Enforcing Agent Communication Laws by means of a Reflective Framework

Antonella Di Stefano
Corrado Santoro
Dipart. di Ing. Informatica e Telecomunicazioni
Università di Catania
Viale A. Doria, 6 - 95125 - Catania, Italy
{adistefa,csanto}@diit.unict.it

Giuseppe Pappalardo
Emiliano Tramontana
Dipartimento di Matematica e Informatica
Università di Catania
Viale A. Doria, 6 - 95125 - Catania, Italy
{pappalardo,tramontana}@dmi.unict.it

ABSTRACT

Agent Coordination Contexts (ACCs) have been proposed as virtual environments where agents live and interact. In this way, as in a human society, interactions may be subject to conventions and laws depending on their context. This is obtained by a suitable ACC that embeds the communication laws relevant to a specific application and checks whether they are fulfilled as interactions take place.

Context modeling, while representing a communication aspect relevant for all the agents of an application, is a crosscutting concern with respect to the design of the activities of each agent. In this paper, we propose an approach allowing a separate design and implementation of, respectively, behaviour and the interaction aspects constituting the context. Once the latter have been formalised in a specification consisting of communication laws, a tool generates the necessary management and checking code from the specification. Moreover, we automate the way laws are enforced on agent communication by suitably re-directing any interaction between agents, so as to ensure it respects the constraints specified by the laws, and take the actions some laws may request, before it actually takes place. Re-direction is accomplished by means of computational reflection, which transparently changes the meaning of the communication primitives normally used by agents programmers.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering

General Terms

Design

Keywords

Agents, Communication, Software Engineering, Computational Reflection

1. INTRODUCTION

SAC'04 March 14-17, 2004, Nicosia, Cyprus

In the design and development of multi-agent systems, communication and coordination among agents represent key issues [23, 8, 7, 3, 24]. As a consequence, *agent communication languages* (ACLs for short) have been the subject of intense research and debate in recent years [25, 19, 15, 18, 17, 2]. ACLs specify a common syntax and semantics for messages exchanged, in order to allow a full comprehension of messages uttered by agents in terms of sender intentions/desires, information exchanged, symbols used, etc.

In this area, the current trend is to study human societies (and in particular interaction models), trying to port the concepts and models thus identified to the world of intelligent agents [21, 4, 6]. To this aim, *Agent Coordination Contexts (ACCs)* [11, 12, 22, 9] have been proposed as virtual environments where agents belonging to the same multi-agent application live and interact. In an ACC, as in a human society, interactions are subject to conventions and laws depending on the context where they take place. Therefore, an ACC is meant to embed the communication laws relevant to a specific multi-agent application, and constrain each interaction to occur in accordance with those laws.

In designing a multi-agent application, the context, and in particular communication rules, is a concern that can be considered largely independent of the behaviour of the individual agents involved. Modelled contexts or significant parts thereof, if suitably general, could be common to several multi-agent applications (which provides opportunities for reuse). On the other hand, the same agent (model) could be useful within different contexts. The distinction between agent behaviour and context is particularly useful in "open" multi-agent systems, where a participant agent could dynamically join the system after startup: in this case, the developer of the agent does not know in advance which context the agent will find itself in.

These ideas are the basis of the software engineering methodology for multi-agent systems proposed in [29, 28]: since the *agent model* and the *behaviour of coordination media* (i.e. context modeling) are *separate concepts* (see Figure 1), they can be dealt with by different programmers and at different times. One could even envisage the co-existence of two distinct roles within a multi-agent application design effort: an "agent designer/programmer", who has the task of implementing agent behaviour, and an "ACC designer/programmer", who specifies and implements the communication rules.

In summary, context modeling is essentially a *crosscutting concern* with respect to the design of individual agents' activities, and pertains to communication aspects involving the agents of an ap-

