Creating and Using Ontologies in Agent Communication*

Chris van Aart
Social Science Informatics
University of Amsterdam
Roetersstraat 15
1018 WB Amsterdam
The Netherlands
aart@swi.psy.uva.nl

Ruurd Pels
Acklin B.V.
Parkstraat 1a
4818 SJ Breda
The Netherlands
ruurd@acklin.nl

Giovanni Caire
Telecom Italia LAB S.p.a.
Via Reiss Romoli 274
10148 Turin
Italy
giovanni.caire@tilab.com

Federico Bergenti
AOT Lab and CNIT
Parco Area delle Scienze 181/A
43100 Parma
Italy
bergenti@ce.unipr.it

ABSTRACT

This paper presents an approach for the creation and use of ontologies to support agent communication. The use of ontologies in message based communication gives meaning to the contents of messages sent between agents. A preliminary upper ontology based on a content reference model is presented that provides the semantics for message content expressions. Furthermore, a tool is presented that can assist agent programmers in designing message content ontologies with the Protégé tool and export this to Java source code. A case study on international insurance traffic shows the process from ontology design to system architecture design.

1. INTRODUCTION

In this work, we present an approach for the creation and use of ontologies to support communication between agents. A lot of effort has been spent in the creation of mechanisms to transport messages between peers, for example CORBA, SOAP, XMLRPC and RMI. Furthermore, research in markup languages for information and knowledge exchange provided languages, such as KQML, KIF and FIPA-SL. However, we think that making services to collaborate through remote procedure calls, simply shifts the complexity of interoperability to end-users. For example, WDSL or DAML-S can express a relatively detailed view on the structure of (SOAP) interfaces to a service. However, when making use of this interface, one still has to work out all details dealing with the signature of the methods to call, behavior and exception handling. Furthermore, although web services are already deployed in various domains, many of these tend to be inflexible: it is not possible to modify the underlying system, to configure it for other domains, or to integrate different services to produce new functionalities.

One way to offer services in a flexible manner is by means of mediators. A software agent can act as a mediator capable of translating agent messages to proprietary instructions to access a service, i.e. it can act as a transducer [8]. A transducer maps instructions from agent to service and results from service to agent. This approach has the advantage that the agent does not have knowledge of the invocation of the services. These agents can be used to form new applications, not by means of integration, i.e. trying to couple different components with proprietary software, but by federation, i.e. standardizing interaction patterns. This means that agents do not have access to each other's code but they can request services by sending messages. By Federating systems, the emphasis lies on agent collaboration instead of service integration. Our view on these kind of distributed systems is that of an information processing and communication structured systems: the agents are seen as information processors (cf. [6]). Challenging problems arise when agents from different organizations have to collaborate with each other, instead of only providing (existing) information retrieval functions. An example of such collaborations is negotiation about price, quality or misinterpretations. One way to design *really* loosely coupled distributed systems, is to regard systems as purely message exchanging. To design such systems, the following questions arise:

- How should messages be generated, transmitted and represented ?
- How can the content of messages be standardized?
- What principles (e.g. concepts, mechanisms and patterns) can be used?
- What heuristics and guidelines are there those tell us when to apply what principle?

In Sec. 2 we briefly discuss message based communication between agents. Sec. 3 describes an approach for creating message content ontologies. In Sec. 4 we show the creation of a message content ontologies for a case study on international insurance traffic. Finally, we discuss some issues arising from this study and suggest future work.

*This work has been partially funded by the IST project IBROW, nr.-1995-19005, http://www.ibrow.org.

2. AGENT COMMUNICATION

Agent Communication can be defined as a form of interaction in which the dynamic relationship between agents is expressed through the intermediary of signals, which, once interpreted, will affect these agents [4]. As discussed by Ferber, a large number of (agent) communication forms exist [4]. Here we see the act of communication of sending some information from a sender to a set of (intended) receivers. This information is encoded with the help of languages and decoded upon arrival by the receivers. An advantage of this approach is that one can get loosely-coupled open systems, that only use message passing as vehicle for collaboration. Unfortunately, there is no standardized definition of what an (operational) agent is and how agents may communicate between with each other. In this paper we refer to the abstract communication model of FIPA [1] that derives from speech act theory [12]. In this model, communication occurs through the exchange of asynchronous messages corresponding to communicative acts. The ACL language format [2] defines the format of these messages. ACL messages can be characterized by:

