Towards HARMON*IA*: automatic generation of e-organisations from institution specifications

Daniel Jiménez Pastor
Department of Computer Science
University of Bath
BATH BA2 7AY, United Kingdom
dani@pitaweb.com

Julian Padget
Department of Computer Science
University of Bath
BATH BA2 7AY, United Kingdom
jap@cs.bath.ac.uk

ABSTRACT

There is a large gap between the emerging theory of institutions, or to be more accurate the norms that characterize them, how agents might reason about a symbolic representation of those norms and software tools to realize agent-trading platforms from high level specifications. We present an initial representation for institutions written in XML and their ontologies written in DAML+OIL and show how it has been used to specify aspects of simple auction house (the FishMarket [12]) from which we are able to generate automatically the components of the performative structure and the agent skeletons for an implementation on top of the JADE platform.

Keywords

Agents, ontology, multi-agent systems, institutions, norms, agent-trading platforms, institution definition language, performative structure, dialogical framework.

1. INTRODUCTION

The work presented here is a continuation of the ideas first reported in [22], in which a view was put forward of how implementations of agent organizations[1] could be generated from high-level descriptions of the structure and their norms. The content of [22] was preliminary in nature, in that the institution specification language was used as a guide for the manual development of the skeletal JADE [2] components, starting from the a description of the FishMarket [19, 12, 18] written in the Islander language [5, 4]. The aim of that paper was to extract parameterizable skeletal software components which might then be re-used in the construction of new institutions, and thus making a first step towards a tool for the rapid prototyping of institutions from their specifications. Since the main goal of the software tools featured here is the automatic generation of any kind of institution, we begin by completing the parameterizable components extracted from the FishMarket implementation, and then move on to describe progress towards creating a generic tool for the generation of e-organizations.

There are few tools in this area at the moment. One is the Islander agent institution graphical specification tool [7], which addresses aspects of organizational structure and dynamics. At a more technical level, there is the Bean Generator [17] plug-in for Protégé [16], which is used to obtain the Java files for the JADE platform [8] from an ontology specification. A more comprehensive solution, similar in scope to ours is the frame-based ontology language developed by Poggi *et al* [15], where a distinct ontology language inspired by KIF has been defined for use in the context of JADE and from which the implementation of agent systems in JADE has been generated. This is the closest in vision to what is described here. The most notable difference is their use of a bespoke, and somewhat simpler, ontology language in contrast to our use of DAML+OIL. The gap we are attempting to fill is the provision of an *integrated* tool for the automatic generation of an agent organization with its associated ontologies from its specification while aiming to utilize and to comply with current and emerging Semantic Web Group standards [24].

There are three aspects to the tools we are developing: (i) the overall HARMON*IA* framework which unifies notions of norm, rule, procedure and policy [5, 10, 21] and within this (ii) the specification of ontologies for concepts related to the organization, for example auction house, and for concepts related to the domain or domains about which agents may discourse, for example fish (!) (iii) the generation of agents to populate the organization and act as proxies for external agents wishing to interact with the organization. Thus we are motivated to attempt to create a complete tool for generating both the ontologies and the agents of the organization, so that the generated agents could be designed to use only the desired ontology, among other advantages. In the context of HARMON*IA* we foresee the chance to generate e-organizations for different agent platforms, but at this stage we are using our own IDL (Institution Definition Language) rather than Islander, JADE as the target platform and DAML+OIL [3] as the ontology language.

For the medium term, we are considering how we may target a range of agent platforms [2] and therefore a richer IDL will be required to accommodate these demands. Also, we are preparing for migration to OWL [23], shortly expected to be approved by W3C. In the longer term, a GUI tool for the graphical creation of the institution specification is planned as part of the HARMON*IA* framework (currently under development).

[1] Although it has become conventional to write of agent institutions, both for kinds of institutions and their implementations, we feel it is important to make the distinction between institutions as classes and organizations as instances (see [5] for a diagrammatic presentation of this argument), which is consistent with the defining work on the subject from an economic perspective by Douglass North [13]

[2] Despite the fact we have chosen JADE as agent platform, due to the large user community, the scope for wide deployment of generated organizations and compliance with FIPA standards [6], there is an large list of agent platforms as seen at [1], and among them we highlight the following: FIPA-OS, AAP and Zeus.