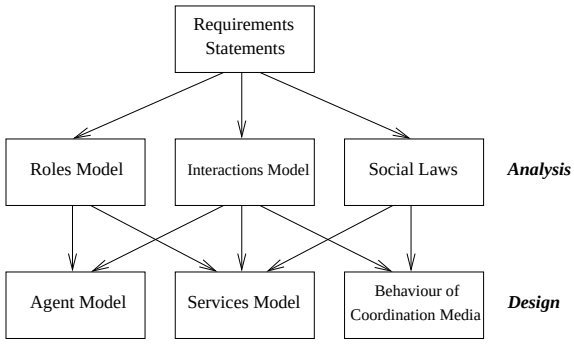

Figure 1: Agent Software Oriented Engineering

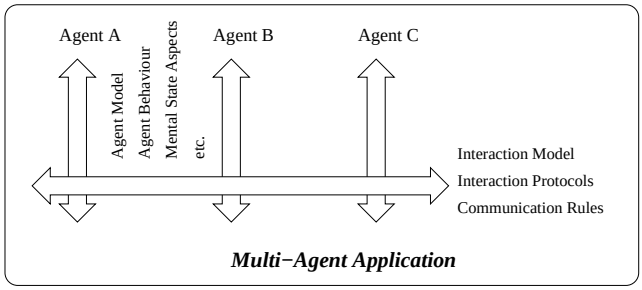

Figure 2: Concerns of a Multi-Agent Application

plication (see Figure 2). An agent developer should not need to understand, specify, or even be aware of, rules constraining inter-agent communication.

In this light, we could imagine that existing agent platforms should provide suitable mechanisms, and possibly tools, to allow agent and context to be separately implemented. In fact, widely known agent programming platforms (such as [1, 16]) do not provide any support for contexts; and even the few agent frameworks that do so [12, 13], by means of additional libraries, end up forcing agent programmers to embed context rules within agents, rather than actually allowing a separate design. The reader can appreciate the advantages of the approach described in section 4 by comparing it to the current development practices illustrated in section 3.

Although, as argued above, developing agents and communication rules should in general be independent activities, it is nevertheless unavoidable that in some cases rules may have an impact on agents. E.g., it might be necessary for agents to be aware of communication rules, so as to decide when to enter/exit a context. In any case, the proposed approach is flexible enough that the degree of context awareness appropriate for agents can be easily chosen by designers.

To this aim, we propose an infrastructure allowing a separate design and implementation of *behaviour* and *interaction* aspects for a multi-agent application. This is accomplished by using *reflective* techniques that glue these aspects together at runtime. In this work, we concentrate on the interaction aspect, proposing a solution that helps automating the relevant development process.

Once communication laws have been expressed by a set of specifications (in particular we use the context and rule model in [12, 13]), no further information is needed to translate such a set into executable code. Thus, as a first step, we propose a translation software system that takes as input a set of specifications for communication laws and autonomously generates an object-oriented library, in the appropriate programming language. This library is used to support agent communication so as to make them interact under given communication constraints.

As a second step, in order to facilitate the work of agent programmers, we propose to automate the way communication laws are enforced on agent interactions. For this purpose, our approach is able to detect, and interfere with, any interaction between agents so as to enforce the desired constraints and rules at run-time, before interactions take place. This is achieved by means of *computational reflection* [20], whereby an invocation of an operation provided by an agent can be intercepted, by an appropriate software layer, before it actually arrives to the destination agent. In this way, the semantics of communication primitives available in the agent platform can be effectively changed. The enhancement of this approach is represented by the assurance of the automatic satisfaction of communication rules by all agents.

The automatic enforcement of communication laws provides two advantages: (a) the agent programmer is not asked to take such laws into account in the code s/he writes; (b) agents cannot escape the rules enforcement, whatever their programmer's intentions.

This paper is structured as follows. Section 2 introduces the notion of context for communicating agents. Section 3 sketches the current development process for a multi-agent system. Section 4 describes the reflective architecture proposed as a solution for providing contexts to agents. Finally, conclusions are drawn in Section 5.

2. AGENT COORDINATION CONTEXTS

The *Agent Coordination Context* (ACC) concept aims at modelling communication in a multi-agent application, getting inspiration from human society. For this purpose, it has been proposed [11, 12, 13]:

(a) to *extend* the ACL speech act model [19, 17] in order to take into account some useful notions suggested by human interaction, and

(b) to make agent communication take place in the presence of a *context*, prescribing relations among, and actions upon, selected characteristics of speech acts.

These objectives can be addressed as described in the following two subsections, respectively.

