Bridging Agent Theory and Object Orientation: Agent-Like Communication Among Objects

Matteo Baldoni[1], Guido Boella[1], and Leendert van der Torre[2]

[1] Dipartimento di Informatica. Università di Torino - Italy
{baldoni,guido}@di.unito.it
[2] University of Luxembourg
leendert@vandertorre.com

Abstract. This paper begins with the comparison of the message-sending mechanism, for communication among agents, and the method-invocation mechanism, for communication among objects. Then, we describe an extension of the method-invocation mechanism by introducing the notion of "sender" of a message, "state" of the interaction and "protocol" using the notion of "role", as it has been introduced in the `powerJava` extension of Java. The use of roles in communication is shown by means of an example of protocol.

1 Introduction

The major differences of the notion of agent w.r.t. the notion of object are often considered to be "autonomy" and "proactivity" [25]. Less attention has been devoted to the peculiarities of the *communication capabilities* of agents, which exchange messages while playing roles in protocols. For example, in the contract net protocol (CNP) an agent in the role of initiator starts by *asking for* bids, while agents playing the role of participants can *propose* bids which are either *accepted* or *rejected* by the Initiator.

The main features of communication among agents which emerge from the CNP example are the following:

1. The message identifies both its *sender* and its *receiver*. E.g., in FIPA the acceptance of a proposal is:
   ```
   (accept-proposal :sender i :receiver j :in-reply-to
       bid089 :content X :language FIPA-SL).
   ```
2. The interaction with each agent is associated to a *state* which evolves according to the messages that are exchanged. The meaning of the messages is influenced by the state. E.g., in the FIPA iterated contract net protocol, a "call for proposal" is a function of the previous calls for proposals, i.e., from the session.
3. Messages are produced according to some *protocol* (e.g., a call for proposal must be followed by a proposal or a reject).
4. The sender and the receiver play one of the *roles* specified in the protocol (e.g., initiator and participant in the contract net protocol).

5. Communication is *asynchronous*: the response to a message does not necessarily follow it immediately. E.g., in the contract net protocol, a proposal must follow a call for proposal and it must arrive, no matter when, before a given deadline.
6. The receiver autonomously decides to comply with the message (e.g., making a proposal after a call for proposal).

The message metaphor has been originally used also for describing method calls among objects, but it is not fully exploited. In particular, message-exchange in the object oriented paradigm has the following features:

1. The message is sent to the receiver without any information concerning the sender.
2. There is no state of the interaction between sender and receiver.
3. The message is independent from the previous messages sent and received.
4. The sender and the receiver do not need to play any role in the message exchange.
5. The interaction is synchronous: an object waits for the result of a method invocation.
6. The receiver always executes the method invoked if it exists.

These two scenarios are rather different but we believe that the object-oriented (OO) paradigm can learn something from the agent-oriented world. The research question of this paper is thus: is it profitable to introduce in the OO paradigm concepts taken from agent communication? how can we introduce in the OO paradigm the way agents communicate? And as subquestions: which of the above properties can be imported and which cannot? How to translate the properties which can be imported in the OO paradigm? What do we learn in the agent-oriented world from this translation?

The methodology that we use in this paper is to map the properties of agent communication to an extension of Java, `powerJava` [3,4,5], which adds roles to objects. Roles are used to represent the sender of a message (also known as the "player of the role"), to represent the state of the interaction via role instances, allowing the definition of protocols and asynchronous communication as well as the representation of the different relations between objects.

The choice of the Java language is due to the fact that it is one of the prototypical OO programming languages; moreover, MAS systems are often implemented in Java and some agent programming languages are extensions of Java, e.g., see the Jade framework [8] or the JACK software tool [24]. In this way we can directly use complex interaction and roles offered by our extension of Java when building MAS systems or extending agent programming languages.

Furthermore, we believe that in order to contribute to the success of the Autonomous Agents and Multiagent Systems research, the theories and concepts developed in this area should be applicable also to more traditional views. It is a challenge for the agent community to apply its concepts outside strictly agent-based applications. The OO paradigm is central in Computer Science and, as observed and suggested also by Juan and Sterling [18], before AO can be widely used in industry, its attractive theoretical properties must be first translated to simple, concrete constructs and mechanisms that are of similar granularity as objects.