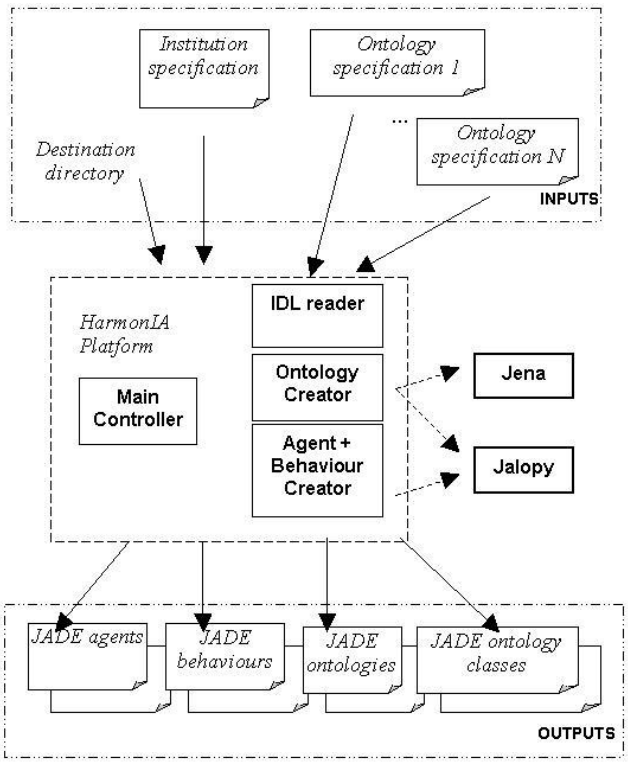

Figure 1: Elements of the HARMON*IA* architecture for the JADE platform

2. SYSTEM ARCHITECTURE

We begin with a description of the overall architecture of the toolset we are constructing – see figure 1 for an overview of the components. The organizational structure is firmly based on the work previously reported in Noriega's thesis [12] and subsequently expanded by Rodriguez Aguilar [18]. The final HARMON*IA* platform will contain much more than is shown here, since it will encompass norm specification and verification tools covering the range from norm through rule to protocol and appropriate theorem provers and model checkers as required.

2.1 Generic Institution Design

The idea norms that characterize institutions and the view of organisations as instantiations of institutions satisfying those norms are both increasingly accepted as important aspects in the engineering of agent-based systems. The particular approach we take began with the exposition by Noriega [12], and later Rodriguez Aguilar [18], of a dialogical specification of agent interaction via sets of illocutions (speech acts), the identification of interaction sequences (protocols) as so-called scenes and the linking of scenes to make a graph called the performative structure. Subsequently these ideas have been captured in the ISLANDER institution description language [5] and a graphical toolkit [4]. At the same time, the original ideas first put forward in the ISLANDER language have been refined to focus more on norms as guides and constraints on agent behaviour [14, 10] in contrast to the somewhat rigid conversation structures conceived originally. What we are presenting here are some of the preliminary thoughts on a second generation institution description language in which we are moving from the ad-hoc syntax of ISLANDER to a widely recognized representation and from the monolithic structure of ISLANDER specifications to one of composable components in conjunction with the development of the translation schemes from specification to implementation that were first sketched by Vickers [22].

The construction of an electronic organization begins with a description written according to the XML schema we have developed for the purpose. The corresponding ontology to ground the elements in this description is currently under construction.

Each institution has one or more dialogical frameworks, which determine the illocutions that can be exchanged between the agents in each scene. In order to do so, an ontology written in DAML+OIL and a list of the possible roles that agents may play are defined for each dialogic framework, which fixes what are the possible values for the concepts in a given domain. This DAML+OIL ontology will be parsed using the Jena toolkit [11].

Each agent role(see later), can have one or more behaviours, each one able to contain multiple sub-behaviours, corresponding to the functionality of JADE platform agents.

2.2 System Overview and Dependencies

Before sketching the key phases in this part of the HARMON*IA* architecture, we will briefly mention two important third party APIs we use in our tool: (i) Jena [11] is used to read the DAML+OIL ontology and hence process the information to create all the files related to the ontologies, used in the JADE environment. (ii) Jalopy [9] is a source code formatter for Java and that is responsible for the layout of all the code presented later in this paper.

2.3 Inputs

There are three inputs to this basic instantiation of the HARMON*IA* framework:

- The name of the directory where the generated organization files will be written.
- The institution description – written in the XML-based IDL.
- One or more ontology files.

The institution description is derived from the ISLANDER language first described in [5], which develops the features described in [18] and which has forms part of the institution editor reported in [4]. Various shortcomings with the ISLANDER approach and representation have lead us to develop a second generation IDL based on DAML+OIL – with the intention to migrate to OWL [23] in the very near future. In addition, for the purpose of this first exercise, we have incorporated JADE platform specific features, see the example in Figure 2, because in the short term we are focusing on delivery on one platform. For the longer term, we are considering how the XML schema may be parameterized to support a range of platforms.