2.1 Extending standard ACL speech acts

A way to pursue goal (a) is to introduce additional fields into ACL messages. A multi-agent interaction instance is considered to be characterised by several aspects: the communicating agents, which may play different roles; the information exchanged, encapsulated in a logical message; and the channel, i.e. the communication medium through which information is exchanged. On this basis, the standard ACL message format is modified by introducing the following information:

- *Sender role*, in addition to sender identity.
- The *possibility to address receivers by name and/or by role*. This is achieved by specifying a predicate $\rho_m(\cdot)$, which, taking a generic agent as parameter, is true if the message is for that agent.
- The *channel characteristics* which mainly model the temporal relationship between the act of message delivery (to a receiver) and the presence of the receiver (in the application

context)[1]. We say that the channel provides *on-time* interaction if, at delivery time, the receiver is inside that context (for example, each verbal communication, in a space where people are able to hear, features this characteristic). On the contrary, channels providing *deferred* interactions are those where the receiver is not inside the context when the message is ready to be delivered, but can be reached at a later time (mailboxes, written notes, posters, showcases are kinds of deferred channels). Temporal relationship is modeled by using an additional ACL message field, called *delivery mode* t_m, which can take the values *ontime* or *deferred*.

- A *set of agents called virtual receivers*, which are able to hear a transiting message, even if the latter is not explicitly addressed to them. Indeed, based on the observation of the human world, we notice that, unless an interaction is strictly private, communications occurring in a social context can be heard by other people even if the message is not directly addressed to them. This is a very important aspect of communication since it is a possible carrier of new unexpected interactions. To this aim, *real receivers*, which are explicitly addressed by the sender, are distinguished from *virtual receivers*, i.e. all the agents that can hear a message on a channel. The presence of virtual receivers depends on the privacy imposed by the sender on the interaction and on the constraints on delivery capability imposed by the channel. Virtual receivers are described by another predicate $v_m(\cdot)$.

With the above introduced extensions, an ACL speech act can be denoted as:

$$m ::=< n_m, < \sigma_n, \sigma_r >, \rho_m(\cdot), v_m(\cdot), t_m, \omega_m, \lambda_m, \mu_m > \quad (1)$$

Here n_m is the performative name (i.e. inform, ask, query, etc.), σ_n and σ_r respectively the name and the role of the sender agent, $\rho_m(\cdot)$ the predicate for real receivers, $v_m(\cdot)$ the predicate for virtual receivers, t_m the delivery mode, ω_m the ontology, λ_m the content language and μ_m the message content[2].

2.2 Introducing context in agent communication

Making agent communication context-aware poses both conceptual and practical issues: the precise description of contexts, and their practical realisation within a specific development framework, respectively. While these could be deeply intertwined in a naive approach to the problem, they can be clearly distinguished as *policy* and *mechanism* aspects, respectively. There are therefore well-known benefits in pursuing approaches that keep them as separate as possible. In this light, context is better described as abstractly as possible, in a formalism neutral with respect to particular implementation languages, possibly based on a suitable logic.

A context is expressed by a set of rules governing interactions (occurring in that context). Formally, given the message model described in Section 2.1, rules aim at capturing target and constraints of the communication laws holding within a social multi-agent activity. For this purpose, each rule specifies how to handle each message through suitable filtering and filling functions. Thus, a context can ensure that interactions provide receivers with acceptable and meaningful information, since each message sent can be filtered as desired, and filled with suitable default values, whenever a mandatory field has not been fully specified by the sender. E.g., a rule could specify that, if the sender omitted to choose a delivery mode for a message, this should be set to *ontime*.

To catch the said aspects a rule r is expressed by the following *pac-expression* (*precondition-assignment-constraint*) [11, 12, 13]:

$$r ::= precondition \Rightarrow assignment \mid constraint \quad (2)$$

where the *pre-condition* is a predicate on one or more message fields, which if true triggers the execution of the *assignment* or the checking of the *constraint*. The *constraint* is a predicate that specifies how a message meeting the pre-condition has to be formed, and is used to model the filtering function. The *assignment* serves to set a message field to a value if the pre-condition is met, thus modeling the filling function. Interactions in a multi-agent application are constrained by a set of rules $r_1, r_2, ..., r_n$, formed as in (2), and are allowed only if *all the rules are met*. As an example, we specify two rules expressing respectively that each message destined to agent Alice must be in Prolog (as the content language), and that each message in LISP must have a *deferred* delivery mode:

$$r_1 \stackrel{\text{def}}{=} \rho_m(Alice) \Rightarrow \lambda_m = Prolog$$
$$r_2 \stackrel{\text{def}}{=} \lambda_m = LISP \Rightarrow t_m = deferred$$