- Intention, e.g., REQUEST, INFORM, QUERY_REF.
- Attendees, i.e. the sender and a set of receivers.
- A content, i.e. the actual information that is exchanged.
- Content description, i.e. an indication of (i) the content language used to express the content and(ii) the ontology by means of which both the sender and the receiver ascribe a proper meaning to the terms used in the content.
- Conversation control, e.g. interaction protocol and conversation identification.

```
<fipa-message act="INFORM" >
   <sender>
      <agent-identifier>
         <name id="Peter@host1:8888/JADE" />
      </agent-identifier>
   </sender>
   <receiver>
      <agent-identifier>
         <name id="John@host2:8888/JADE" />
      </agent-identifier>
   </receiver>
   <content>
     (weather today raining)
   </content>
   <language>English</language>
   <ontology>Weather-ontology</ontology>
   <conversation-id>Peter-John253781</conversation-id>
</fipa-message>
```

ACLMessage 1: Example message expressed in XML with intention INFORM; attendees Peter and John; content an expression about the weather; plain English as content language; an ontology about the weather; and conversation control

FIPA has also defined valid formats to represent an ACL message among which an XML based one. The XMLs can be found at the FIPA repository[3]. ACLMessage 1 shows an example in which agent Peter informs agent John that the weather today is raining. As shown in the example, ACL messages are built up of layers of languages. Elements in the world are defined in a *domain ontology*. A *content language* expression (i.e. the element between content tags) is used to represent statements of the world. Finally, a *speech act* (i.e. INFORM) as the agent's intention to describe or alter the world is wrapped around the content expression.

The three layers are also linked by constraints such as, the message expresses which content language, encoding and ontologies are used for the content. For example, when an agent sends a message containing a prolog expression, the language will be `Prolog`, the encoding `String` and the ontology `family-ontology`. Furthermore, the speech act used constrains the type of content allowed . For example, a message with the intention: request should have a content asking for an action and not an answer. For a more elaborated discussion, see [14].

The FIPA communication model, however, in order to preserve agent autonomy as much as possible, is based on the speech act as communication attempt idea: when a rational agent utters a speech act to another one, it is only trying to get a message through and it is not entitled to believe the rational effect of its utterance. It is clear therefore, that something more is needed to enable agents to communicate effectively.

That is, in order for agents to be able to reason about the effects of their communications, ACL messages could be inserted into proper Agent Interaction Protocols (AIP). AIPs describe communication patterns as allowed sequences of messages between agents and the constraints on the content of those messages. An example of the application of AIPs (expressed in the AUML notation [11]) can be found in Fig. 3.

3. AGENT ORIENTED TOOLKITS

An important enabling factor for the development of intelligent agents is constituted by the existence of a number of agent-oriented toolkits[3] that natively provides basic services such as communication, life cycle management, yellow pages and so on. In this paper we will focus on one of the best known: JADE (Java Agent DEvelopment framework) [1]. JADE is a software framework that simplifies the implementation of multi-agent systems through a middleware that complies with the FIPA specifications[3], a library of classes that developers can use or extend while creating agents and a set of graphical tools that support the debugging and deployment phases. JADE agents communicate by exchanging messages in compliance with the FIPA ACL language[3]. Furthermore, JADE supports the AMS (Agent Message Service)and the DF (Directory Facilitator), which represent the white and yellow page for agent (service) discovery.

3.1 Handling message content

A content expression within an ACL message is typically encoded as a string. However, considering software agents

[1] http://www.fipa.org
[2] http://www.fipa.org/repository/ips.html
[3] http://www.agentlink.org/resources/agent-software.html

written in an object oriented programming language, this representation is not convenient to manipulate the information conveyed by the content expression within the agent code. A better representation is by means of objects. This is to say that for instance the expression `(Person :name "Maxima" :age 30)` is easy to manipulate if represented as an object of a class, as illustrated in Fig. 1.

```
class Person {
  private String name;
  private int age;
  public void setName(String n) {name = n;}
  public String getName() {return name;}
  public void setAge(int a) {age = a;}
  public int getAge() {return age;}
}
```

Figure 1: JAVA class of concept Person, showing members and `set` and `get` methods

Proper conversion and validation operations must be carried out each time a content expression represented as an object has to be inserted into or extracted from an ACL message. The JADE toolkit, as discussed in Sec. 3, provides support for performing these conversion and validation operations automatically. thus allowing developers manipulating information within their agents as objects without the need of any extra work. More in details, a `Codec` object deals with the syntax of a given content language and an `Ontology` object checks that content expressions are meaningful to the ontology of the addressed domain.

