CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2000; **00**:1–7 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

The Transparent Implementation of Agent Communication Contexts

Antonella Di Stefano[1] , Giuseppe Pappalardo[2],
Corrado Santoro[1,*]and Emiliano Tramontana[2]

[1] *Dipartimento di Ingegneria Informatica e delle Telecomunicazioni, Università di Catania, Italy.*
[2] *Dipartimento di Matematica e Informatica, Università di Catania, Italy.*

SUMMARY

Agent Communication Contexts (ACCs) are virtual environments where agents may live and interact. In this way, as in a human society, interactions may be subject to conventions and laws depending on the context where they occur. For this to be possible, an ACC should embed the communication laws relevant to the intended class of agent applications and enforce them, as interactions among agents take place.

Although context is a communication aspect relevant for all the agents of an application, its modelling should be, in principle, an orthogonal concern with respect to the design of the activities of each agent. Consistently with this view, this work advocates the separate development of, respectively, agent behaviour, and the interaction aspects constituting the context. The latter is first abstractly specified as a set of communication laws, then automatically implemented by a tool that generates the necessary ACC management and checking code from the specification. The appropriate portions of this code should be activated whenever an interaction between agents takes place, so as to ensure that (i) the constraints specified by the laws are respected by the interaction, and (ii) the actions some of the laws require are carried out before the interaction actually occurs.

Moreover, this work proposes an infrastructure whereby ACC code is triggered at runtime, whenever agents interact with each other. No source code modification or recompilation is required for this. All is seamlessly accomplished by means of computational reflection, which transparently changes the meaning of the communication primitives normally used by agent programmers.

KEY WORDS: Multi-Agent Systems, Communication, Contexts, Software Engineering, Computational Reflection

*Correspondence to: Corrado Santoro, Dipartimento di Ing. Informatica e Telecomunicazioni,
Università di Catania, Viale A. Doria, 6 - 95125 - Catania, Italy
E-mail: csanto@diit.unict.it

Received
Revised

1. Introduction

In the design and development of multi-agent systems, *communication* and *coordination* among agents represent key issues [28, 10, 9, 2, 29].

Agent communication languages (ACLs for short) have been widely employed [30, 24, 20, 23, 16], and standardised [22], for the abstract description of communication among agents. Communication instances are encoded as ACL *messages* or *speech acts*.

In the area of agent coordination and interaction, the current, emerging, approach is to study human societies (and especially their interaction models), trying to port the identified concepts and models to the world of intelligent agents [26, 4, 8]. As part of this trend, *Agent Communication/Coordination Contexts (ACCs)* [15, 16, 27, 11, 13] have been proposed as virtual environments where agents belonging to the same multi-agent application live and interact. In an ACC, 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.

ACCs are meant to address concerns that may be considered [35, 34] largely independent of the behaviour of the individual agents involved. Thanks to this concern separation, an ACC, or significant parts thereof, if sufficiently general, could be reused in the design of various multi-agent applications. On the other hand, the same agent(s) could be employed with different ACCs. For example, an agent could operate simultaneously on behalf of different multi-agent applications and thus be effectively deployed within different ACCs; or a multi-agent application could be evolved by changing its communication rules, that is its ACC, while leaving the agents' behaviours unchanged.

At this stage, it is worth stressing that, notwithstanding the potential for combining agents and ACCs in various ways *at runtime*, in our approach we assume that a multi-agent application is built around exactly one ACC.

In any case, it stands to reason that, *at design time*, agent and context modelling may—and should—be separate, independent activities, possibly carried out 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", dealing with agent behaviour, and an "ACC designer/programmer", who specifies/implements communication rules. The former should not need to understand, specify, or even be aware of, rules constraining inter-agent communication.

In the light of the previous discussion, existing agent platforms might be expected to provide suitable mechanisms, and possibly tools, allowing agents and contexts to be separately implemented. In fact, widely known platforms (such as [1, 21]) do not provide any support for contexts; and even the few agent frameworks that do [16, 17], 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 alternative approach advocated in this paper, by comparing the sample code provided in Section 4 to the current development practices illustrated in Section 3.

Our proposal is based on an infrastructure allowing a separate design and implementation of *behaviour* and *context*, i.e. *interaction*, aspects for a multi-agent application. This is accomplished by using *reflective* techniques, which glue these aspects together at runtime. In this work, we concentrate on the context-interaction aspect, putting forward a solution that helps automating the relevant development process.

The starting point is to specify the desired ACC as a set of communication laws (in particular, we use the context and rule model of [16, 17]). No further information is needed to turn the specification into executable code. Thus, as a first development aid, we provide a software tool that translates the specification into an object-oriented library, capable of constraining and influencing agent communication as desired.

As a second measure, in order to facilitate agent programmers' work, we automate the way communication laws (encoded in the generated library) are imposed on agent interactions. For this purpose, we rely on the ability to detect, and interfere with, any interaction between agents, in such a way to enforce the desired constraints and rules at run-time, before the interaction takes place. This is achieved via *computational reflection* [25], 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. Thus, the semantics of agent platform communication primitives can be effectively changed in a fashion that ensures the automatic satisfaction of communication rules by all agents.

This paper is structured as follows. Section 2 introduces the notion of context for communicating agents. Section 3 sketches the development practices currently adopted for a multi-agent system. Section 4 describes the reflective architecture proposed as a solution for providing contexts to agents. Section 5 outlines a development tool for generating contexts. Section 6 describes a sample application using the proposed architecture. Finally, the authors' conclusions are drawn in Section 7.

2. Agent Communication Contexts

A possible approach to model communication in a multi-agent application, getting inspiration from human society, is based on the *Agent Communication Context* (ACC) concept [11, 27, 29]. For this purpose, in [15, 16, 17] Di Stefano and Santoro proposed:

(a) to *extend* the ACL speech act model [24, 22], taking into account additional 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, along the lines of [15, 16, 17], as reported in the following two sections 2.1 and 2.2, respectively.