The paper is organized as follows. In Section 2 we show which properties of agent communication can be mapped to objects. In Section 3 we introduce how we model interaction in `powerJava` and in Section 4 we discuss how to use roles in order to model complex forms of interaction between object inspired by agent interaction, we also illustrate the contract net protocol among objects using `powerJava`. Conclusions end the paper.

2 Communication Between Objects

When approaching an extension of a language or of a method, the first issue that should be answered is whether that extension brings along some advantages. In our specific case, the question can be rephrased as: Is it useful for the OO paradigm to introduce a notion of communication as developed in MAS? We argue that there are several acknowledged limitations in OO method invocation which could be overcome, thus realizing what we could call a "session-aware interaction".

First of all, objects exhibit only one state in all interactions with any other object. The methods always have the same meaning, independently of the identity or type of the object from which they are called.

Second, the operational interface of Abstract Data Types induces an *asymmetrical* semantic dependency of the callers of operations on the operation provider: the caller *takes the decision* on what operation to perform and it relies on the provider to carry out the operation. Moreover, method invocation does not allow to reach a minimum level of "control from the outside" of the participating objects [2].

Third, the state of the interaction is not maintained and methods always offer the same behavior to all callers under every circumstance. This limit could be circumvented by passing the caller as a further parameter to each method and by indexing, in each method, the possible callers.

Finally, even though asynchronous method calls can be simulated by using buffers, it is still necessary to keep track of the caller explicitly.

The above problems can be solved by using the way communication is managed between agents and defining it as a primitive of the language. By adopting agent-like communication, in fact, the properties presented in Section 1 – with the only exception of autonomy, (6), which is a property distinguishing agents from objects – can be rewritten as in the following:

1. When methods are invoked on an object also the object invoking the method (the "sender") must be specified.
2. The state of the interaction between two objects must be maintained.
3. In presence of state information, it is possible to implement interaction protocols because methods are enabled to adapt their behavior according to

the interaction that has occurred so far. So, for instance, a proposal method whose execution is not preceded by a call for proposals can detect this fact and raise an exception.

4. The object whose method is invoked and the object invoking the method play each one of the roles specified by the other, and they respect the *requirements* imposed on the roles. Intuitively, requirements are the capabilities that an object must have in order to be able to play the role.
5. The interaction can be asynchronous, thanks to the fact that the state of the interaction is maintained.

For a better intuition, let us consider as an example the case of a simple interaction schema which accounts for two objects. We expect the first object to wait for a "call for proposal" by the other object; afterwords, it will invoke the method "propose" on the caller. The idea is that the call for proposal can be performed by different callers and, depending on the caller, a different information (e.g. the information that it can understand) should be returned by the first object. More specifically, we can, then, imagine to have an object `a`, which exposes a method `cfp` and waits for other objects to invoke it. After such a call has been performed, the object `a` invokes a method `propose` on the caller. Let us suppose that two different objects, `b` and `c`, do invoke `cfp`. We desire the data returned by `a` to be different for the two callers.

Since we look at the agent paradigm the solution is to have two different interaction states, one for the interaction between `a` and `b` and one for the interaction between `a` and `c`. In our terminology, `b` and `c` interact with `a` in two distinct roles (or better, *role instances*) which have distinct states: thus it is possible to have distinct behaviors depending on the invoker. If the next move is to "accept" a proposal, then we must be able to associate the acceptance to the right proposal.

In order to implement these properties we use the notion of role introduced in the `powerJava` language in a different way with respect to how it has been designed for.

3 Modelling Interaction with `powerJava`

In [1,12,21,23] the concept of "role" has been proved extremely useful in programming languages for several reasons. These reasons range from dealing with the separation of concerns between the core behavior of an object and its interaction possibilities, reflecting the ontological structure of domains where roles are present, from modelling dynamic changes of behavior in a class to fostering coordination among components. In [3,4,5] the language `powerJava` is introduced: `powerJava` is an extension of the well-known Java language, which accounts for roles, defined within social entities like institutions, organizations, normative systems, or groups [7,14,26]. The name `powerJava` is due to the fact that the key feature of the proposed model is that institutions use roles to supply the *powers* for acting (*empowerment*). In particular, three are the properties that characterize roles, according to the model of normative multiagent systems [9,10,11]:

Foundation: A (instance of) role must always be associated with an instance of the institution it belongs to (see Guarino and Welty [16]), besides being associated with an instance of its player.