As seen in Figure 2, we follow the design set out in Section 2.1. We use this file to create the data structure of the institution for the JADE platform. Each dialogic framework, with the tag of the same name, has one ontology, and as we see in the code, the tag `Ontology` indicates where to find the ontology, the type of which is defined in the `specificationLanguage` attribute (this example uses a DAML+OIL ontology), so the system knows how to parse it.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Institution id="FM2003">
  <DialogicFramework id="FishDialFrw"
     contentLanguage="PROLOG">
    <Ontology id="FishONTO"
       specificationLanguage="DAML"
       file="file:///C:/oasPaper/fish.daml" />
    <Role id="Boss"
       type="InternalRole">
      <Behaviour id="Open"
         type ="SimpleBehaviour"/>
      <Behaviour id="Discuss"
         type ="CompositeBehaviour">
        <Behaviour id="InitialResolution"
           type ="OneShotBehaviour"/>
        <Behaviour id="FinalResolution"
           type ="OneShotBehaviour"/>
      </Behaviour>
    </Role>
    <Role id="Admitter"
       type="InternalRole">
      <Behaviour id="Admit"
         type ="CyclicBehaviour"/>
    </Role>
  </DialogicFramework>
</Institution>
```

Figure 2: XML based IDL file for the JADE platform

A word of clarification is in order about the term "role". Superficially, this can be thought of as a synonym for type, but what it actually captures is both deeper and more flexible. The principle of role is borrowed from a line of research in security – both in software and in physical organizations – called Role Based Access Control (RBAC) [20] which associates ideas of responsibility, constraint and obligation with a given role and captures relationships between roles, such as whether one subsumes another and whether one role is incompatible with another, such as because it is either impossible to fulfil the requirements of each role simultaneously or because their combination may create a security hole. From the norm perspective, where we recall that a norm is an expression of a guide or constraint on behaviour, it becomes clear that a role is a coherent subset of the norms of an institution that taken together prescribe the limits of action when playing a particular role. Thus it is that in describing an institution, a key part of the modelling process also identifies roles that agents may play within that institution (e.g. buyer, seller, admitter, accountant, auctioneer, etc., see Esteva *et al* [5] for more detail and Vazquez [10] for a comprehensive treatment).

Hence we use the word "role" to refer to the constraints on an agent's behaviour. From past practice, we have found it convenient to classify roles as one of two kinds: `InternalRole` if is a staff agent, responsible for aspects of running the institution, like the Boss or the Admitter agent in an auction or `ExternalRole` for a client-agent written by a third party that visits the institution, like the Buyer or Seller agents.

Finally, in the `Behaviour` tag, we have two attributes: the `id` and the `type`, which identifies the `Behaviour` class of the JADE platform from which we are inheriting.

As it stands, the IDL file contains enough information to describe the gross behaviour of the agent, but not how it communicates, for which we need the illocutions to be used in the conversation protocols of the scenes of the performative structure. These aspects are currently under development and will be reported on separately later.

To illustrate how the framework we have operates, instead of providing a long DAML+OIL ontology file, we will show the complete UML diagram representation of the DAML+OIL file in the Figure 3. Then, to show each aspect of the translation in detail, some DAML+OIL code will be posted.

Finally, we provide solutions for the problems or constraints found while dealing with the translation of a DAML+OIL ontology into a Jade ontology, due to the particularities of the Jade platform and model incompatibilities between DAML+OIL and Java, such as multiple inheritance, multiple ranges, anonymous classes and some aspects that will be discussed in the following subsections.

2.3.1 Multiple inheritance

MI is a key aspect of DAML+OIL, but is not a feature of Java, because a class can only inherit from one class and from multiple interfaces, and therefore only the methods inherited from the superclass and not those from the interfaces may have an implementation.

But as an ontology is basically data, not operations, we just need Java classes with their attributes and only accessor methods to represent it in the Jade platform. So, we can translate all the classes of the DAML+OIL ontology into interfaces to obtain the multiple inheritance capability, and then, for each interface, generate a wrapper class with the accessor methods to get and set their attributes. The only issue we have to control is to rename attributes from different parents if they have the same name. This solution will be fully working before summer 2003, but at present, if a class in the input has more than one parent, we will only consider the first and ignore the rest (with a suitable warning). Example:

```
<daml:Class rdf:ID="BabySquid">
  <rdfs:subClassOf rdf:resource="#Squid"/>
  <rdfs:subClassOf rdf:resource="#SmallFish"/>
</daml:Class>
```

Class BabySquid as specified above has two parents: Squid and SmallFish. In the translation, only Squid class will appear as the parent.

Some other solutions based on design patterns, as the Bridge, State and Strategy design pattern have been studied, but they provide solutions basically for different method implementations that work in a plain environment, but not embedded in a framework like Jade, because Jade can only know the accessor methods of the classes, but no others.

Another solution that requires the collaboration of the ontology designer would be using a delegation – "black-box" inheritance – approach, where BabySquid would have two attributes, one of class Squid and one of class SmallFish, but we discarded it because we want automatic generated solutions. We note that the ontology language of Poggi *et al* [15] provides single inheritance making the job of translation to Java/JADE somewhat less difficult than the task we have here in working from DAML+OIL.

2.3.2 Multiple ranges

In Java, each variable can be only of one type. This contrasts with the permitted multi-range properties in DAML+OIL. A way

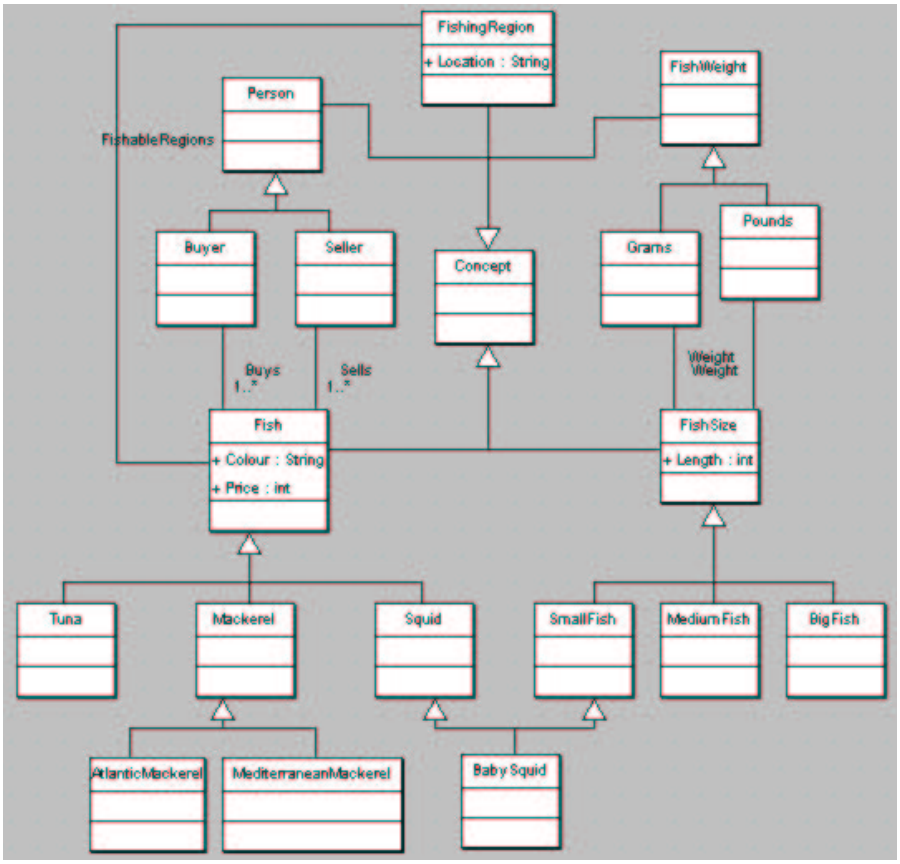

Figure 3: UML class diagram corresponding to the DAML+OIL ontology

to work around this in Java is to have many properties each with a different type instead of one property with many types. As with multiple inheritance, if this constraint is not observed, we only consider the first type given for that property and ignore the rest. Consider the following object property declaration:

```
<daml:ObjectProperty rdf:ID="Weight">
  <daml:domain rdf:resource="#FishSize"/>
  <daml:range rdf:resource="#Grams"/>
  <daml:range rdf:resource="#Pounds"/>
  <rdf:type rdf:resource =
     "http://www.w3.org/2001/10/daml+oil#UniqueProperty"/>
</daml:ObjectProperty>
```

Here, only Grams will be recognized as the possible range class (and therefore type) for the Weight object property. The conflict can be resolved by creating two different object properties for the FishSize class, named WeightGrams and WeightPounds respectively, as follows:

```
<daml:ObjectProperty rdf:ID="WeightGrams">
  <daml:domain rdf:resource="#FishSize"/>
  <daml:range rdf:resource="#Grams"/>
  <rdf:type rdf:resource =
     "http://www.w3.org/2001/10/daml+oil#UniqueProperty"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="WeightPounds">
  <daml:domain rdf:resource="#FishSize"/>
  <daml:range rdf:resource="#Pounds"/>
  <rdf:type rdf:resource =
     "http://www.w3.org/2001/10/daml+oil#UniqueProperty"/>
</daml:ObjectProperty>
```

Although the solution of splitting automatically a multiple range can be done easily, in the OWL specifications they explain that a multiple range has to be treated as the intersection of ranges, and as we explain later (see section 2.3.5) with respect to class union, intersection and negation, we do not generate any solutions for logic/set operators.

2.3.3 Identifier syntax

This is a perennial problem in any language translation task when the source language has a less restrictive identifier syntax than the target language. Thus, the input:

```
<daml:Class rdf:ID="Mediterranean-Mackerel">
  <rdfs:subClassOf rdf:resource="#Mackerel"/>
</daml:Class>
```

contains a variable which is not a valid as a Java variable because it contains a hyphen. The variable will be renamed to `Mediterranean_Mackerel` and a warning will be echoed with the new variable name. Also, variables cannot be reserved Java keywords, cannot have a hyphen as in the example, or start with a number. For all this cases, HARMON*IA* will rename the variable and print a warning.

Agent Boss imports the ontologies the agent uses

```
package FM2003;
import FM2003.Behaviours.*;
import FM2003.Ontologies.FishONTO.*;
```

Standard agent header, followed by initialization of properties and getting an instance of the ontology the agent uses.