Assignment is denoted by the "left-arrow" operator, as $field \leftarrow new\_val$. For example, we can model that each message in LISP has to be delivered *ontime* (by default):

$$r_3 \stackrel{\text{def}}{=} (\lambda_m = LISP) \wedge (t_m \; is \; nil) \Rightarrow t_m \leftarrow ontime$$

From an architectural and practical point of view, since rules must be checked during message exchange, a *communication infrastructure* is needed at runtime to support communication, so as to check and enforce the rules defining the context. A proposal for this infrastructure is the aim of Section 4.

3. CURRENT DEVELOPMENT PRACTICES

To better appreciate the advantages of our automated-reflective approach, which will be described in Section 4, let us sketch how a set of communication rules, specified as discussed in Section 2 could be tackled "by hand" within a multi-agent application development

First, an "ACC programmer" would translate specified rules into a library suitable for the target agent platform. Agent programmers would now include appropriate invocations to the noted library, so that agent interaction complies with the desired laws. As an example, to discuss how this inclusion can be accomplished, let us suppose we adopt an agent platform with the following characteristics:

- the agent platform is developed in an object-oriented language (such as Java) [1, 16];
- each agent is encapsulated in an `Agent` class, provided by the platform itself;
- the `Agent` class provides two methods, e.g. `send()` and `receive()`, to exchange messages.

[1]Here, we consider only the temporal relationship since, in our opinion, it is one of the most important characteristics of channels. Indeed, the concept of channel in the human world is more complex and is also related to physical constraints posed by the environments. For example, a channel such as a phone line is needed if two interacting parts are not in the same physical place. However, not all human world constraints fit an agent environment; as a result modeling agent channels, starting from human channels, requires a more in-depth analysis which will be the aim of our future work.

[2]Other ACL message fields (such as "reply-with" or "in-reply-to") are not modeled since they are not sensitive for our analysis.

We note that two main approaches exist for the agent development process when using ACCs: (*i*) using objects that represent context and rules, offering the needed access methods, or (*ii*) implementing rule checking in a `RuleAwareAgent` abstract class (subclass of `Agent`) that embeds rule checking and is used to represent all the agents of the application.

The first approach (*i*), an example of which is reported in Figure 3a, amounts to designing and implementing a library that offers classes such as `Context` and `Rule`, which represent abstractions respectively for ACCs and the relevant communication rules (this approach is used specifically in [13], but its general principles are the basis for other ACC proposals such as [7]).

Class `Context` implements the methods to enter the ACC and communicate through it, by means of e.g. methods `join()`, `send()` and `receive()`. Class `Rule` provides a `Context` object with the ability to instantiate a new rule implementing a *pac-expression*, and is intended to process ACL messages, whose validity is checked with respect to the new rule.

Although approach (*i*) is well suited for realising ACCs, it implies that applications originally implemented without recourse to ACCs need to be rewritten in order to use the classes supporting ACCs. This could imply changing many lines of code (e.g. all the calls to method `send()` would have to be changed into `ctx.send()` as in Figure 3a, moreover a call to `Context.join()` has to be added, etc.).

```
public class MyAgent extends Agent {
  void agentBehaviour () {
    // join to the context of the application named 'e-auction'
    Context ctx = Context.join ("e-auction");
    ...
    // send a message through the context
    ...
    ctx.send (new ACLMessage (...));
    ...
    // receive a message sent through the context
    ACLMessage m = ctx.receive ();
  }
}
```

(a)

```
public class RuleAwareAgent extends Agent {
  void applyRules (ACLMessage m)
  throws RuleViolationException {
    if (m.getPerformative().equals("inform") &&
        !m.getLanguage().equals("LISP"))
           throws new RuleViolationException("rule 1");
    if (m.getPerformative().equals("ask-if")) {
      m.addReceiver(
        "another-agent@pciso.iit.unict.it:1099/JADE");
    }
  }
  void send (ACLMessage m) throws Exception {
    applyRules(m);
    super.send(m);
  }
}
```

(b)

Figure 3: Two approaches for engineering context aware agents

As to the second development approach (*ii*) noted above, it amounts to writing a class `RuleAwareAgent` (see Figure 3b) that extends `Agent` and will be used as a super-class for all agents having to communicate through an ACC with given rules. The programmer of `RuleAwareAgent` should implement a `send()` method performing rule checking, by means e.g. of a series of "if", before sending the message (see `applyRules()` in Figure 3b). In this case, the design and implementation of agent behaviour is kept relatively separated from the design of ACCs. However, if an existing non-ACC based application has to be transformed into an ACC-based one, the original source code would have to be rewritten, at least to change the name of the ancestor class of each agent class.