2.1. Extending Standard ACL Speech Acts

A multi-agent interaction instance is 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.

Roles are not explicitly expressed in standardised [23, 22] ACL messages. Yet they are deemed to be very important [3]—especially in an open system—since they provide a complete knowledge of the characteristics, tasks, abilities and relationships of each involved agent. Introducing roles into an ACL message allows them to be exploited both at the sender's and receivers' sides. Including the sender's role name within a message permits its receivers a more in-depth understanding of the sender's characteristics. On the other hand, specifying a receiver's role name addresses the case (frequent in human interaction) that the message incorporating it should be sent to all agents playing that role, rather than to specific named agents. In this light, in order to gain flexibility in defining receiver agents, we introduce, as part of an ACL message, a *predicate* ρ_m satisfied by exactly those agents for which the message is intended. These shall be termed *real receivers*. The definition of ρ_m may be based on the explicit name and/or the role of its argument agent. E.g.:

$$\rho_m \quad \overset{\text{def}}{=} \quad \lambda a.\ (a{=}\mathit{Alice} \vee \mathit{role}(a){=}\mathit{Bidder})$$

Another issue related to message exchange, clearly suggested by human interaction, is the possibility for an agent to hear a transiting message, even if the latter is not explicitly addressed to it. It is important to model this phenomenon, as a possible carrier of new unexpected interactions. Thus we introduce the concept of *virtual receivers*, as the agents that can hear a message in transit on a channel, and specify them as those satisfying a predicate v_m, which is incorporated into messages. The presence and form of predicate v_m in a message depends on the constraints imposed by (*i*) the sender on the interaction privacy, and (*ii*) the channel on delivery capability (e.g. broadcasting).

Channel characteristics are essentially modeled on the basis of the temporal relationship between the act of message delivery to a receiver, and the presence of the receiver in the application context[†]. We say that a channel provides *ontime* interaction if the receiver is always inside the application context at message delivery time (for example, a space where people are able to hear is an ontime channel for verbal communication). Instead, a channel provides *deferred* interactions if, when a message m is ready to be delivered, its receiver may be outside the application context, but will be able to obtain m later, after joining the context. Mailboxes, written notes, posters, showcases are kinds of deferred channels. Temporal relationship is modelled by using an additional ACL message field, called *delivery mode* t_m, which can take the values *ontime* or *deferred*.

With the above introduced extensions, an ACL message has the form:

$$m ::= < n_m, \sigma_n, \sigma_r, \rho_m, v_m, t_m, \omega_m, \kappa_m, \mu_m > \qquad (1)$$

where n_m is the performative name (i.e. *inform*, *ask*, *query*, etc.); σ_n and σ_r the name and the role of the sender agent respectively; ρ_m the predicate for real receivers, v_m the predicate for virtual receivers, t_m the delivery mode, ω_m the ontology, κ_m the content language and μ_m the message content[‡].

2.2. Introducing Contexts in Agent Communication

Making agent communication context-aware poses both conceptual and practical issues: on one hand the precise description of contexts, on the other hand their development within a specific agent framework. While these could be deeply intertwined in a naive approach, it is better, for the sake of generality and concern separation, to describe context 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 the target and constraints

[†]Here, we consider only the temporal relationship since, in our opinion, it is one of the most important characteristics of channels.
[‡]Other ACL message fi elds (such as "reply-with" or "in-reply-to") are not modelled since they are not sensitive for our study.

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). For example, a rule could specify that, if the sender omitted the delivery mode for a message, this should be set to $ontime$.

To catch the aforementioned aspects, a rule r is denoted by a *precondition-assignment-constraint* expression [15, 16, 17], or *pac-expression*, of the form:

$$r ::= precondition \Rightarrow assignment \mid constraint \tag{2}$$

Here, *precondition* is a predicate on one or more message fields, which, if true, triggers the execution of the *assignment* or the checking of the *constraint*; *constraint* is a predicate that specifies how a message meeting the precondition has to be formed, and is used to model the filtering function; *assignment* is meant to set a message field to a value if the precondition is met, thus modelling the filling function. Interactions within 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:

$$\begin{aligned} r_1 &\overset{\text{def}}{=} \rho_m(Alice) \Rightarrow \kappa_m{=}Prolog \\ r_2 &\overset{\text{def}}{=} \kappa_m{=}LISP \Rightarrow t_m{=}deferred \end{aligned}$$

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:

$$r_3 \overset{\text{def}}{=} \kappa_m{=}LISP \wedge isnil(t_m) \Rightarrow t_m{\leftarrow}ontime$$

From a practical point of view, since rules defining a context must apply during message exchange, a *communication infrastructure* is needed at runtime, to support communication while at the same time checking and enforcing the rules. Such an infrastructure can be set up "by hand"—the current practices are discussed in Section 3—or in an automated and transparent fashion, as proposed in 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 hardwired "by hand" within the implementation of a multi-agent application.

First, an "ACC programmer" would translate the specified rules into a library suitable for the target agent platform. Agent programmers would then encode agent interaction, through appropriate invocations to the noted library, so that it automatically complies with the desired laws. As an example, let us suppose the following assumptions are satisfied:

A1 the agent platform is developed in an object-oriented language (such as Java) [1, 21];

A2 each agent is developed as a subclass of an `Agent` base class, supplied by the platform itself.

A3 the `Agent` class provides two methods, say `send()` and `receive()`, to exchange messages;

Two main approaches exist for ACC-based agent development:

(*i*) using objects that represent context and rules, offering the needed access methods, or
(*ii*) delegating rule checking to `RuleAwareAgent`, an abstract class extending `Agent`, which should be used as a superclass for all the agents of the application.