```
public class Boss extends Agent {
    private ServiceDescription sd;
    private Codec codec;
    private Ontology ontology;
    private ContentManager manager;
    private DFAgentDescription dfd;
    private MessageTemplate mt;

    public Boss() {
        sd = new ServiceDescription();
        codec = new SLCodec();
        manager = (ContentManager) getContentManager();
        dfd = new DFAgentDescription();
        ontology = FishONTO.getInstance();
    }
```

Here we see the setup method, where the agent registers with the Service Descriptor as an internal agent and with the Directory Facilitator with FishONTO as ontology and FM2003 as ownership. After registering in the DF, creates and adds the behaviours Open and Discuss.

```
    protected void setup() {
        sd.setName(getName());
        sd.setType("InternalRole");
        sd.setOwnership("FM2003");
        dfd.setName(getAID());
        dfd.addServices(sd);
        dfd.addOntologies("FishONTO");
        try {
            DFService.register(this, dfd);
        } catch (FIPAException e) {
            System.err.println(getLocalName() +
             " registration with the DF failed because - "
               + e.getMessage());
            doDelete();
        }
        manager.registerLanguage(codec, "PROLOG");
        manager.registerOntology(ontology, "FishONTO");
        Open behaviour_0 = new Open(this);
        addBehaviour(behaviour_0);
        Discuss behaviour_1 = new Discuss(this);
        addBehaviour(behaviour_1);
    }
...
}
```

Figure 4: Boss agent imports, initialization and setup

2.3.4 Root classes

As we want to generate automatically an ontology for the JADE platform, each class must inherit from one of the following classes: AID, AgentAction, Concept or Predicate, which must also be defined in the ontology as we see below. We note that we have only as yet worked with a few simple ontologies and this will have far from demonstrated the full range of inputs – and problems – that we will need to be able to handle.

```
<daml:Class rdf:ID="AgentAction">
</daml:Class>

<daml:Class rdf:ID="AID">
</daml:Class>

<daml:Class rdf:ID="Concept">
</daml:Class>

<daml:Class rdf:ID="Predicate">
</daml:Class>

<daml:Class rdf:ID="Fish">
  <rdfs:subClassOf rdf:resource="#Concept"/>
</daml:Class>

<daml:Class rdf:ID="Mackerel">
  <rdfs:subClassOf rdf:resource="#Fish"/>
</daml:Class>
```

From the perspective of Java generation, AID, AgentAction, Concept and Predicate are interfaces. So, class Fish will extend the Concept interface, and class Mackerel will inherit from Fish class.

2.3.5 Union, intersection and negation classes

This is one feature of DAML+OIL that is far too powerful to be used at present with programming language such Java, so we cannot provide a solution for this capability, but we will continue researching all the possibilities to benefit from this richer expressiveness of DAML+OIL or OWL.

```
package FM2003.Behaviours;
import FM2003.*;
import FM2003.Ontologies.FishONTO.*;
```

As in the agent imports, we import their own ontology. And we create the behaviour inside the package Behaviours.

```
public class Discuss extends CompositeBehaviour {
    private InitialResolution subBehaviour_0;
    private FinalResolution subBehaviour_1;

    public Discuss(Agent a) {
        super(a);
    }
```

As is work for the programmer which order a behaviour uses their subBehaviours, we only add as many properties as subBehaviours the behaviour has, in this case one for InitialResolution and other for FinalResolution.

```
    public boolean checkTermination(
                      boolean currentDone,
                      int currentResult) {
        //This method must be implemented
        return false;
    }
    public Collection getChildren() {
        //This method must be implemented
        return null;
    }
    public Behaviour getCurrent() {
        //This method must be implemented
        return null;
    }
    public void scheduleFirst() {
        //This method must be implemented
    }
    public void scheduleNext(
                   boolean currentDone,
                   int currentResult) {
        //This method must be implemented
    }
}
```

Figure 5: Behaviour Discuss methods

We create a new package for every ontology, inside the Ontologies package. Then we write the imports, although we do not show them. Finally, we create many constants for the vocabulary, useful for the understanding of the code, although it is also possible not to create them, but the code would be much more untidy and less clear.