In both cases (*i*) and (*ii*), agent programmers must be to a certain extent aware of rules and take care of rule checking by suitably modifying agent code. Moreover, should rules change (due to new application requirements), this awareness will force agent programmers to perform a re-engineering process for agents in order to take into account the necessary changes. Thus, the interaction aspect, which is basically crosscutting, induces changes to all the agents as it was *just another* agent-specific concern (such as behaviour, mental states, etc.). This represents an evident contradiction to the view we expressed in Section 1 (see Figure 2).

4. A REFLECTIVE ARCHITECTURE TO ENFORCE COMMUNICATION LAWS

4.1 Outline of the Proposed Development Process

For the design and implementation of a multi-agent application that bases agent interaction on the concept of ACC and rules, we propose a software engineering method based on the following guidelines:

1. As discussed in the Introduction, two distinct figures of designer are envisaged, i.e. an ACC designer and an agent designer.

2. The ACC designer is entrusted with the task of establishing interaction protocols and communication rules, providing them through a suitable specification language (cf. Section 2).

3. The agent designer takes care of agent behavioural aspects and implements them by means of a Java-based agent platform [1, 16]. S/he need not be aware of the presence of an ACC and its rules, but must only be ready to handle a possible exception thrown by the underlying ACC support when a message sent violates a communication constraint. The agent designer is not forced to change any line of its code, should new application requirements cause communication rules to be added, removed or changed.[3]

4. A tool, called *ACC Builder*, targeted for the agent platform employed, generates the code needed to implement communication rules. It should be noted that the ACC Builder is only needed before the application runs. The classes implementing the ACC rules are transparently integrated into the agents' code, by changing selected bytecode parts, either at load-time or permanently (i.e. by changing the class files). This does not require source code to be manually modified

[3]For those cases where, as mentioned in the Introduction, the agent programmer actually needs to be aware of the presence of a context, a possible strategy is for the ACC support to provide information about rules violated within the exception raised in this event, and for the agent programmer to analyse this information and take the desired measures in the exception handling code. Alternatively, the agent programmer could use special messages intended to inform the ACC of specific needs the agent may have in interacting with other agents; in response to such requests, the ACC would check whether they are inconsistent with its set of rules, and if so decide whether to reject the request, or accept it relaxing the rules involved.

and is made possible by *computational reflection*, which allows all the limitations dealt with in Section 3 to be overcome.

4.2 Computational Reflection

A reflective software system consists of a part that performs some computation and another part that reasons about and influences the first part by *inspecting* it and *intercepting* its operations [20]. Usually reflective systems are represented as two-level systems, where a *baselevel* implements an application and the *metalevel* monitors and controls the application.

One of the most widespread reflective models for object-oriented systems is the *metaobject model*, whereby a metalevel class can be associated with a baselevel class, to allow instances of the metalevel class, called *metaobjects*, to intercept operations performed on objects instances of the baselevel class [20, 14].

As reported in the literature, reflective systems have been used aiming at the provision of additional functionalities, such as synchronisation [27], distribution [10], fault-tolerance [26], etc.

Among reflective language extensions allowing the implementation of these systems, we have selected for the investigation reported the Javassist library [5], which is based on Java.

4.3 The Reflective Software Architecture

The reflective software architecture that allows communication laws to be enforced in an environment where agents interact consists of a baselevel, containing agents, and a metalevel, controlling all the communications between agents.

Agents developed within current object-oriented agent platforms (e.g. Jade [1]) consist of interacting classes, each implementing parts of an agent. In order to implement an agent, a sub-class of the `Agent` class of the platform is defined. Agents collaborate with each other by sending messages.

Using the proposed architecture, agents are not forced to be aware of communication contexts; however, upon invocation of the standard send primitive, their messages are effectively regulated by the actual context. Invocation of a send, implemented at the baselevel by the agent framework, is in fact handled by the metalevel in order to enforce the rules of the actual communication context. As a benefit of the reflective approach, this is completely transparent to agents.