The first approach (*i*), an example of which is reported in Figure 1a, amounts to designing and implementing a library that offers class `Context`, which represents an abstraction for ACCs (this approach is used specifically in [17], but its general principles are the basis for other ACC proposals such as [9]). Class `Context` provides the methods to enter the ACC and communicate through it (called e.g. `join()`, `send()` and `receive()`). It has also the ability to instantiate a new rule implementing a *pac-expression*, and is intended to process ACL messages by checking their validity against 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. For example, calls to method `send()` should be changed into `ctx.send()` and `Context.join()` calls should be added as appropriate (cf. Figure 1a).

The second "conventional" development approach (*ii*) identified above amounts to writing a class `RuleAwareAgent` (see Figure 1b) that extends `Agent` and will be used as a superclass for all agents needing to communicate through an ACC with given rules. The programmer of `RuleAwareAgent` should implement a `send()` method performing rule checking, for example by means of a series of "if", before sending the message by `Agent`'s `send()` method (see `applyRules()` in Figure 1b). 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 suitably modify agent code. Moreover, should rules change (due to new application requirements), this awareness will force agent applications to be re-compiled in order to take into account the changes introduced. Thus, the interaction aspect, instead of staying independent, induces changes on all the agents, just like another agent-specific concern (such as behaviour, mental states, etc.). This represents a blaring contradiction of the views stated in Section 1.

4. A Reflective Architecture to Enforce Communication Laws

A multi-agent application whose interaction mechanisms exploit the ACC concept can be structured in accordance with the architecture sketched in Figure 3 (discussed later). Suitable, specialised components separately handle agent behaviour and context rule enforcement, while the "glue" that seamlessly connects them together is given by *computational reflection*. Thanks to it, agent

```
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);     // send m by Agent's send() method
  }
}
```

(b)

Figure 1. Two approaches for engineering context aware agents

programmers need not be aware of the presence of an ACC and its rules. (At most, when applicable, they might be required to handle a possible exception thrown by the underlying ACC support if a message sent violates a communication constraint[§]). Programmers are essentially freed from the burden of changing lines of code, should new application requirements cause communication rules to be added, removed or changed.

The architecture details are given in later subsections, after a brief overview of computational reflection.

[§]For such cases, 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.

4.1. 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 [25]. Usually, reflective systems are conceived as two-level systems, where a *baselevel* implements an application and the *metalevel* monitors and controls the application. As reported in the literature, reflective systems have been used to enhance existing systems with additional functionalities, for example synchronisation [33], distribution [14] and fault-tolerance [31].

One of the most widespread reflective models for object-oriented systems is the *metaobject model*, whereby a metalevel class, say `MetaFoo` in Figure 2a, can be *associated* with a baselevel class, say `Foo`. This allows instances of the metalevel class, called *metaobjects*, to intercept operations, such as *method invocation*, *object creation* and *access to object's fields*, performed on objects that are instances of the associated baselevel class [25, 18].

The association between a baselevel class and a metalevel class is performed in various ways, depending on the reflective environment employed. With OpenC++ [5] or OpenJava [32], some keywords need to be inserted by the programmer into baselevel source code, to tell a pre-compiler where to insert jumps to the metalevel. In the Javassist [6] reflective environment, instead, selected bytecode parts of baselevel Java application classes are injected with the appropriate jumps to the metalevel: thus, source code is unmodified and even unnecessary.

Because of this advantage, and of its high quality, our choice for this work has fallen on Javassist. For a better understanding of the code presented in the following, it is useful to briefly sketch how Javassist allows an existing application to be made reflective. Further information can be obtained from the Javassist documentation [7].

Assuming that the application's `main()` method is implemented in a Java class `App`, all that is required is to write for it a wrapper class `WrApp` that:

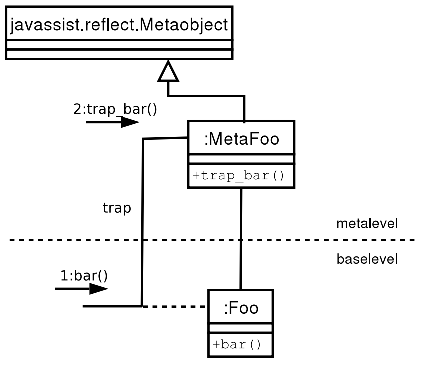
1. instantiates the Javassist class loader as, say, object `cl`;
2. calls the `cl.makeReflective()` method as many times as necessary, to connect each baselevel class to be made reflective, with the relevant metalevel class; and
3. calls the `cl.run()` method with argument `App` in order to launch the application.

These steps are concretely illustrated by the code in Figure 2b, which connects class `Foo` to metalevel class `MetaFoo`, and then calls method `run()`, which will start up the application, i.e. the `App` class, by invoking its original `main()` method. As a result, at runtime, each instance of baselevel class `Foo` will be associated with a different instance of metalevel class `MetaFoo`. Operations on the former instance will be trapped to the latter by the jumps injected by Javassist into `Foo`'s bytecode[¶].

The technique described modifies baselevel class bytecode on-the-fly, at class load-time. Another solution offered by Javassist is to employ a command line tool to inject jumps persistently into the files of the baselevel classes to be reflectively associated.

Finally, it should be noted that Javassist allows a certain degree of configuration, so that it would be possible to trap method `bar()` of the baselevel class only if the associated metalevel class has

[¶]More precisely, the baselevel instance, thanks to reflective modifications made by Javassist to its class (constructor and fields) bytecode, will have a reference to the associated metalevel instance, and can therefore call that instance's methods



```
import javassist.reflect.ClassMetaobject;
import javassist.reflect.Loader;
public class WrApp
{
  public static void main(String[] args)
    throws Throwable
  {
    Loader cl = new Loader();
    cl.makeReflective("Foo", "MetaFoo",
    "javassist.reflect.ClassMetaobject");
    cl.run("App", args);
  }
}
```

(a) (b)

Figure 2. A reflective association between a baselevel and a metalevel class, and its implementation with Javassist

a corresponding method `trap_bar()`. This helps reducing the performance overhead caused by reflection, and is exploited in our environment when appropriate.

