several agents maintaining parts of the ontology and the remaining of the multi-agent system, which is only prepared to handle the domain as whole.

Since agent societies will be constantly growing, new services being created by different agent development teams, it will be impossible and undesirable to ensure that there will not exist agents with different underlying domain ontologies. The second and also more difficult future problem will be overcoming such ontology mismatches.

6. AKNOWLEDGEMENTS

The research described in this paper is partly supported by the EC project Agentcities.RTD, reference IST-2000-28385 and partly by UNIDE/ISCTE. The opinions expressed in this paper are those of the authors and are not necessarily those of the Agentcities.RTD partners. The authors are also indebted to all other past and current members of the Agentcities ADETTI team.

7. REFERENCES

[1] Booch, G.; Rumbaugh, J.; and Jacobson, I. The Unified Modeling Language User Guide. Addison-Wesley Publishing Company. 1999

[2] Botelho, L.M.; and Ramos, P. Extending the FIPA ACL Language. From Object Based Descriptions to Relational Representations. Proc. of the Workshop on Distributed

Artificial Intelligence and Multi-Agent Systems (DAIMAS2000) 2000

[3] Cattell, R.G.G; and Barry, D. The Object Data Standard: ODMG 3.0. Morgan Kaufman Publishers. London. 2000

[4] Cranefield, S.; and Purvis, M. A UML profile and mapping for the generation of ontology-specific content languages. Submitted. 2002

[5] Foundation for Intelligent Physical Agents. FIPA Communicative Act Library Specification. Specification Document XC00037H. 2001

[6] Foundation for Intelligent Physical Agents. FIPA SL Content Language Specification. Specification Document XC00008G. 2001

[7] McDermott, D.; and Dou, D. 2002. Representing Disjunction and Quantifiers in RDF. In Proc. of the Semantic Web Conference. Forthcoming. 2002.

[8] National Committee for Information Technology Standards. Knowledge Interchange Format: Draft proposed American National Standards. *Technical Report* NCITS.T2/98-004. 1998. http://logic.stanford.edu/kif/dpans.html

[9] Rio, L.; and Ramos, P. Mapping Object Oriented Models into Relational Models: a formal approach. In Proc. of International Conference on Infrastructure for E-Business, E-Education, E-Science, and E-Medicine. Forthcoming. 2002

[10] Searle, J.R. Speech Acts. Cambridge University Press. 1969

[11] van Harmelen, F.; Patel-Schneider, P.F.; and Horrocks, I. Reference Description of the DAML+OIL (March 2001) Ontology Markup Language. DAML+OIL Document, URL 2001. http://www.daml.org/2000/12/reference.html

[12] Willmott, S.; Dale, J.; Burg, B.; Charlton, P; and O'Brien, P. Agentcities: a worldwide open agent network. Agentlink News, 2001. 8:13-15