The metalevel that we propose consists of two classes: `CommunicationMO` that captures the messages exchanged; and `Context`, containing the context name, the (references to) agents that joined that context, the role of each agent in the context, and the rules that messages sent by agents must respect, as well as the actions that rules can generate when messages are checked.

4.3.1 CommunicationMO

Class `CommunicationMO` is developed as a metaobject class intended to be associated with class `Agent` of the baselevel agent framework. This association makes it possible to intercept all the messages that agents wish to send. Depending on the reflective language extension used it is possible to selectively trap methods. This avoids to introduce unnecessary jumps to the metalevel and performance overhead[4]. Whenever a method call is intercepted by `CommunicationMO`, it checks which one of the communication primitives (such as `send()` or `receive()`) has been invoked. Then, the appropriate context is retrieved and the rules of that context are processed to verify the message. When these rules hold, the message is actually delivered (see Figure 4).

`CommunicationMO` is designed to be a general metaobject class so as to be associated with any agent framework and act on the interaction primitives that the framework provides to agents. It detects at runtime agents communicating and stores their names and messages into the active `Context` object.

```
public class MyAgent extends Agent {

  void agentBehaviour() {
    ...
    // send a message
    try {
      send(new ACLMessage(...));
    } catch (Exception e) { ... }

    // the exception can be raised by both the agent
    // platform (baselevel) and the ACC (metalevel)
  }
}

// Baselevel class Agent will be associated with
// metalevel class CommunicationMO

public class CommunicationMO extends Metaobject {

  // trapMethodcall automatically receives control when
  // a method of the associated baselevel class is invoked
  public Object trapMethodcall(int id, Object[] args)
  throws Throwable {
    // check if a message is going to be sent
    if ( getMethodName(id).compareTo("send") ) {
      // get the agent identifier
      Object agId = getObject();
      // retrieve the agent context
      Context ctx = Context.get(agId);
      if (ctx == null) {
        // or make the agent join the defaul context
        Context ctx = Context.join(agId, "default");
      }
      // process the rules of the context for the message to be sent
      ctx.applyRules(args);   // args is the argument of the
                              // trapped method, i.e. send(...)
    }
    // control gets here when the intercepted method is not send() or
    // applyRules() does not raise an exception

    // trapMethodcall of Metaobject invokes the intercepted method
    // of the associated baselevel class, i.e. send(...) of class Agent
    return super.trapMethodcall(id, args);
  }
}
```

Figure 4: Metaobject that allows redefinition of communication primitives

4.3.2 Context

Class `Context` is a metalevel class responsible to check a message and perform some activities on it. Checks consist in analysing whether sender, receiver, structure and arguments of a message satisfy the communication constraints imposed by rules. The activities performed on a message are based on a repository of rules and include: throwing an exception, if a rule is violated; transforming it; filling it with other arguments; changing its receiver field; sending a copy to other receivers, etc.

For this purpose, class `Context` provides method `applyRules()` that, taking as parameter an object representing the message to be exchanged, checks the latter with respect to the defined rules and performs the needed actions.

4.4 Generating Context Classes

As reported in Section 4.1, in our approach the *ACC Builder* tool automatically generates metalevel classes `CommunicationMO` and

[4]The reflective language extension we have used, Javassist, allow a certain degree of configuration, so that it would be possible to trap just certain methods of a class.

`Context`. ACC Builder operates by processing a text file which specifies context characteristics and rules, and generates the source code of the metalevel classes to be glued with the multi-agent application. The context specification language employed is XML-based. A sample fragment of a specification file is shown in figure 5. A full context specification provides the following information:

```
<AgentPlatform> JADE </AgentPlatform>
<Context> Auction </Context>
    <AgentRole> Bidder </AgentRole>
    <AgentRole> Auctioneer </AgentRole>
    <AgentRole> Guest </AgentRole>
    <CommunicationRule>
        unknown(AgentRole) => AgentRole = Guest
    </CommunicationRule>
    <CommunicationRule>
        sender.AgentRole == Auctioneer and
        receiver.AgentRole == Bidder
        => message.content == OpenAuction
           or message.content == CloseAuction
    </CommunicationRule>
</Context>
```

Figure 5: Context specification file

- *Type of agent platform used to build the multi-agent application*, this is needed to generate suitable metalevel classes which have to be associated to the existing baselevel classes of the platform.
- *Context name*.
- *Identifiers of participating agents and their role in the context*[5].
- *Predicates definition for receivers*, expressed as first-order logic expressions.
- *Communication rules*, expressed as first-order logic expressions.