4.2. The Reflective Software Architecture

To enforce communication laws in an agent environment, we employ a reflective software architecture consisting of a baselevel, containing agents, and a metalevel, controlling all communication among agents.

Concerning the development environment, we make exactly the same assumptions as in Section 3, which describes standard practices, that is:

A1 the agent platform is developed in an object-oriented language (such as Java), which is Jade [1] in our implementation;

A2 each agent is developed as a subclass of an `Agent` base class, supplied by the platform itself.

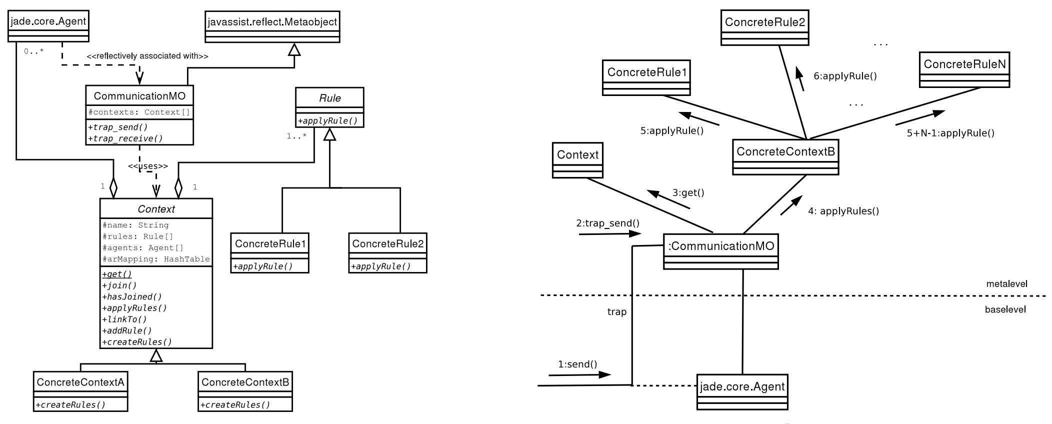
As a result a complex agent may be developed as a set of interactive classes, with object-oriented technology. Concerning communication between agents, we stipulate:

A3 the `Agent` bass class provides two methods, say `send()` and `receive()`, to exchange messages;

A4 if an agent sends a message, it ultimately does so by passing exactly that message to `Agent`'s `send()` method.

Again, **A3** is as in Section 3. **A4** is new, but quite reasonable and light.

In the proposed architecture, agents are not constrained to be aware of communication contexts; instead, upon invocation of method `send()` provided by the ancestor class `Agent`, interaction will



(a) (b)

Figure 3. Reflective software architecture for ACCs

be effectively regulated by the actual context. For the invocation of a `send()`, performed at the baselevel by an agent is intercepted and handled by the metalevel, which will enforce the rules of the appropriate communication context. Again, as a benefit of the reflective approach, these mechanisms are completely transparent to agents.

Our metalevel consists of the classes depicted in Figure 3; here the left-side (Figure 3a) depicts the class diagram of our architecture, while the collaboration diagram is shown in Figure 3b. `CommunicationMO` is the metalevel class that connects agents to contexts, by capturing agents' messages in order to apply the context rules to them. `Context` is an abstract class representing an ACC and containing, for a context: its name, the (references to) agents that joined it, each agent's role in it, and its rules. Finally, `Rule` is an abstract class representing a generic rule of an ACC.

More details on these classes, and on concrete subclasses of the latter two, are provided in the following.

4.2.1. CommunicationMO

Class `CommunicationMO` is the metalevel class intended to be reflectively associated with class `Agent` of the baselevel agent platform. This association is performed by Javassist, as described in Section 4.2 and Figure 2 specialising to Figure 4 (class `ContextAwareMultiAgentApplication`), in such a way to cause all (and only) method calls `send()` and `receive()`, performed by an `Agent` instance, to be trapped to the associated metalevel instance of `CommunicationMO`. This enables the metalevel to be aware of all the messages that agents wish to send.

Thus, upon intercepting a baselevel method call `send()` or `receive()`, `CommunicationMO` passes control to its methods `trap_send()` and `trap_receive()`, respectively. These, as shown

```java
public class MyAgent extends jade.core.Agent {
  void agentBehaviour() {
    try { send(new ACLMessage(...)); } // send a message (this is Agent's method send())
    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 javassist.reflect.Metaobject {
  // trap_send() automatically acquires control when method send() of the associated baselevel class is invoked
  public Object trap_send(int id, Object[] args) throws Throwable {
    // N.B.: getObject() returns a reference to the baselevel object associated to this
    Object agent = this.getObject();    // get a reference to the agent
    // get message of the trapped send() (its 1st arg)
    ACLMessage msg = (ACLMessage)args[0];
    // retrieve the applicable context, based on the agent's message
    Context ctx = Context.get(msg);
    // check if the agent has already joined the context
    if (!ctx.hasJoined (agent)) {
      ctx.join(agent);  // if not, join the context
    }
    // process the rules of the context for the message to be sent
    ctx.applyRules(args);
    // control gets here unless method applyRules() does not raise an exception

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

  // trap_receive automatically acquires control when method receive() of the associated baselevel class is invoked
  public Object trap_receive(int id, Object[] args) throws Throwable {
     ...
  }
}

public class ContextAwareMultiAgentApplication {
  public static void main(String[] args) throws Throwable {
    javassist.reflect.Loader cl = new javassist.reflect.Loader();
    cl.makeReflective("jade.core.Agent", "CommunicationMO",
                      "javassist.reflect.ClassMetaobject");
    cl.run("MyMultiAgentApplication", args);
  }
}
```

Figure 4. Metaobject that allows redefinition of communication primitives

for the former in the sample code of Figure 4, will determine the relevant context (see Section 4.2.2), and then apply its rules (see Section 4.2.3) to the message carried by the trapped `send()` or `receive()`.