Definitional dependence: The definition of the role must be given inside the definition of the institution it belongs to. This is a stronger version of the definitional dependence notion proposed by Masolo *et al.* [19], where the definition of a role must include the concept of the institution.

Institutional empowerment: The actions defined for the role in the definition of the institution have access to the state and actions of the institution and to the other roles' state and actions: they are powers.

Roles require to specify both *who can play the role* and *which powers are offered* by the institution in which the role is defined. The objects which can play the role might be of different classes, so that roles can be specified independently of the particular class playing the role. For example a role customer can be played both by a person and by an organization. Role specification is a sort of double face interface, which specifies both the methods required to a class playing the role (*requirements*, keyword "playedby") and the methods offered to objects playing the role (*powers* keyword "role"). An object, which plays a role, is empowered with new methods as specified by the interface.

To make an example, let us suppose to have a printer which supplies two different ways of accessing to it: one as a normal user, and the other as a superuser. Normal users can print their jobs and the number of printable pages is limited to a given maximum. Superusers can print any number of pages and can query for the total number of prints done so far. In order to be a user one must have an account which is printed on the pages. The role specification for the user is the following:

```
role User playedby AccountedPerson {
  int print(Job job);
  int getPrintedPages();
}

interface AccountedPerson {
  Login getLogin();
}
```

The superuser, instead:

```
role SuperUser playedby AccountedPerson {
  int print(Job job);
  int getTotalPrintedPages();
}
```

Requirements must be implemented by the objects which act as players.

```
class Person implements AccountedPerson {
  Login login;  // ...
  Login getLogin() {
    return login;
  }
}
```

Instead, powers are implemented in the class defining the institution in which the role itself is defined. To implement roles inside an institution we revise the notion of *Java inner class*, by introducing the new keyword `definerole` instead of `class` followed the name of the role definition that the class is implementing.

```
class Printer {
  final static int MAX_PAGES_PER_USER;
  private int totalPrintedPages = 0;

  private void print(Job job, Login login) {
    totalPrintedPages += job.getNumberPages();
    // performs printing
  }

  definerole User {
    int counter = 0;
    public int print(Job job) {
      if (counter > MAX_PAGES_USER)
        throws new IllegalPrintException();
      counter += job.getNumebrPages();
      Printer.this.print(job, that.getLogin());
      return counter;
    }
    public int getPrintedPages(){
      return counter;
    }
  }

  definerole SuperUser {
    public int print(Job job) {
      Printer.this.print(job, that.getLogin());
      return totalPrintedPages;
    }
    public int getTotalPrintedpages() {
      return totalPrintedPages;
    }
  }

}
```

Roles cannot be implemented in different ways in the same institution and we do not consider the possibility of extending role implementations (which is, instead, possible with inner classes), see [5] for a deeper discussion.

As a Java inner class, a role implementation has access to the private fields and methods of the outer class (in the above example the private method *print* of *Printer* used both in role *User* and in role *SuperUser*) and of the other roles defined in the outer class. This possibility does not disrupt the encapsulation principle since all roles of an institution are defined by who defines the institution itself. In other words, an object that has assumed a given role, by means of it, has access and can change the state of the corresponding institution and of the sibling roles. In this way, we realize the powers envisaged by our analysis of the notion of role.

The class implementing the role is instantiated by passing to the constructor an instance of an object satisfying the requirements. The behavior of a role instance depends on the player instance of the role, so in the method implementation the player instance can be retrieved via a new reserved keyword: `that`, which is used only in the role implementation. In the example the invocation of `that.getLogin()` as a parameter of the method `print`.

All the constructors of all roles have an implicit first parameter which must be passed as value the player of the role. The reason is that to construct a role we need both the institution the role belongs to (the object the construct `new` is invoked on) and the player of the role (the first implicit parameter). For this reason, the parameter has as its type the requirements of the role. A role instance is created by means of the construct `new` and by specifying the name of the "inner class" implementing the role which we want to instantiate. This is like it is done in Java for inner class instance creation. Differently than other objects, role instances do not exist by themselves and are always associated to their players.