Once metalevel classes have been generated by the ACC Builder, the association between `CommunicationMO` and `Agent` is performed by using Javassist. At load time, when the class `Agent` is loaded by the JVM, its bytecode is modified by inserting the appropriate jumps to the metaobject `CommunicationMO`. Whenever instance objects of class `Agent` are executed, control is re-directed to the associated `CommunicationMO` metaobject.

Javassist also allows permanent modification of the bytecode of class `Agent` (by changing its class file). This alternative approach is better followed when the magnitude of use of the defined context is large.

5. CONCLUSIONS

In this paper we have proposed an approach for engineering a multi-agent application that allows separate development of communication and behavioural concerns. Communication concerns are taken care of by providing the definition of an Agent Coordination Context, which specifies a set of rules governing agent interactions within a certain multi-agent application. Since agent communication aspects are crosscutting with respect to agent behaviour, we have proposed to clearly separate them by a development approach based on a reflective architecture.

We have also proposed a tool, called ACC Builder, that automates the development process of a multi-agent application by generating the code needed to enforce at runtime communication rules, starting from a specification. In a prototype implementation developed, the ACC Builder generates the code for the JADE platform starting from a set of rules specified in a restricted Prolog-like language. Tests performed with this prototype have shown the validity of the proposed approach. Future work planned include an expansion of the rule specification language to represent all requirements that may arise in an agent communication environment.

As a further extension of the work presented, we plan to deal with scenarios requiring agents to be ultimately aware of the context they will operate in (e.g. because they should organise their work accordingly, or for security reasons, so that an agent can keep some messages private even in contexts that would broadcast them). In such cases, we envisage that a configuration phase would be performed, e.g. when agents are deployed, to insert them into a specific ACC whose rules have been made publicly available. In this way, we manage to cater for different ACCs, depending on the deployment scenarios, and yet to retain the approach advocated in this work, i.e. that no a priori knowledge of ACCs should be required for the development of agents.

[5]If the application is open, there may be agents, unknown at design time, which, at runtime, want to participate in the application. In this case, unknown agents are mapped to a special role, called *guest*.

6. 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] Bologna, Italy. *First Intl. ACM Joint Conference on Autonomous Agents and Multi-Agent Systems*, July 15-19 2002.

[3] G. Cabri, L. Leonardi, and F. Zambonelli. Mobile-Agent Coordination Models for Internet Applications. *IEEE Computer*, 33(2), February 2000.

[4] C. Castelfranchi, F. Dignum, C. Jonker, and J. Treur. Deliberate normative agents: Principles and architecture. In *Proc. of The Sixth International Workshop on Agent Theories, Architectures, and Languages (ATAL-99), Orlando, FL*, 1999.

[5] S. Chiba. Load-time Structural Reflection in Java. In *Proceedings of the ECOOP 2000*, volume 1850 of *Lecture Notes in Computer Science*, 2000.

[6] R. Conte and C. Castelfranchi. *Cognitive and Social Action*. UCL Press., 1995.

[7] M. Cremonini, A. Omicini, and F. Zambonelli. Coordination and access control in open distributed agent systems: The TuCSoN approach. In A. Porto and G.-C. Roman, editors, *Coordination Languages and Models*, volume 1906 of *LNCS*, pages 99–114. Springer-Verlag, 2000.

[8] R. De Nicola, G. L. Ferrari, and R. Pugliese. KLAIM: A Kernel Language for Agents Interaction and Mobility. *IEEE Transaction on Software Engineering*, 24 - No. 5, 1998.

[9] E. Denti, A. Omicini, and A. Ricci. Coordination tools for MAS development and deployment. *Applied Artificial Intelligence*, 16(9/10):721–752, Oct./Dec. 2002. Special Issue: Engineering Agent Systems – Best of "From Agent Theory to Agent Implementation (AT2AI-3)".