`CommunicationMO` is designed to be a general metaobject class, easy to associate with any agent platform for the purpose of acting on its agent interaction primitives.

4.2.2. Context

Class `Context` is a metalevel abstract class used to represent an ACC. It includes methods for rule processing and context management.

Static method `get()` is called as in `get(m)`, to retrieve a reference to the context applicable when an agent exchanges message `m`. This can be observed in method `trap_send()`, Figure 4. The right `Context` object is found by `get()` in the fashion discussed in Section 4.3.

Method `join()` enables an agent to enter a certain specific context, which condition can be checked by `hasJoined()`. It is called by `CommunicationMO`, within `trap_send()`, to make an agent whose first message has been intercepted join the relevant context (see Figure 4).

Rule processing and definition is performed by methods `applyRules()`, `createRules()` and `addRule()`. Method `applyRules()` is called by `CommunicationMO`, within `trap_send()` (see Figure 4) or `trap_receive()`, to apply context rules to the message to be sent or received; in turn, it will repeatedly invoke method `applyRule()` for each relevant rule (cf. Section 4.2.3). Method `createRules()` has to be implemented by any `Context` subclass that represents a specific ACC. The body of this method will contain a call to `addRule()` for each rule defined in the particular ACC to be implemented.

Method `linkTo()` is employed to handle distribution of an ACC over different sites, according to the architecture described in Section 4.5.

4.2.3. Rule

`Rule` is a metalevel abstract class representing a generic rule for an ACC. Its concrete subclasses have to implement, in the method `applyRule()`, the code suitable for checking and enforcing the specific rule. Checks consist in analysing whether sender, receiver, structure and arguments of a message satisfy the communication constraints imposed by rules. Rule processing actions 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.

The `applyRule()` method takes a message as parameter and may return: `null`, if the rule is violated; the modified message, if the rule contains assignment expressions; the message itself otherwise.

Section 6 reports a case study showing also how a rule, expressed in the formalism introduced in Section 2, can be implemented in the proposed approach.

4.3. Handling Multiple Contexts

It was stipulated in the Introduction that a multi-agent application is built around exactly one ACC. However, an agent may take part at the same time to multiple applications, being thus simultaneously involved in different ACCs.

As a result, when `CommunicationMO` intercepts a message exchanged by the associated agent, it cannot recognise *a priori* which ACC, that is specific concrete `Context`, should be applied to the message. To handle this recognition automatically, we distinguish two situations.

In the first, it can be assumed that no role occurs in more than one application. If so, the role an agent is playing when exchanging an intercepted message (determined from the message, see Section 4.4) is sufficient for the metalevel to identify the application,

If instead the same role can be found in multiple applications, we require that agents should explicitly state in each message sent, through a user-defined slot, the application for which the message is intended.

In both the situations described, knowledge of the application is enough for the metalevel to identify the sole application's ACC.

In the latter situation, it is worth stressing the implication that agents are expected to be aware that they may be participating in more than one application. Furthermore, they must be able to identify the relevant one and encode it in a way meaningful for the metalevel. This is however quite a reasonable assumption for agents designed to take part in multiple applications.

4.4. Identifying Agent Roles

As discussed in Section 2, in the context model we refer to, the concept of *role* is an important feature of agent interaction; indeed, rules expressing ACCs are often defined using agent roles. However, messages need not always carry their sender's role. Thus, it is mandatory that the metalevel should possess the ability to detect or understand the role each agent plays within the relevant context.

However, current agent platforms only support the specification of agent names, not roles[‖]: an association mechanism *agent-name* $\mapsto$ *agent-role* is hence needed in our architecture. The mappings relevant for a given `Context` are stored into its `arMappings` attribute. In order to manage it, we have devised a dynamic strategy applicable to open multi-agent applications, in which the participating agents are unknown at design time[**].

Unfortunately, any strategy aiming at automatically discovering the role played by an agent, based on the observation of its behaviour in terms of messages sent and/or received or actions performed onto the environment, is undesirable. For in this way an agent's role could be determined only *after* a series of observations, leaving the role undetermined for some time, during which it would be impossible to apply rules.

As a solution, we require agents to notify the relevant contexts the role they intend to start playing. For this purpose, since the connection among `Agent` and `Context` objects relies only on (reflective) ACL message trapping, we introduce a special `inform` message containing the role the agent wants to play. In particular, a special name "`Context`" has to be specified as the destination agent, so that `CommunicationMO`, on intercepting the message, can understand its meaning. Should an agent start to communicate without announcing its role first, it will be attributed a default one by the metalevel.

To provide more flexibility in role specification, the ACC designer can organise the roles relevant for a context into schemes with "is-a" relationships[††]. For example, within role scheme *auction*, s/he could define roles *hearer* and *guest*, and additionally state that *guest* **is-a** *hearer*.

It should be noted that asking programmers to announce agent roles does not violate the key principle around which our approach hinges: context design is still kept strictly separated from agent design. The use of "is-a" relationships for roles contributes to such a separation, since a variation affecting roles could be faced by the ACC designer by modifying a role scheme rather than requiring a change in the agent's announcing code.

Finally, it is worth observing that role announcing messages can of course be avoided if all messages specify the sender's role.