3.2 Content Languages

While an ontology is typically specific to a given domain, content languages are domain independent. Therefore, unlike ontologies that normally must be defined ad hoc for the domain addressed by an agent application, a content language is typically selected among those already available without the need of defining a new one. JADE provides codecs for the next two content languages. The *FIPA SL language*[3] is a human-readable string-encoded (i.e. a content expression in FIPA SL is a string) content language and is probably besides KIF the mostly diffused content language in the scientific community dealing with intelligent agents, see for example the Agentcities project[4]. FIPA SL is particularly indicated in *open* applications where agents from different developers and running in different environments have to communicate. Moreover, the property of being human-readable can be very helpful when debugging and testing an application. FIPA SL deals with agent actions particularly: all agent actions in FIPA SL must be inserted into the `ACTION` construct that associates the agent action to the identifier of the agent that is intended to perform the action. For that reason we use FIPA SL as content language in this paper.

The *LEAP language*[5] is a non-human-readable byte-encoded (i.e. a content expression in LEAP is a sequence of bytes) content language that has been defined ad hoc for JADE. It is therefore clear that only JADE agents will be able to *speak* the LEAP language. There are some cases however in which the LEAP language is preferable with respect to FIPA SL. Finally, the `LEAPCodec` class is lighter than the `SLCodec` class. When there are strong memory limitations the LEAP language is preferable. LEAP also supports sequences of bytes. Finally, while the LEAP language was defined with the Content Reference Model in mind, which we will discuss in the next section.

[4]http://www.agentcities.org
[5]http://leap.crm-paris.com

4. MESSAGE CONTENT ONTOLOGIES

A message content ontology helps agents to describe facts, beliefs, hypotheses and predications about a domain. Ontologies range in abstraction to very general terms to terms that are restricted to specific domain of knowledge. Terms at very general levels of described are called *upper ontology.* Chandrasekaran et al., has discussed that there are agreements between proposed upper ontologies. There are *objects* in the world; these objects have *properties* that can take *values*; these object may exists in various *relations* with each other; properties may change over *time*, there are *events* that occur at different *time instances*; there are *processes* in *states*; events may *cause* other events as *effects*; and object have *parts* [2]. Further, there is the notion of *classes*, *instances* and *subclasses.* Other views on upperontologies, such as Generalized Upper Model, Gensim, WordNet and CYC are discussed in [5].

Here we propose a preliminary *upper message content ontology* based on the *content reference model (CRM)* used in JADE. The CRM is derived from the semantics of FIPA ACL that requires content expressions inserted into ACL messages to have proper characteristics according to the message performative (INFORM, REQUEST). In the CRM, the following elements are distinguished.

- *Concepts* expressing entities that "exist" in the world. For example

  ```
  (Person :name "Maxima" :address
  (Address :street "Dam 1" :place "Amsterdam"))
  ```

- *Predicates* expressing the status of a part of the world. These can be true or false. For example

  ```
  (Married (Person :name "Maxima")
  (Person :name "Alexander"))
  ```

- *Agent actions* expressing the actions an agent can be requested to perform. For example:

  ```
  (Marry (Person :name "Maxima")
  (Person :name "Alexander"))
  ```

- *Primitives* are atomic expressions such as Strings and Integers.
- *Aggregates*, expressing entities that are groups of other entities. For example,

  ```
  (sequence (Person :name "Maxima")
  (Person :name "Alexander"))
  ```

- *Identifying Referential Expressions (IRE)* are expressions identifying entities for which a given predicate is true. These are typically used in queries (i.e. as the content of a QUERY_REF message). For example

  ```
  (All ?x (Present-at ?x (Marriage :date "2002/2/2")))
  ```

- *Variables* expressing a generic element not known a priori and typically used within IRE.

Fig. 2 shows the relations among these elements taking into account that only predicates, agent actions and IREs are meaningful content of some ACL message. In fact a predicate can be the content of an INFORM message, and agent action can be the content of a REQUEST message and an IRE can be the content of a QUERY_REF message. Only terms can be meaningful values for the slots in a concept.

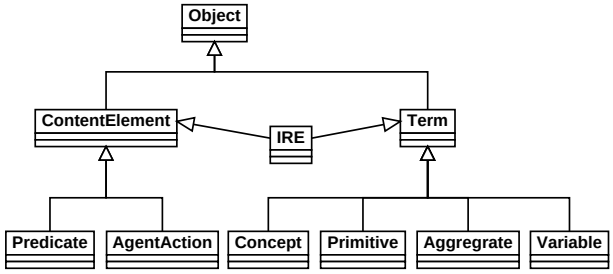

Figure 2: Content Reference Model, showing the relations between possible elements of ACL messages, according to the semantics of FIPA ACL.