Methods can be invoked from the players, given that the player is seen in its role. To do this, we introduce the new construct

receiver `<-(`*role*`)` *sender*

This operation allows the sender (player of the role) to use the powers given by "role" when it interacts with the receiver (institution) the role belongs to. It is similar to *role cast* as introduced in [3,4,5] but it stresses more strongly the interaction aspect of the two involved objects: the sender uses the role defined by the receiver for interacting with it. Let us see how to use this construct in our running example. The first instructions in the main create a printer object `hp8100` and two person objects, `chris` and `sergio`. `chris` is a normal user while `sergio` is a superuser. Indeed, instructions four and five define the roles of these two objects w.r.t. the created printer. The two users invoke method `print` on `hp8100`. They can do this because they have been empowered of printing by their roles. The act of printing is carried on by the private method `print`. Nevertheless, the two roles of `User` and `SuperUser` offer two different way to interact with it: `User` counts the printed pages and allows a user to print a job if the number of pages printed so far is less than a given maximum; `SuperUser` does not have such a limitation. Moreover, `SuperUser` is empowered also for viewing the total number of printed pages. Notice that the page counter is maintained in the role state and persists through different calls to methods performed by a same sender/player towards the same receiver/institution as long as it plays the role.

```
 class PrintingExample {
  public static void main(String[] args) {

    Printer hp8100 = new Printer();
    Person chris = new Person();
    Person sergio = new Person();

    hp8100.new User(chris);
    hp8100.new SuperUser(sergio);

    (hp8100 <-(User) chris).print(job1);
    (hp8100 <-(SuperUser) sergio).print(job2);
    (hp8100 <-(User) chris).print(job3);

    System.out.println("Chris has printed " +
      (hp8100 <-(User) chris).getPrintedPages() + " pages");
    System.out.println("The printer hp8100 has printed a total of " +
      (hp8100 <-(User) sergio).getTotalPrintedPages() + " pages");

  }
}
```

By maintaining a state, a role can be seen as realizing a *session-aware interaction*, in a way that is analogous to what done by cookies or Java sessions for JSP and Servlet. So in our example, it is possible to visualize the number of currently printed pages, as in the above example. Note that, when we talk about playing a role we always mean playing a role instance (or *qua individual* [19] or *role enacting agent* [13]) which maintains the properties of the role.

An object has different (or additional) properties when it plays a certain role, and it can perform new activities, as specified by the role definition. Moreover, a role represents a specific state which is different from the player's one, which can evolve with time by invoking methods on the roles. The relation between the object and the role must be transparent to the programmer: it is the object which has to maintain a reference to its roles. However, a role is not an independent object, it is a facet of the player.

Since an object can play multiple roles, the same method will have a different behavior, depending on the role which the object is playing when it is invoked. It is sufficient to specify which the role of a given object, we are referring to, is. In the example `chris` can become also `superuser` of `hp8100`, besides being a normal `user`

```
    hp8100.new SuperUser(chris);
    (hp8100 <-(SuperUser) chris).print(job4);
    (hp8100 <-(User) chris).print(job5);
```

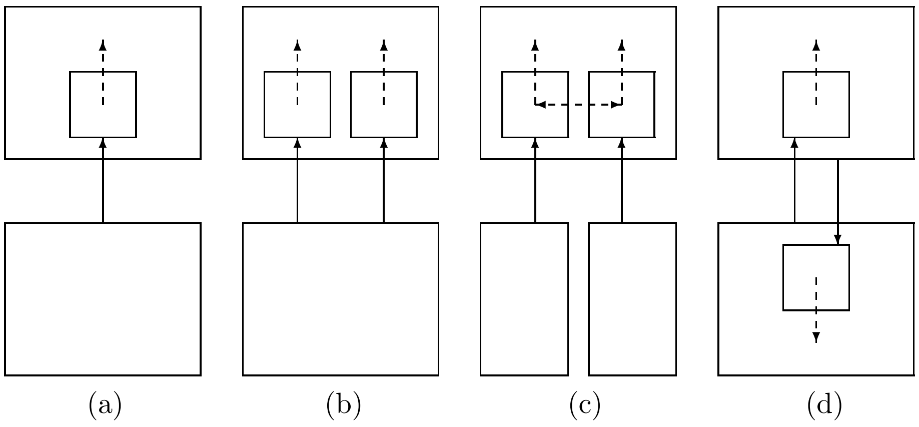

Fig. 1. The possible uses of roles

Notice that in this case two different sessions will be kept: one for `chris` as normal `user` and the other for `chris` as `superuser`. Only when it prints its jobs as a normal `user` the page counter is incremented.