[‖]Sometimes the specifi cation of a role within a conversation protocol is possible (e.g. initiator, responder, broker, etc.), but is only meaningful for a specifi c conversation that is a part of the application. Our notion of role has instead to do with the activities an agent is asked or able to perform within the whole application.
[**]For closed systems a simple static mapping management suffi ces.
[††]This is the typical approach of role-based access control (RBAC) models [19].

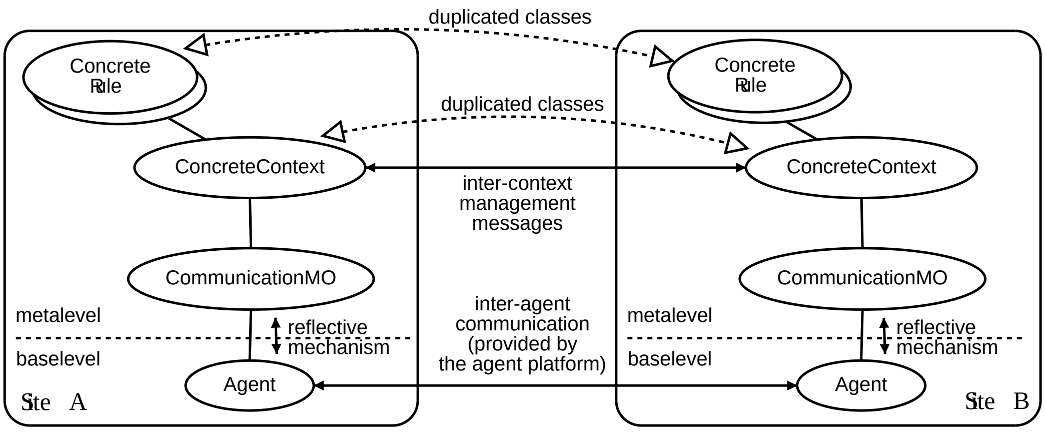


Figure 5. Agent Communication Contexts in a distributed environment

4.5. Handling Distribution

In a distributed environment, agents are located on different hosts, but interact with each other by sending messages exactly as in the centralised environment. Each baselevel class `Agent` (of the platform employed) is still associated with metalevel class `CommunicationMO`, which executes on the same host as the agent.

As depicted in Figure 5, for each ACC, the relevant `ConcreteContext` and `ConcreteRule`s metalevel classes are replicated (at design time) and placed on each site where ACC rules need to be enforced on an agent. The different instances of the same `ConcreteContext`, created in the various sites hosting the agents bound by the same ACC, must therefore be linked together so that they may share a global state. For this purpose, the `ConcreteContext()` constructor contains as many calls to method `linkTo(`*`sitename`*`)` as the involved sites. Thanks to the described infrastructure, when a message from a local agent triggers a change on the state of the associated instance of `ConcreteContext`, this new value is promptly communicated to other instances running on the other sites. Changes of state mainly concern agents that join or leave a context.

5. Generating Contexts and Integrating them with Agents

To support computer-assisted engineering and deploying/integration of ACCs, we have implemented a tool, called *ACC Builder*, capable of (*i*) automatically generating, for a given multi-agent application, the relevant metalevel classes `ConcreteContext` and `ContextRule`s, and (*ii*) integrating them with the chosen agent platform. This includes the automatic generation of `CommunicationMO`, which is application independent but depends on the agent platform.

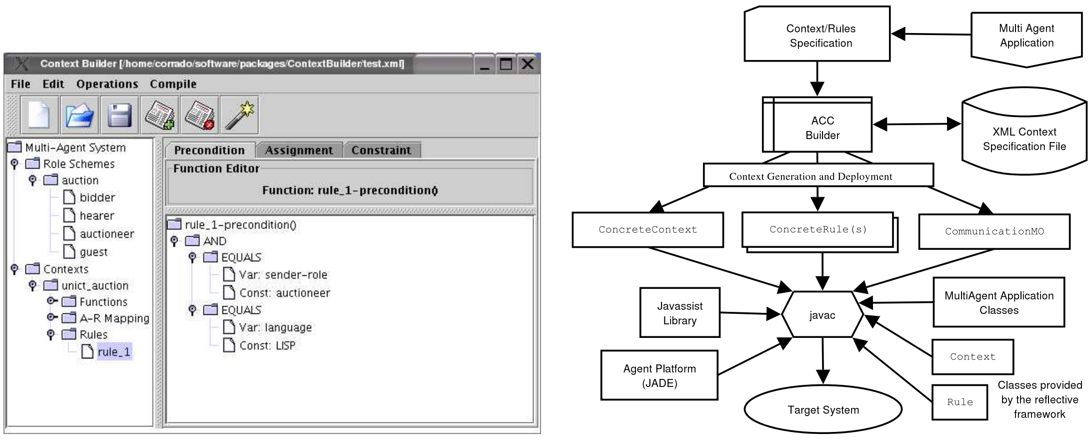


Figure 6. ACC Builder

The ACC Builder (a screen snapshot of which is given in Figure 6, left) operates by allowing the user to define and manage a library of different contexts[‡‡]. In particular, the user is able to handle the following entities:

1. *agent role schemes* (such as `auction` in Figure 6) to be associated with contexts;
2. *contexts*, (such as `unict_auction` in Figure 6) each consisting of

    (a) a set of *predicate definitions for receivers*, expressed as first-order logic lambda functions;
    (b) *agent-name* $\mapsto$ *agent-role mappings* (cf. Section 4.4);
    (c) the *communication rules*, expressed as first-order logic expressions (as Figure 6 shows for the precondition of `rule_1`, rules and functions are expressed using a prefix notation).