4.1 Construction of Ontologies

The Protégé 2000[6] is a tool with which a user can construct ontologies. This ontology is stored in a frame-based knowledge model [9]. This model consists of classes, slots, facets and axioms. Classes are concepts in the domain of discourse, with which a taxonomic hierarchy can de constructed. Slots describe properties or attributes of these classes. A slot in itself is a frame that has a type. This can be a primitive class, like String, Integer and Float, or an instance of another class. Furthermore, a slot has a value. Facets describe properties of slots. These properties (or constraints) include:

- *Cardinality* of a slot, i.e. how many values the slot can have, i.e. 0,1,N. For example, a class *person* can only have one father (i.e. $cardinality = 1$). Or a class *father* can have multiple children (i.e. $cardinality = n$).
- *Allowed values* restriction of the value type of a slot. For example Integer, String, Instance of a class.
- *Numeric boundaries* i.e. the minimum and maximum value for a numeric slot. For example between the slot *age* is between 0 and 150.
- *Required or optional.* For example, the slot *name* is required for the class *person.*

Defining a message content ontology means defining the types of concepts, predicates and agent actions that are relevant to the addressed domain. Within Protégé a project can be included that has already defined these classes[7]. An example is shown in Fig. 4. For creating ontologies in Protégé we refer to [10]. The use of the upper agent message content ontology based on CRM is illustrated in the ACLMessage 2 including an `IRE` and a `concept` (note that both `IRE` and `concepts` are terms).

[6]http://protege.stanford.edu

[7]http://www.swi.psy.uva.nl/usr/aart/beangenerator

```
<fipa-message act="QUERY_REF" >
   <sender>
      <agent-identifier>
         <name id="Initiator@host1:8888/JADE" />
      </agent-identifier>
   </sender>
   <receiver>
      <agent-identifier>
         <name id="Responder@host2:8888/JADE" />
      </agent-identifier>
   </receiver>
   <content>
        (iota ?x (father-of ?x
                (set (Person :name "Alexander")
                     (Person :name "John"))))
   </content>
   <language>FIPA SL</language>
   <ontology>http://www.royalfamily.nl/WeddingOntology
   </ontology>
   <conversation-id>id7642974365275</conversation-id>
</fipa-message>
```

ACLMessage 2: QUERY_REF showing the use of the FIPA SL operator `iota`, which is used by $agent_{Initiator}$ to query whether the predicate on family affairs is true.

In this example, the content is an IRE of type `iota`. This IRE includes a variable whose name is x; A Predicate of type `father-of` that has two slots: (1) the value of the first slot is again x; (2) the value of the second slot is an *aggregate* of type `set` including two elements: (a) The first element is a *concept* of type `Person`; (b) The second element is another *concept* of type `Person`. It should be noticed that `iota` is an operator of the FIPA SL language, while `father-of` and Person are a predicate and a concept defined in the `WeddingOntology`.

4.2 Mapping from Content Ontologies to Java Beans

To support the agent engineer in creating and using ontologies, we developed a plug-in for the Protégé 2000 environment called the *BeanGenerator*[8]. With this plug-in a domain ontology within Protégé can be developed and exported to Java classes. In particular Java beans. A Java Bean is a special type of a Java class, which adheres to a specific design. A Java Bean has members (i.e. attributes) that can be written with a `set` operation and be read with a `get` operation or an `is` operation.

The translation from Protégé knowledge base to Java Beans works as follows: Every class in the CRM, i.e. `Concept`, `Predicate`, and `AgentAction` is the basis for the generation of a Java class. The taxonomic structure (i.e. inheritance relations) of the domain model is mapped on the inheritance capabilities of Java. Therefore if `S1` is a super-schema of `S2` then the class `C2` associated to schema `S2` must `extend` the class C1. Slots of a Class are associated with data members of the Java Bean associated with the Class. If the type of the slot is a primitive class, like String, Integer or Float, then the BeanGenerator maps them onto their Java equivalents, otherwise the member of the class is an instance of the corresponding Java class generated. If the cardinality is higher than one, a `Collection` is used.

5. INTERNATIONAL INSURANCE TRAFFIC

The European Commission has recently enacted the so-called 4th guideline: Fourth Motor Insurance Directive (Directive 2000/26/EC), operational from February 2003, that obliges all EU insurance companies to execute and settle insurance claim submissions within 3 months after the date of the incident. Today, for incidents that exceed international borders, such procedures typically take more than six months. Insurance companies have started to consult each other for mechanisms that would enable them to smooth this procedure. However, different kinds of solutions have met serious objections as they implied making their internal data available to competing companies. A Dutch insurance company, Interpolis N.V. has therefore asked Acklin B.V. to develop a prototype of a multi-agent system to tackle this problem [13]. This approach turned out to be so appealing that Interpolis asked Acklin to turn the prototype into an operational application, in cooperation with Interpolis in Holland, KBC in Belgium and R+V in Germany.