4 Uses of Roles in `powerJava`

In this paper we exploit the language `powerJava` in a new way which allows modelling the agent inspired vision of interaction among objects. The basic idea of `powerJava` is that objects (e.g. `hp8100`), called institutions, are composed of roles which can access the state of the institution and of other sibling roles and, thus, can coordinate with each other [3]. However, since an institution is just an object which happens to contain role implementations, nothing prevents us to consider *every object* as an institution, and to consider the roles as different ways of interacting with it. Many objects can play the same role (a printer can have many users) as well as the same object can play different roles (`chris` is both a user and a superuser). Each role instance has its own state, which represents the state of the interaction with the player of the role.

Figure 1 illustrates the different interaction possibilities given by roles, which do not exclude the traditional direct interaction with the object when roles are not necessary. Other possibilities like sessions shared by multiple objects are not considered for space reasons.

Arrows represent the relations between players and their respective roles, *dashed arrows* represent the access relation between objects, i.e., their powers.

- Drawing (a) illustrates the situation where an object interacts with another one by means of the role offered by it. This is, for instance, the case of `sergio` being a `SuperUser` of `hp8100`.
- Drawing (b) illustrates an object (e.g., `chris`) interacting in two different roles with another one (`hp8100` in the example). This situation is used when

an object implements two different interfaces for interacting with it, which have methods (like `print`) with the same signature but with different meaning. In our model the methods of the interfaces are implemented in the roles offered by the objects to interact with them. The role represent also the different sessions of the interaction with the different objects.
- Drawing (c) illustrates the case of two objects which interact by means of the roles of an institution (which can be considered as the context of execution). This is the original case, `powerJava` has been developed for [3]; in this paper, we used as a running example the well-known 5 philosophers scenario. The institution is the table, at which philosophers are sitting and coordinate to take the chopsticks and eat since they can access the state of each other. The coordinated objects are the players of the role `chopstick` and `philosopher`. The former role is played by objects which produce information, the latter by objects which consume them. None of the players contains the code necessary to coordinate with the others, which is supplied by the roles.
- In drawing (d) two objects interact with each other, each playing a role offered by the other. This is often the case of interaction protocols: e.g., an object can play the role of *initiator* in the Contract Net Protocol if and only if the other object plays the role of *participant*. Indeed, the Contract Net Protocol is reported as an example in the following section.

The four cases can be combined to represent more complex interaction schemas.

This view of roles inspires a new vision of the the OO paradigm, which accounts for the way humans conceptualize objects performed in philosophy and above all in cognitive science [15]. In particular, cognitive science has highlighted that properties of objects are not objective properties of the world, but they depend on the properties of the agent conceptualizing the object: objects are conceptualized on the basis of what they "afford" to the actions of the entities interacting with them. Thus, different entities conceptualize the same object in different ways. We translate this intuition in the fact that an object offers different methods according to which type of object it is calling it: the methods offered (the powers of a role) depend on the requirements offered by the caller.

4.1 The Contract Net Protocol Example

Hereafter, we report an example set in the framework of interaction protocols, describing an implementation of the well-known *contract net* protocol. The example follows the interaction schema (d), reported in the previous section, and it is substantially different than the analogous example reported in a previous paper [4]. In fact, the solution proposed here is *distributed* instead of being *centralized* (let us denote by this name a solution respecting case (c) in the previous section). The advantage of the old solution was that players did not need to know anything about the coordination mechanism. In this case, instead, each object also supplies a role for its counterpart, which describes the powers that are given to the counterpart in the interaction. For instance, the object that will play the `initiator` role will define the powers of the `participants`, and vice versa. The powers are the messages that the `initiator` will understand; this is very different than our previous proposal, where the powers only allowed to start a negotiation or to take part to a negotiation, depending on the role, and the exchanged messages were hidden inside the institution.

In this new version, roles are also used for maintaining interaction sessions. In the following example, `refuseProposal` can be executed only if `cfp` has already been executed, this can be tracked thanks to the role state and, in particular, thanks to variable `state`.

Observe that when the object, offering a role, is supposed to answer something, it needs to invoke a method, which is supplied as a power of a role, which is in turn offered by the object to which it is responding. In the contract net, a possible answer to a `cfp` is the performative `propose`. In this case, see also the code reported at the end of this section, the above interaction is implemented by the instruction:

```
(that <-(Participant) Peer.this).propose(getProposal(task))
```

Here, `Peer.this` refers to the object offering the role `initiator`; such an object means to play the role of `Participant` and, in particular, to invoke the power `propose` offered by this role. The role `participant` is offered by the object which is currently playing the initiator (identified in the above code line by `that`), see Fig. 2.