[10] A. Di Stefano, G. Pappalardo, and E. Tramontana. Introducing Distribution into Applications: a Reflective Approach for Transparency and Dynamic Fine-Grained Object Allocation. In *Proceedings of the Seventh IEEE*

Symposium on Computers and Communications (ISCC'02), Taormina, Italy, 2002.

[11] A. Di Stefano and C. Santoro. Coordinating mobile agents by means of communicators. In A. Omicini and M. Viroli, editors, *WOA 2001 – Dagli oggetti agli agenti: tendenze evolutive dei sistemi software*, Modena, Italy, Sept. 4–5 2001. Pitagora Editrice Bologna.

[12] A. Di Stefano and C. Santoro. Modeling Multi-Agent Communication Contexts. In *First Intl. ACM Joint Conference on Autonomous Agents and Multi-Agent Systems* [2].

[13] A. Di Stefano and C. Santoro. Integrating Agent Communication Contexts in JADE. *Telecom Italia Journal EXP*, Sept. 2003.

[14] J. Ferber. Computational Reflection in Class Based Object Oriented Languages. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'89)*, volume 24 of *Sigplan Notices*, pages 317–326, New York, NY, 1989.

[15] T. Finin and Y. Labour. A Proposal for a New KQML Specification. Technical Report TR-CS-97-03, Computer Science and Electrical Engineering Dept., Univ. of Maryland., 1997.

[16] FIPA-OS Home Page. *http://fipa-os.sourceforge.net*.

[17] Foundation for Intelligent Physical Agents. FIPA-ACL Specification, available at http://www.fipa.org/specs/fipa00061/.

[18] Y. Labrou, T. Finin, and J. Mayfield. KQML as an Agent Communication Language. In J. Bradshaw et al., editor, *Software Agents*. AAAI Press, Cambrigde, Mass., 1997.

[19] Y. Labrou, T. Finin, and Y. Peng. Agent Communication Languages: the Current Landscape. *IEEE Intelligent Systems*, March-April 1999.

[20] P. Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, volume 22 (12) of *Sigplan Notices*, pages 147–155, Orlando, FA, 1987.

[21] T. Malsch. Naming the Unnamable: Socionics or the Sociological Turn of/to Distributed Artificial Intelligence. *Journal of Autonomous Agents and Multi-Agent Systems*, 4(3):155–186, September 2001.

[22] A. Omicini. Towards a notion of agent coordination context. In D. C. Marinescu and C. Lee, editors, *Process Coordination and Ubiquitous Computing*, chapter 12, pages 187–200. CRC Press, Oct. 2002.

[23] G. Papadopoulos and F. Arbab. Coordination models and languages. In *Advances in Computer*, volume 46. Academic Press, 1998.

[24] A. Ricci, E. Denti, and A. Omicini. Agent coordination infrastructures for virtual enterprises and workflow management. In M. Klusch and F. Zambonelli, editors, *Cooperative Information Agents V*, volume 2182 of *LNCS*, pages 235–246. Springer-Verlag, 2001. 5th International Workshop (CIA 2001), Modena, Italy, 6–8 Sept. 2001. Proceedings.

[25] M. P. Singh. Agent Communication Languages: Rethinking the Principles. *IEEE Computer*, 31(12):40–47, December 1998.

[26] R. J. Stroud and Z. Wu. Using Metaobject Protocols to Satisfy Non-Functional Requirements. In C. Zimmermann, editor, *Advances in Object-Oriented Metalevel Architectures and Reflection*. CRC Press, 1996.

[27] E. Tramontana. Managing Evolution Using Cooperative Designs and a Reflective Architecture. In W. Cazzola, R. J. Stroud, and F. Tisato, editors, *Reflection and Software Engineering*, volume 1826 of *Lecture Notes in Computer Science*, pages 59–78. Springer-Verlag, June 2000.

[28] M. Wooldridge, N. Jennings, and D. Kinny. The Gaia Methodology for Agent-Oriented Analysis and Design. *Journal of Autonomous Agents and Multi-Agent Systems*, 3(3), 2000.

[29] F. Zambonelli, N. Jennings, A. Omicini, and M. Wooldridge. Agent-oriented software engineering for Internet applications. In A. Omicini, F. Zambonelli, M. Klusch, and R. Tolksdorf, editors, *Coordination of Internet Agents: Models, Technologies, and Applications*, chapter 13, pages 326–346. Springer-Verlag, Mar. 2001.