The agent platform used to build the multi-agent application is specified on selection of the "Generate/Deploy" option (the "Wand" button in Figure 6); this is needed to generate the suitable metalevel classes to be associated to the existing baselevel classes of the platform. The resulting process, sketched in Figure 6, right, generates the source code of the specific `Context` and `Rules` classes, and runs the Java compiler to compile/link everything together, that is: generated metalevel classes (including `CommunicationMO` and the concrete `Contexts` and `Rules`), abstract metalevel classes `Context` and `Rule`, agent (baselevel) classes, classes of the Javassist library and the agent platform. This will produce the target system, which is now ready to run.

[‡‡]Eventually, consistently with our assumption that a multi-agent application is built around one context, the ACC Builder will deploy exactly one (user-selected) ACC in the application generated

6. Case Study: Assisting a Web Browser

Assistance activities for users can be designed in such a way that several autonomous agents be employed to work on different small tasks. Such agents, which are used as assistants, need to communicate in order to share results and to interact with the application and the user. Another type of agent is needed to play the role of a coordinator between assistants and the application, so as to sense application activities and intervene when appropriate to pour assistants results into the application.

We have been experimenting with several assistants for a web browser [12]. Some assistants, called *Advisors*, are given the ability to modify the appearance of portions of web pages working autonomously and asynchronously from each other and the application. These assistants send *Advice* messages to communicate results that change the application. Changes they can carry out concern: words or paragraphs colours, paragraphs order, links or text additions, etc. In order to provide suggestions, Advisors exploit the accumulated knowledge about user behaviour. Assistant *Categoriser* analyses a web page and returns the category to which it belongs, by sending *Category* message. Assistant *UserProfiler* extracts the user profile by observing user activities and sends *Profile* messages. Assistant *DataExtractor* extracts useful data from web pages and sends *Data* messages. The latter two assistants need to know the category of the visited pages to build their database. Other assistants have also been designed and implemented, for example to handle the repository of extracted data and find data on the web on the basis of the user profile. Agent *Coordinator* has been employed to receive results from the assistants and choose the appropriate timing to unobtrusively change the web browser behaviour. Moreover, Coordinator composes the modified portions of pages received from Advisor assistants, using a priority system to make decisions in case multiple, conflicting changes have been proposed on the same portion.

The following messages are typically exchanged between assistants.

1. Coordinator announces that a new page is being accessed by the user.
2. Advisor assistants propose changes to the page passing a modified web page to Coordinator.
3. Categoriser communicates the page category to Coordinator.
4. UserProfiler sends the user profile to Coordinator, if it has been effected by the last page analysed.
5. DataExtractor sends Coordinator any interesting data found on the last page analysed.
6. Coordinator announces that a page has become active or has been closed.

According to the interactions above, the following communication rules, which are specified in Figure 7 (top) can be derived:

r_1 Messages from Coordinator that concern visited, active and open pages, which were originally sent to all the Advisors, have to be forwarded to all the assistants, since it is useful for them to know about the state of the page.

r_2 Category messages that Categoriser sends to Coordinator must also be sent to UserProfiler and DataExtractor.

r_3 Messages containing user profiles, which UserProfiler sends to Coordinator, are also sent to Advisors.

$$
\begin{array}{lcl}
r_1 & \overset{\text{def}}{=} & \sigma_n = Coordinator \wedge \omega_m = trapped\_actions \Rightarrow v_m \leftarrow \lambda a.\, role(a) = Assistant \\
r_2 & \overset{\text{def}}{=} & \sigma_n = Categoriser \wedge \rho_m(Coordinator) \wedge \omega_m = categories \\
 & & \Rightarrow v_m \leftarrow \lambda a.\, (a = UserProfiler \vee a = DataExtractor) \\
r_3 & \overset{\text{def}}{=} & \sigma_n = UserProfiler \wedge \rho_m(Coordinator) \wedge \omega_m = user\_profile \\
 & & \Rightarrow v_m \leftarrow \lambda a.\, role(a) = Advisor
\end{array}
$$

```java
public class ContextAssist extends Context {
  public ContextAssist() {
    super();
    setName ("Web-Assistance");
  }

  public void createRules() {
    addRule(new RuleAssist1());
    addRule(new RuleAssist2());
    addRule(new RuleAssist3());
  }
}

public class RuleAssist1 extends Rule {
  public ACLMessage applyRule(ACLMessage m) {
    if (m.getSenderRole().equals("Coordinator") &&
        m.getOntology().equals("trapped_actions")) {
      m.setVirtualReceiverPredicate (new Lambda(new HasRole("Assistant")));
    }
    return m;
  }
}

public class RuleAssist2 extends Rule {
  public ACLMessage applyRule(ACLMessage m) { /* ... */ }
}

public class RuleAssist3 extends Rule {
  public ACLMessage applyRule(ACLMessage m) { /* ... */ }
}
```

Figure 7. Rules defined in the case study and the code generated by the ACC Builder

As the sample code in Figure 7 (bottom frame) implementing such rules amounts to defining the context in a class `ContextAssist`, subclass of abstract class `Context`, and the rules r_1, r_2 and r_3 in the classes `RuleAssist1`, `RuleAssist2` and `RuleAssist3`, derived from the abstract class `Rule`. In particular, the code shown is a portion of that generated by the ACC Builder, which has been employed in engineering this case-study.

The UML diagram of the resulting multi-agent application is essentially that of Figure 3, taking into account that classes `ConcreteContext`s and `ConcreteRule`s specialise here into `ContextAssist`s and `RuleAssist`s respectively.

6.1. Reusing ACCs

A scenario in which a reuse opportunity for ACCs arises is when new assistant agents are added to enhance a web browser's standard assistance facilities. In such a case, no changes are required to `CommunicationMO` or `ContextAssist` to work with the additional agents, because the presence of the context is transparent (i.e. hidden) to assistants, and rules r_1 to r_3 apply without changes to many different types of assistance activities.