In the KIR[8] application, each insurance company is represented by two agents: $\text{agent}_{handler}$ and agent_{payer}. The $\text{agent}_{handler}$ handles a claim and initiates the process. It contacts the $\text{agent}_{payer}$of the insurance company of the person that caused the accident. Both agents use transducers to interact with back offices (and databases). Imagine the following scenario: Suppose a Dutch driver causes a car accident in Germany. The accident is reported to the German insurance company R+V in Wiesbaden. R+V is hence acting as handling bureau for this incident. R+V will open a file locally and then contact its Dutch partner Interpolis to find out where and how the Dutch counter-party is insured in Holland. If the Dutch driver is insured in Holland, Interpolis is going to act as paying office. With two reasonably simple agent types, each instantiated once for each insurance company that participates in this network, the KIR application is able to handle all standard cases swiftly, as well as to monitor all complex cases to avoid that time gets lost because they lay neglected on someone's office. Only the transducer needs to be adapted for every insurance company in the network. With this component, the agent can get and store information in the proprietary environment of the insurance company.

The process ascribed above is universal for the companies. All 17 insurance companies reason with the same concepts, i.e. insured object, car, owner, driver and policyholder. However, the ways these concepts (data) are stored in the respective databases differ per company. The reason for this can range from historical reasons to differences in sociopolitical and technical climate.

5.1 Agent collaboration

We mapped the green card traffic process on an agent collaboration diagram from AUML introduced by [11]. The Green Card Transactions are illustrated in Fig. 3 showing the handling and paying role and their pattern of interactions.

Two packages show the two main processes: *client identification* using green card number and license plate and *case identification*, using policy number and local claim number. A package shows the applied agent interaction protocols for enabling the cooperation between the agents, where an AIP describes communication patterns as an allowed sequence of message between agents and the constraints of the content of those messages. Here existing AIPs are placed in sequence to enable the process. The used AIPs are all based on the FIPA REQUEST-protocol[3]. The next communicative acts make up the process between $\text{agent}_{handler}$ and agent_{payer} and replaces, the communication by hand into communication by agent. The idea is that the manager of the department delegates the two identification tasks to the $\text{agent}_{handler}$ instead of a claim handler.

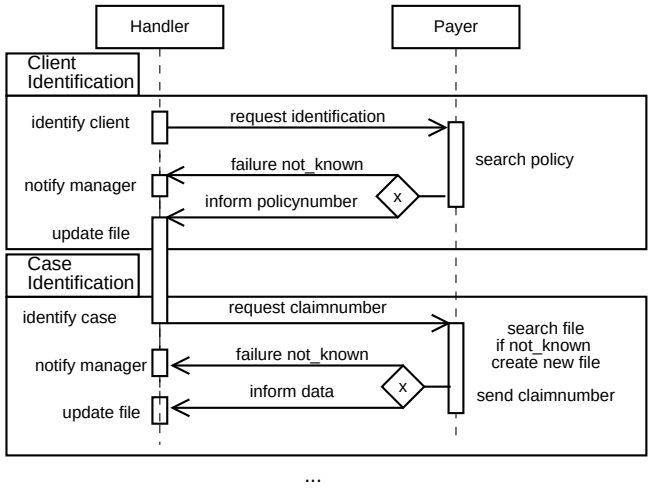

Figure 3: Part of the Green Card Traffic operations design in AUML collaboration diagram showing patterns of interaction with the operation. The dotted lines represent life-lines of agents. The arrows show interactions and the packages show the applied agent interaction protocols.

Client Identification starts with a **request** from $\text{agent}_{handler}$ and agent_{payer} for identification of the client. The identification contains a license plate and green card number. The agent_{payer} validates the identification and can response with:

- **Failure** not_known, which means that the client is not known. In many cases this is caused by typical errors such as typos in the license plate or policy number as fed by the paper reports;
- **Inform** policy number, meaning that the payer has identified the local insurance taker.

Case Identification starts with $\text{agent}_{handler}$ sending a **request** for identification of the claim to agent_{payer}. With this the $\text{agent}_{handler}$ asks for all known data from the file of the agent_{payer}. agent_{payer} with:

- **Failure** not_known, which means that the claim number is not known.

[8]KIR stands for KBC - Interpolis - R+V.