```
package FM2003.Ontologies.FishONTO;
...
public class FishONTO extends jade.content.onto.Ontology {
    protected static Ontology theInstance = new FishONTO();

    // Vocabulary
    public static final java.lang.String FISH = "Fish";
    public static final java.lang.String FISH_FISHABLEREGIONS = "FishableRegions";
    public static final java.lang.String FISH_PRICE = "Price";
```

We create all the primitive schemas at the beginning to make the code nicer, like the vocabulary. Then, we add the concepts classes to the ontology. We only show how to add the concepts, but the AID, AgentAction and Predicates can be added the same way.

```
    protected FishONTO() {
        super("FishONTO", BasicOntology.getInstance(),
            new ReflectiveIntrospector());

        try {
            PrimitiveSchema stringSchema = (PrimitiveSchema) getSchema(BasicOntology.STRING);
            PrimitiveSchema integerSchema = (PrimitiveSchema) getSchema(BasicOntology.INTEGER);
            PrimitiveSchema floatSchema = (PrimitiveSchema) getSchema(BasicOntology.FLOAT);
            PrimitiveSchema dateSchema = (PrimitiveSchema) getSchema(BasicOntology.DATE);
            PrimitiveSchema booleanSchema = (PrimitiveSchema) getSchema(BasicOntology.BOOLEAN);

            // adding Concept(s)
            ConceptSchema FishSchema = new ConceptSchema(FISH);
            add(FishSchema, Fish.class);
            ConceptSchema MackerelSchema = new ConceptSchema(MACKEREL);
            add(MackerelSchema, Mackerel.class);
```

After creating all the schemas for every class of the ontology, we add to them all their fields. We only show how to add fields to the Fish concept, to be brief.

```
            // adding Fields
            FishSchema.add(FISH_FISHABLEREGIONS, FishingRegionSchema, ObjectSchema.MANDATORY);
            FishSchema.add(FISH_PRICE, integerSchema, ObjectSchema.MANDATORY);
```

For all the schemas, we add the inheritance expressed in the model. With this, the creation of the ontology is done. We also add the getInstance method to get an instance of the ontology, used in the Agents files.

```
            // adding Inheritance
            MackerelSchema.addSuperSchema(FishSchema);
        } catch (java.lang.Exception e) {
            e.printStackTrace();
        }
    }

    public static Ontology getInstance() {
        return theInstance;
    }
}
```

Figure 6: Translation of the FishONTO ontology

2.3.6 Anonymous classes

Although it is fairly easy to implement this DAML+OIL functionality in Java by using the old Lisp GENSYM technique, at present we do not support DAML+OIL anonymous classes due to time constraints. These classes do not have meaning by themselves, rather they are used to capture abstractions embedded in other classes, and we are also mindful of the fact that OWL-Lite does not support anonymous classes, and depending on which version of OWL becomes the preferred means of expression, the problem could potentially go away.

2.3.7 Datatypes as types

Due to actual technical limitations of the Jena toolkit, and and bearing in mind future compatibility with OWL, the ontology input does not support the use of datatypes as types. In the next DatatypeProperty, we see that the range specifies a datatype, without referencing the integer schema. Thus, the following is incorrect:

```
<daml:DatatypeProperty rdf:ID="Length">
  <daml:domain rdf:resource="#FishSize"/>
  <rdfs:range>
    <xsd:integer />
  </rdfs:range>
  <rdf:type rdf:resource =
     "http://www.w3.org/2001/10/daml+oil#UniqueProperty"/>
</daml:DatatypeProperty>
```

The range part has to be rewritten with a reference to the integer schema:

```
<daml:DatatypeProperty rdf:ID="Length">
  <daml:domain rdf:resource="#FishSize"/>
  <daml:range rdf:resource =
     "http://www.w3.org/2000/10/XMLSchema#integer"/>
  <rdf:type rdf:resource =
     "http://www.w3.org/2001/10/daml+oil#UniqueProperty"/>
</daml:DatatypeProperty>
```

2.3.8 Split properties

Again, due to current technical limitations, we require the multiplicity information of the property to be supplied in the type attribute of the class tag, instead of having two tags: ObjectProperty/DatatypeProperty and UniqueProperty. So, all the information related to a property must be embedded in their tags, without splitting the information between siblings. For example:

```
<daml:DatatypeProperty rdf:ID="Price">
  <daml:domain rdf:resource="#Fish"/>
  <daml:range rdf:resource =
     "http://www.w3.org/2000/10/XMLSchema#integer"/>
</daml:DatatypeProperty>
<daml:UniqueProperty rdf:about =
   "file:/C:/oasPaper/fish.daml#Price"/>
```

Has to be expressed this way:

```
<daml:DatatypeProperty rdf:ID="Price">
  <daml:domain rdf:resource="#Fish"/>
  <daml:range rdf:resource =
     "http://www.w3.org/2000/10/XMLSchema#integer"/>
  <rdf:type rdf:resource =
     "http://www.w3.org/2001/10/daml+oil#UniqueProperty"/>
</daml:DatatypeProperty>
```

2.4 Outputs

The outputs of the system are the Java skeleton files of the organization for the JADE platform. All the files are created in the specified destination directory, while within the files, all the objects are in a package named by the institution ID. So, if the institution ID is `FM2003`, and we have as destination directory the root directory, and the code from Figure 2 as the IDL input file, we will get four kinds of files:

1. Agents – found in the root destination directory, belonging to the package FM2003:
   `FM2003/Boss.java`
   `FM2003/Admitter.java`
2. Behaviours – found in the Behaviours directory and belonging to the package FM2003.Behaviours:
   `FM2003/Behaviours/Open.java`
   `FM2003/Behaviours/Discuss.java`
   `FM2003/Behaviours/InitialResolution.java`
   `...`
3. An ontology file. In this case, is located inside package FM2003.Ontologies.FishOnto, and in a directory equal to the package name:
   `FM2003/Ontologies/FishONTO/FishONTO.java`
4. Ontology classes, located in the same package and directory as their own ontology file:
   `FM2003/Ontologies/FishONTO/BabySquid.java`
   `FM2003/Ontologies/FishONTO/Mackerel.java`
   `FM2003/Ontologies/FishONTO/Fish.java`
   `...`

We add this class to the same package as the ontology file. Again, we have omitted all the imports. As we see in the DAML+OIL examples, class Fish implements the Concept interface.

```
package FM2003.Ontologies.FishONTO;
...
public class Fish implements Concept {
    protected int Price;
    protected List FishableRegions = new ArrayList();