A further scenario is the development of assistance activities for an application different from a web browser, such as a word processor. In this case each assistant could provide the user with: (*i*) a list of all the personal documents related to the edited one, on the basis of the determined categories; (*ii*) the synonyms of the word next to the cursor; (*iii*) the results of a web search, within the category of the document, for the last word typed. The suggested lists of results appear on a separate pane, so the assistant does not need to modify the current document. The Categoriser assistant would be helpful in this scenario. The previous metalevel classes can be re-used again as they are. In particular `ContextAssist`, the class implementing rules, would use: (*a*) rule r_1, to inform assistants about user switching between edited documents or introducing changes into them; (*b*) rule r_2, to pass the result of the categorisation to these new assistants. Rule r_3 would never match at runtime, since no UserProfiler agent is active.

7. Conclusions

We have proposed an approach for engineering a multi-agent application, allowing separate development of behavioural and communication/coordination concerns. The latter are taken care of by providing the definition of an Agent Communication Context, which consists in a set of rules governing agent interactions within a certain multi-agent application. Since agent communication concerns and agent behaviour are arguably independent, we have advocated to clearly separate them by a development approach based on a reflective architecture.

The automatic enforcement of communication laws provides two general advantages: (a) agents cannot escape it, whatever their programmers' intentions; (b) on the other hand, programmers are not required to take such laws into account in the code they write. The latter fact is particularly useful in the development of "open" multi-agent systems, which can be dynamically joined or left after startup by participating agents. As a result, programmers do not know in advance which context an agent will find itself in. Therefore, appropriate design, deployment and runtime support, of the kind proposed in this work, is in order.

However, despite the noted advantages of considering behaviour and coordination independent, hence developing agents and contexts separately, it is nevertheless unavoidable that in some cases context rules may have an impact on agents. For example, it might be necessary for agents to be aware of communication rules, so as to decide when to enter/exit a context; or, for security reasons, an agent might want to keep some messages private even in contexts that would broadcast them.

In any case, the proposed approach is flexible enough that the degree of context awareness appropriate for agents can be easily introduced by designers. Indeed, we plan to deal with such issues as a further extension of the work presented. We envisage that a configuration phase would be performed, perhaps 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 this work's approach, whereby no a priori knowledge of ACCs should be required for the development of agents.

ACKNOWLEDGEMENTS

This work has been supported by the research projects "FIRB-WEBMINDS" and "COFIN-ANEMONE" of the Italian MIUR. The authors also wish to thank the anonymous referees for their insightful comments and advice in revising the paper.

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. G. Cabri, L. Leonardi, and F. Zambonelli. Mobile-Agent Coordination Models for Internet Applications. *IEEE Computer*, 33(2), February 2000.
3. G. Cabri, L. Leonardi, and F. Zambonelli. BRAIN: A Framework for Flexible Role-Based Interactions in Multiagent Systems. In R. Meersman and Z. Tari, editors, *On the Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, Berlin, 2003. Lecture Notes in Computer Science 2888, Springer Verlag.
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. A Metaobject Protocol for C++. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95)*, Sigplan Notices, pages 285–299, 1995.
6. S. Chiba. Load-time Structural Reflection in Java. In *Proceedings of the ECOOP 2000*, volume 1850 of *Lecture Notes in Computer Science*, 2000.
7. S. Chiba. *Javassist tutorial*. http://www.csg.is.titech.ac.jp/~chiba/javassist/tutorial/tutorial.html, 2002.
8. R. Conte and C. Castelfranchi. *Cognitive and Social Action*. UCL Press., 1995.
9. 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.
10. 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.
11. 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)".
12. A. Di Stefano, G. Pappalardo, C. Santoro, and E. Tramontana. A Multi-Agent Reflective Architecture for User Assistance and its Application to E-Commerce. In *Cooperative Information Agents (CIA 2002)*. LNAI Springer, Sept. 18-20 2004.
13. A. Di Stefano, G. Pappalardo, C. Santoro, and E. Tramontana. Enforcing Agent Communication Laws by means of a Reflective Framework. In *Symposium on Applied Computing (SAC 2004). Track on Coordination Models*. ACM, Mar. 14-17 2004.
14. 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.
15. 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.
16. A. Di Stefano and C. Santoro. Modeling Multi-Agent Communication Contexts. In *First Intl. ACM Joint Conference on Autonomous Agents and Multi-Agent Systems*. Bologna, Italy, July 15-19 2002.
17. A. Di Stefano and C. Santoro. Integrating Agent Communication Contexts in JADE. *Telecom Italia Journal EXP*, Sept. 2003.
18. 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.
19. D. F. Ferraiolo, D. R. Kuhn, and R. Chandramouli. *Role-Based Access Control*. Artech House, 2003.
20. 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.
21. FIPA-OS Home Page. *http://fipa-os.sourceforge.net*.
22. Foundation for Intelligent Physical Agents. FIPA-ACL Specification, available at http://www.fipa.org/specs/fipa00061/.
23. 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.
24. Y. Labrou, T. Finin, and Y. Peng. Agent Communication Languages: the Current Landscape. *IEEE Intelligent Systems*, March-April 1999.

25. 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.
26. 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.
27. 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.
28. G. Papadopoulos and F. Arbab. Coordination models and languages. In *Advances in Computer*, volume 46. Academic Press, 1998.
29. 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.
30. M. P. Singh. Agent Communication Languages: Rethinking the Principles. *IEEE Computer*, 31(12):40–47, December 1998.
31. 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.
32. M. Tatsubori, S. Chiba, K. Itano, and M.-O. Killijian. OpenJava: A Class-Based Macro System for Java. In W. Cazzola, R. J. Stroud, and F. Tisato, editors, *Reflection and Software Engineering*, volume 1826 of *Lecture Notes in Computer Science*, pages 117–133. Springer-Verlag, June 2000.
33. 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.
34. 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.
35. 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.