- **Inform** policy number, which means that the payer has either created a new file with the data and locally know data, or has already created a file for this case.

The latter can happen when the insured party has already registered this accident, before the handler asks for it. In both cases, the payer will send a claim number, which is the key to the file of the accident.

5.2 Ontology Design

The design of the ontology for KIR, called `GreenCardOntology`, is based on the content reference model. Conform the CRM, we first define the AgentActions. These are:

- *IdentifyParty*, which is used to identify a party.
- *IdentifyCase*, which is used to check whether a case is known at the partner.
- *UpdateCase*, which is used to update data from a particular case (this is not illustrated in the collaboration diagram).

The process is transaction based between users. It is therefore logical to first start from top-down by defining the agent actions. The agent actions use both concepts and relations. We then define the following primitive concepts, including aggregate relations:

- *Accident*, i.e. containing elements like location.
- *Party*, i.e. involved person and car.
- *Car*, i.e. insured object.
- *Owner*, i.e. the owner of the car, which does not necessarily has to be the chauffeur of the car.
- *Driver*, i.e. the one that drove in the car when it was involved in the accident.
- *Address*, i.e. a physical location indicated by street, number and zip code.

Besides that, we define the following relations: *OwnedBy*, which puts the relation between Car and Owner; and *DrivenBy*, which lays the relation between Car and Chauffeur. The resulting ontology is (partially) illustrated in Fig. 4

5.3 Agent Design

The KIR system uses mailbox semantics (cf. [7]) with per-agent serial message processing. The agents in the KIR system are derived from a common `Agent` class. The `Agent` class contains its queue and contains a `QueueMonitor` object that runs in a thread. The `QueueMonitor` thread monitors the queue and waits for a new message for a limited time. This way, the agent can also react to status changes made by the agent platform. If there are no status changes, the `QueueMonitor` object loops. If there is a message, the message is handled and after that, the `QueueMonitor` object resumes waiting for messages.

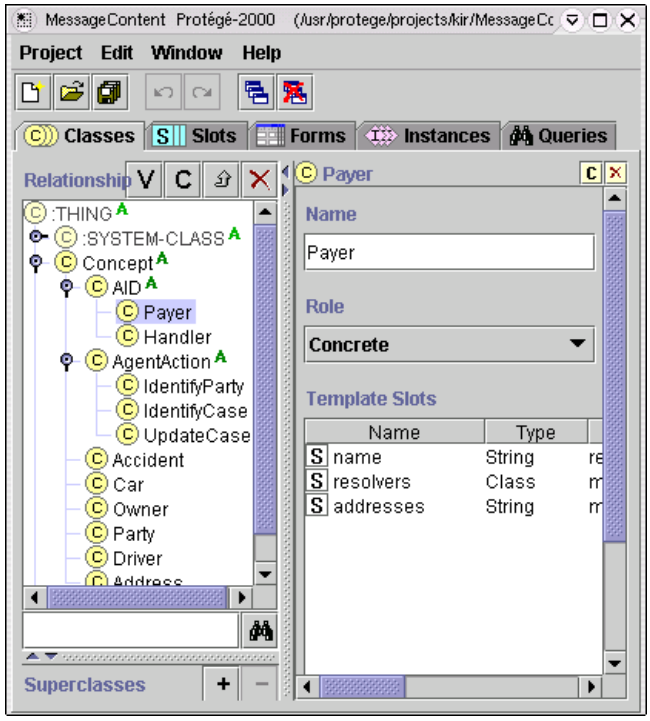

Figure 4: GreenCardOntology in Protégé, based on CRM showing AgentTypes (AID), AgentActions and Concepts.

The KIR system uses a `DispatchManager` to receive messages destined for other agents and dispatches them to the intended recipient by pushing the message object in the queue of the recipient agent. By derivation from the common `Agent` object a number of agents are defined that can perform tasks such as writing and reading databases, retrieving en sending e-mail between parties, and processing messages.

The main agent is triggered either by messages containing a flattened representation of the ontology in use, either coming from other partners or from the back office of the respective partner. In both cases, an internal representation is built from the flattened representation. Based on the values in a number of slots, the main agent decides what action is necessary. In order to be able to convert from and to a flattened representation that can be sent over-the-wire, be it through e-mail or through the transducer, two codecs have been developed. Using ANTLR[9], we defined the necessary grammars to generate the decoding parts of the codecs. The encoders where handwritten.