    public void setPrice(int value) {
        this.Price = value;
    }
    public int getPrice() {
        return this.Price;
    }
    public void addFishableRegions(FishingRegion elem) {
        List oldList = this.FishableRegions;
        FishableRegions.add(elem);
    }
    public boolean removeFishableRegions(
                                FishingRegion elem) {
        List oldList = this.FishableRegions;
        boolean result = FishableRegions.remove(elem);
        return result;
    }
    public void clearAllFishableRegions() {
        List oldList = this.FishableRegions;
        FishableRegions.clear();
    }
    public Iterator getAllFishableRegions() {
        return FishableRegions.iterator();
    }
    public List getFishableRegions() {
        return FishableRegions;
    }
    public void setFishableRegions(List l) {
        FishableRegions = l;
    }
}
```

Class Fish has two attributes: Price (single cardinality) and FishableRegions (multiple cardinality). Depending of the cardinality, we well create accessor methods to access a list or to a single type.

Figure 7: Ontology class Fish

We will now take a closer look at the contents of these files. We will start with the Boss agent file, although for brevity we omit the code that is common to every agent (takeDown method, JADE imports...). See Figure 4 and also the comments between the generated code fragments. Drilling down further we examine in more detail the behaviour Discuss, where again, for brevity, we have omitted the common code – see Figure 5 and the interpolated comments.

Depending on the JADE behaviour type we are inheriting, we must implement one or more additional methods. As this class inherits from CompositeBehaviour, me must implement the methods shown in Figure 5.

At present, we only create skeletons of the behaviour, depending on each behaviour type, but we shortly will complete the code for message passing between agents, extracted from the illocutions given in the institution description and then the bodies of these methods will also be created automatically.

Next, it is the turn of the ontology file, the details of which along with some commentary appear in Figure 6 and finally we have one of the classes of the ontology in Figure 7.

3. CONCLUSIONS AND FUTURE WORK

We have described the translation into Java (JADE) of the specification of aspects of an institution description written in DAML+OIL, noting at the same time various generic issues with respect to translation from DAML+OIL to Java. Because of the very active and changing nature of this area at the moment, a number of issues remain open, in particular, we expect very soon to move the IDL from DAML+OIL to OWL, more or less as soon as the support in the Jena API is released. We have also listed a number of aspects of DAML+OIL that could be, but are not translated yet, simply for lack of time, such as facets, anonymous classes or multiple inheritance. What we have presented here is the translation of the ontological aspects of electronic organizations: other aspects, such as the performative structure, scenes, transitions and conversation graphs are being finalized. Indeed, the completion of behaviour generation for the JADE platform is intended to be fully working before summer 2003, for which we will generate automatically the code of message passing between agents from the information extracted from the illocutions.

For the longer term, we are considering how we might support multiple target agent platforms. A new GUI tool is also under development which could be used to front end this, generating the OWL representation of the institution, which combined with schema verification tools opens up a path to round-trip engineering of institutional specifications.

4. ACKNOWLEDGEMENTS

Daniel Jiménez Pastor is a student of Computer Science Engineering at Facultat d'Informàtica de Barcelona, Universitat Politècnica de Catalunya (Spain). His work has been partially supported by an Agenticities scholarship while an Erasmus student at the Department of Computer Science, University of Bath (United Kingdom).

5. REFERENCES

[1] AgentLink – European Network of Excellence for Agent-Based Computing. `http://www.agentlink.org`, March 2003.

[2] F. Bellifemine, A. Poggi, and G. Rimassa. JADE — A FIPA-compliant agent framework. In *Proceedings of the 4th International Conference on the Practical Applications of Agents and Multi-Agent Systems (PAAM-99)*, pages 97–108, London, UK, 1999. The Practical Application Company Ltd.

[3] DAML+OIL – DARPA Agent Markup Language + Ontology Inference Layer. `http://www.w3.org/TR/daml+oil-reference`, March 2003.

[4] M. Esteva, D. de la Cruz, and C. Sierra. Islander an electronic institutions editor. In *Proceedings of The First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2002)*, 2002. to appear.

[5] M. Esteva, J. Padget, and C. Sierra. Formalizing a language for institutions and norms. In J.-J. Meyer and M. Tambe, editors, *Intelligent Agents VIII*, volume 2333 of *Lecture Notes in Artificial Intelligence*, pages 348–366. Springer Verlag, 2001. ISBN 3-540-43858-0.

[6] FIPA – The Foundation for Intelligent Physical Agents. `http://www.fipa.org`, March 2003.

[7] Islander, a graphical editor for institution specifications (software package). `http://e-institutor.iiia.csic.es/e-institutor/software/islander.html` (April 2002).

[8] Java agent development environment. `http://jade.cselt.it`, March 2003.

[9] Jalopy. `http://jalopy.sourceforge.net`, March 2003.

[10] Javier Vázquez-Salceda. *The role of Norms and Electronic Institutionsin Multi-Agent Systems applied to complex domains. The* HARMON*IA framework*. PhD thesis, Universitat Politècnica de Catalunya, 2003.

[11] Jena Semantic Web toolkit. `http://www.hpl.hp.com/semweb`, March 2003.

[12] P. Noriega. *Agent mediated auctions: The Fishmarket Metaphor*. PhD thesis, Universitat Autonoma de Barcelona, 1997.

[13] D. C. North. *Institutions, Institutional Change and Economic Performance*. Cambridge University Press, 1991.

[14] J. Padget. Modelling simple market structures in process algebras with locations. In L. Moreau, editor, *AISB'01 Symposium on Software Mobility and Adaptive Behaviour*, pages 1–9. The Society for the Study of Artificial Intelligence and the Simulation of Behaviour, AISB, 2001. ISBN 1 902956 22 1.

[15] A. Poggi, F. Bergenti, and F. Bellifemine. An ontology description language for FIPA agent systems. Technical Report DII-CE-TR001-99, University of Parma, 1999.

[16] Protege, an editor for ontologies (software package). `http://protege.stanford.edu/` (April 2002).

[17] Bean generator plug/in for protégé. `http://gaper.wi.psy.uva.nl/beangenerator`, March 2003.

[18] J.-A. Rodríguez. *On the Design and Construction of Agent-mediated Institutions*. PhD thesis, Universitat Autonoma de Barcelona, July 2001.

[19] J.-A. Rodríguez, P. Noriega, C. Sierra, and J. Padget. FM96.5 A Java-based Electronic Auction House. In *Proceedings of 2nd Conference on Practical Applications of Intelligent Agents and MultiAgent Technology (PAAM'97)*, pages 207–224, London, UK, Apr. 1997. ISBN 0-9525554-6-8.

[20] R. Sandhu, D. Ferraiolo, and R. Kuhn. The NIST model for role-based access control: Towards a unified standard. In *Proceedings of the 5th ACM Workshop on Role-Based Access Control (RBAC-00)*, pages 47–64, N.Y., July 26–27 2000. ACM Press.

[21] J. Vázquez-Salceda and F. Dignum. Modelling electronic organizations. accepted for the The 3rd International/Central and Eastern European Conference on Multi-Agent Systems -CEEMAS'03-, Prague, Czech Republic, June 2003, June 2003.

[22] O. Vickers and J. Padget. Skeletal JADE Components for the Construction of Institutions. In J. Padget, D. Parkes, N. Sadeh, O. Shehory, and W. Walsh, editors, *Agent Mediated Electronic Commerce IV*, volume 2531 of *Lecture Notes in Artificial Intelligence*, pages 174–192. Springer Verlag, December 2002.

[23] Word Wide Web Consortium (W3C). OWL – Web Ontology Language. `http://www.w3.org/TR/owl-ref`, March 2003.

[24] World Wide Web Consortium (W3C). Semantic web (daml+oil and owl working drafts). `http://www.w3.org/2001/sw`, March 2003.