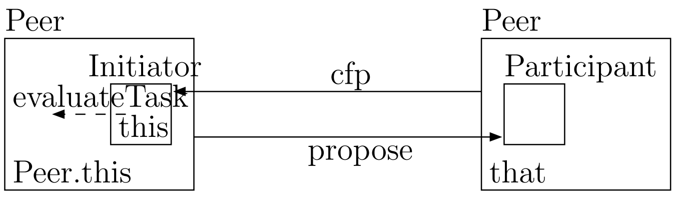

Fig. 2. Description of the interaction between an Initiator and a Participant, when, after a "cfp" performative, the answer will be a "propose" performative

The communication is asynchronous, since the proposal is not returned by the `cfp` method.

Notice that an object which is currently playing the role of participant in a given interaction, can at the same time play the role of initiator in another interaction. See the method `evaluateTask`, in which a new interaction is started for executing a subtask by creating the two roles in the respective objects and by linking players to them:

```
role Initiator playedby InitiatorReq {
  void cfp(Task task);
  void rejectProposal(Proposal proposal);
  void acceptProposal(Proposal proposal);
```

```
}
interface InitiatorReq { // must implement the role specification Participant
}

role Participant playedby ParticipantReq {
  void propose(Proposal proposal);
  void refuse(Task task);
  void inform(Object result);
  void failure(Object error);
}

interface ParticipantReq { // must implement the role specification Initiator
}

class Peer implements ParticipantReq, InitiatorReq
{

  definerole Initiator {
    final static int STATE_1 = 1;
    final static int STATE_2 = 2;
    int state = STATE_1;

    public void cfp(Task task) {
      if (state != STATE_1)
        throws new IllegalPerfomativeException();
      state = STATE_2;
      if (evaluateTask(task))
        (that <-(Participant) Peer.this).propose(getProposal(task));
      else
        (that <-(Participant) Peer.this).refuse(task);
    }

    public void refuseProposal(Proposal proposal) {
      if (state != STATE_2)
        throws new IllegalPerformativeException();
      removeProposal(proposal);
      state = STATE_1;
    }

    public void acceptProposal(Proposal proposal) {
      if (state != STATE_2)
        throws new IllegalPerfomativeException();
      try {
        (that <-(Participant)Peer.this).inform(performTask(proposal,task));
      } catch(TaskExecException err) {
        (that <-(Participant) Peer.this).failure(err);
      }
      state = STATE_1;
    }
  }

  private boolean evaluateTask(Task task) {
    Task subTask;  // ...
    this.new Participant(peer);
    peer.new Initiator(this);
    (peer <-(Initiator) this).cfp(subTask);   // ...
  }

  definerole Initiator { ... }

}
```

5 Conclusion

In this work, we have proposed the introduction of a form of interaction between objects, in the OO paradigm, which borrows from the theory about agent communication. The main advantage is to allow session-aware interactions in which the history of the occurred method invocations can be taken into account and, thus, introducing the possibility of realizing, in a quite natural way, agent interaction protocols. The key concept which allows communication is the role played by an object in the interaction with another object. Besides proposing a model that describes this form of interaction, we have also proposed an extension of the language `powerJava` that accounts for it.

One might wonder whether the introduction of agent-like communication between objects gives us some feedback to the agent world. We believe that the following lessons can be learnt, in particular, concerning roles:

– Roles must be distinguished in role types and role instances: role instances must be related to the concept of session of an interaction.
– The notion of role is useful not only for structuring institutions and organizations but for dealing with interaction among agents.
– The notion of affordance can be used to allow agents to interacts in different ways with different kind of agents.

In this paper, we show a different way of using `powerJava` exploiting roles to model communications where: the method call specifies the caller of the object, the state of the interaction is maintained, methods can be part of protocols, objects play roles in the interaction and method calls can be asynchronous as in agent protocols.