5.4 Scenario

Here we briefly illustrate a scenario with the use of two messages sent between the $\text{agent}_{handler}$ and agent_{payer}. The first message is where $\text{agent}_{handler}$ sends an REQUEST to agent_{payer} in the communicative act *Client identification*. The ACL representation of this message is given in ACLMessage 3. The second message is the response of agent_{payer} as given in ACLMessage 4.

[9] http://www.antl.org

```
<fipa-message act="REQUEST" >
   <sender>
      <agent-identifier>
         <name id="handler@interpolis.nl" />
      </agent-identifier>
   </sender>
   <receiver>
      <agent-identifier>
         <name id="payer@kbc.be" />
      </agent-identifier>
   </receiver>
   <content>((action
                (agent-identifier :name "payer@kbc.be")
                (IdentifyParty
                 :licenseplate  "AA-10"
                 :policynumber "7890489"))))
   </content>
   <language>FIPA SL</language>
   <ontology>http://www.interpolis.nl/GreenCardOntology
   </ontology>
   <protocol>FIPA-REQUEST</protocol>
   <conversation-id>Req1008770622742</conversation-id>
</fipa-message>
```

ACLMessage 3: REQUEST sent from $\text{agent}_{handler}$ to agent_{payer} containing the action to Identify a party.

```
<fipa-message act="INFORM" >
   <sender>
      <agent-identifier>
         <name id="payer@kbc.be"/>
      </agent-identifier>
   </sender>
   <receiver>
      <agent-identifier>
         <name id="handler@interpolis.nl" />
      </agent-identifier>
   </receiver>
   <content>
       ((result
           ((action
              (agent-identifier :name payer@kbc.be)
              (IdentifyParty
                  :licenseplate  "AA-10"
                  :policynumber "7890489")))
              (Party
                 :name "Maxima"
                 :address "Dam 1"
                 :place "Amsterdam"
            ))))
   </content>
   <language>FIPA SL</language>
   <ontology>http://www.interpolis.nl/GreenCardOntology
   </ontology>
   <protocol>FIPA-REQUEST</protocol>
   <conversation-id>Req1008770622742</conversation-id>
```

ACLMessage 4: INFORM sent from agent_{payer} to $\text{agent}_{handler}$ containing the result on the action to Identify a party.

5.5 Implementation

The final implementation of the KIR system did not use Jade, mainly because a large number of features in Jade were unnecessary. However, a number of principles used in Jade are actually also used. The IT-department of Interpolis developed the transducer between the database of Interpolis and the agent within 30 days. The agent are built in Java in less than 60 days. The models/tiers are implemented as Java-thread objects with asynchronous mail box semantics meaning that every model is a separate computational process with own control that has a mailbox with which objects communicate [7]. The mailboxes between the agents in the KIR system are regular mailboxes from a mailer, for example Sendmail or Exchange. The transport protocols used are POP3 and SMTP. The transducer mailbox is implemented as a database in which records represent messages. The mailbox also serves as the storage for the model state. Forwarding a message from one model to another means that the sending model adds a record to the mailbox database of the receiving model. This ensures that when the agent or a model (i.e. Java-thread) goes down, the state can be restored. Every action of a model is logged in a log file, for both maintenance reasons and tracking and tracing of flows within the agent. The use of databases ensures robustness and the ability to resume after a shut down. By using a fairly simplistic agent model the KIR system has become an industry strength solution. During production the KIR system frequently reaches uptime of several weeks at end, and generally is not the cause for restarts.

Because the KIR system must be implemented in a number of different organizations, each with a different technology set and in different stages of automation, the transport between agent systems needed to be a mechanism that is available in a large number of cases. Therefore, the choice was made to use e-mail and functional e-mail addresses for inter-company agent communications and for a database for communication between the agent system and the transducer. Another option for the transducers was the use of middleware, such as CORBA, but the state of the technology at several insurance companies prevented this. After testing the KIR Agent, it was taken in production. This immediately resulted in a work pressure release of three people and reduces the process of identification of client and claim from 6 months to 2 minutes. The KIR Agent is built once and re-used for each insurance company, only the competence model and transducer had to be adapted.

6. DISCUSSION

The goal of this paper was to get insights in the application of message content ontologies in the design of agents systems. A preliminary upper ontology based on a content reference model was presented that provides the semantics or message content expressions. A lot of work has been done on representing and manipulating ontologies with different types of languages. interesting experiments are to develop systems that work with these ontologies, such as translating incoming message to actions, instructions, knowledge, information and data. This work presented how processes, i.e. how software-agents use message content ontologies for these kind of ontology operations. It showed, to some extent, that:

- The use of ontologies in message exchange communication gives meaning to the contents of messages sent between agents. However to make agents collaborate a lot of standardization work has to be done.
- Message content ontologies designed with the Protégé tool can be operationalized and exported to Java source code.
- FIPA SL can be used as content language. However, the other described content language could be more useful in certain settings. For example, languages based on XML and RDF, could be more easily integrated in the work of the semantic web community.

What we did not investigate are, and what is still open for further research:

- Agent messaging comes with a combination of layers, including the use of ontologies in combination with interaction protocols. For example, agents that use the CONTRACT-NET protocol needs an ontology that provides the proper semantics for the concept *proposal*, while the REQUEST protocol needs an ontology that deals with the notion of *action*.
- More elaborated upper message content ontology suited for the process the agents play a role. This work presented a case where with an information driven character (cf. [6]), with notions of information processing actions. More interesting are processes with a knowledge driven character. For this message, content ontologies should be studied that can handling concepts like competences, knowledge, methods and signatures. A starting point can be UPML [3].
- Comparison with existing agent message content ontologies and approaches, see for example [14].
- Guidelines for what elements of an ontology to use in what context. Research on agent roles and agent organizations implicitly leads to the application of ontologies in the design and management of agent based systems, handling concepts from organization design. For example, the notions of *organizational position*, *delegation*, *authority*, and *coordination* can be used when designing an agent based system as organizing a company.

In the case study, we described how a service of an insurance company is made available to other insurance companies. Most of these types of service are not meant to be on the web, they form a part of a larger (centrally controlled) back office system. Also, they do not address issues like communication, session management, multi user support and using web standards. Another issue is that the interface to such an service is not always specified or clear. Sometimes the knowledge of how the systems works is not longer present or consistent. Furthermore, insurance companies do not want their service open for everyone, especially competing insurance companies. Therefore, software agents can play an important role as mediators between user (e.g. other software agents) and the actual service. This instead of an API (as in SOAP) of such a service that can be distributed to anyone how wants to use it.

Although the agents get the ontology translation hard coded, it is technically possible for agent to manipulate existing ontologies and even to learn new ones. This can be done on the symbol level i.e. one agent telling the other agent what elements of a ontology are and what the relations is to other elements. A more low-level approach is that the agent sends some serialized code, then load in order to be able to actually use the learned ontology. This requires more research however.

7. REFERENCES

[1] F. Bellifemine, A. Poggi, and G. Rimassa. Developing multi agent systems with a FIPA-compliant agent framework. *Software - Practice And Experience*, 31:103–128, 2001.

[2] B. Chandrasekaran, J. Josephson, and V. Benjamins. Ontology of Tasks and Methods. *IEEE Intelligent Systems*, 14(1):20–26, 1999.

[3] D. Fensel, R. Benjamins, E. Motta, and B. Wielinga. UPML: A framework for knowledge system reuse. In *Proceedings of IJCAI-99, Stockholm, Sweden*, 1999.

[4] J. Ferber. *Multi-Agent Systems.* Addison-Wesley, Reading, MA, 1999.

[5] N. Fridman and C. Hafner. The state of the art in ontology design. *AI Magazine*, 18(3):53–74, 1997.

[6] J. Galbraith. *Designing complex Organizations.* Addison-Wesley, 1973.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison Wesley Reading, MA, 1995.

[8] M. Genesereth and S. Ketchpel. Software agents. *Communications of the ACM*, 37(7):48–53, 1994.

[9] N. Noy, R. Fergerson, and M. Musen. The knowledge model of Protege-2000: Combining interoperability and flexibility. In *2th International Conference on Knowledge Engineering and Knowledge Management (EKAW'2000), Juan-les-Pins, France*, 2000.

[10] N. Noy and D. L. McGuinness. Ontology Development 101: A Guide to Creating Your First Ontology. 2001. Technical report, Stanford Medical Informatics, 2001.

[11] J. Odell, H. V. Dyke, and B. Bauer. Extending UML for agents. In G. Wagner, Lesperance Y., and E. Yu, editors, *Proc. Of the Agent-Oriented Information Systems Workshop at the 17th National Conference on Artificial Intelligence*, 2000.

[12] J. Searle. *Seech Acts.* Cambridge University Press, 1969.

[13] C. van Aart, K. Van Marcke, R. Pels, and J. Smulders. International Insurance Traffic with Software Agents. In F. van Harmelen, editor, *Proceedings of the 15th European Conference on Artificial Intelligence.* IOS Press, Amsterdam, 2002.

[14] S. Willmott, I. Constantinescu, and M. Calisti. Multilingual Agents: Ontologies, Languages and Abstractions. In *OAS2001 Workshop*, 2001.