This proposal builds upon the experience that the authors gathered on the language `powerJava` [3,4,5,6], which is implemented by means of a precompiler. Basically `powerJava` shares the idea of gathering roles inside wider entities with languages like Object Teams [17] and Ceasar [20]. These languages emerge as refinements of aspect oriented languages aiming at resolving practical limitations of other languages. In contrast, our language starts from a conceptual modelling of roles and then it implements the model as language constructs. Differently than these languages we do not model aspects. The motivation is that we want to stick as much as possible to the Java language. However, aspects can be included in our conceptual model as well, under the idea that actions of an agent playing a role "count as" actions executed by the role itself. In the same way, the execution of methods of an object can give raise by advice weaving to the execution of a method of a role. On the other hand, these languages do not provide the notion of role casting we introduce in `powerJava`. Roles as double face interfaces have some similarities with Traits [22] and Mixins. However, they are distinguished because roles are used to extend instances and not classes. Finally, C# allows for multiple implementations of interfaces. None of the previous works, however, considers the fact that roles work as sessions of the interaction between objects.

By implementing agent like communication in an OO programming language, we gain in simplicity in the language development, importing concepts that have been developed by the agent community inside the Java language itself. This language is, undoubtedly, one of the most successful currently existing programming languages, which is also used to implement agents even though it does not supply specific features for doing it. The language extension that we propose is a step towards the overcoming of these limits.

At the same time, introducing theoretically attractive agent concepts in a widely used language can contribute to the success of the Autonomous Agents and Multiagent Systems research in other fields. Developers not interested in the complexity of agent systems can anyway benefit from the advances in this area by using simple and concrete constructs in a traditional programming language.

Future work concerns making explicit the notion of state of a protocol so to make it transparent to the programmer and allow to define the same method with different meanings in each state. Finally, the integration of centralized and decentralized approaches to coordination among roles (drawings (c) and (d) of Figure 1) must be studied.

References

1. A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An object data model with roles. In *Procs. of VLDB'93*, pages 39–51, 1993.
2. F. Arbab. Abstract behavior types: A foundation model for components and their composition. In *Formal Methods for Components and Objects, LNCS 2852*, pages 33–70. Springer Verlag, Berlin, 2003.
3. M. Baldoni, G. Boella, and L. van der Torre. Roles as a coordination construct: Introducing powerJava. In *Procs. of MTCoord'05 workshop at COORDINATION'05*, 2005.
4. M. Baldoni, G. Boella, and L. van der Torre. Bridging agent theory and object orientation: Importing social roles in object oriented languages. In *Post-Procs. of PROMAS'05 workshop at AAMAS'05*, volume 3862 of LNCS, pages 57-75, Springer, 2006.

5. M. Baldoni, G. Boella, and L. van der Torre. Powerjava: ontologically founded roles in object oriented programming language. In *Proc. of 21st ACM Symposium on Applied Computing, SAC 2006, Special Track on Object-Oriented Programming Languages and Systems (OOPS 2006)*, pages 1414-1418, Dijon, France, April 2006. ACM.
6. M. Baldoni, G. Boella, and L. van der Torre. Interaction among Objects via Roles – Sessions and Affordances in Java. In *Proc. of the 4th International Conference on Principles and Practices of Programming In Java (PPPJ 2006)*, pages 188-193, Mannheim, Germany, 2006).
7. B. Bauer, J.P. Muller, and J. Odell. Agent UML: A formalism for specifying multiagent software systems. *Int. Journal of Software Engineering and Knowledge Engineering*, 11(3):207–230, 2001.
8. F. Bellifemine, A. Poggi, and G. Rimassa. Developing multi-agent systems with a FIPA-compliant agent framework. *Software - Practice And Experience*, 31(2):103–128, 2001.
9. G. Boella and L. van der Torre. Attributing mental attitudes to roles: The agent metaphor applied to organizational design. In *Procs. of ICEC'04*. IEEE Press, 2004.
10. G. Boella and L. van der Torre. A game theoretic approach to contracts in multiagent systems. *IEEE Transactions on Systems, Man and Cybernetics - Part C*, 2006.
11. G. Boella and L. van der Torre. Security policies for sharing knowledge in virtual communities. *IEEE Transactions on Systems, Man and Cybernetics - Part A*, 2006.
12. M. Dahchour, A. Pirotte, and E. Zimanyi. A generic role model for dynamic objects. In *Procs. of CAiSE'02*, volume 2348 of *LNCS*, pages 643–658. Springer, 2002.
13. M. Dastani, V. Dignum, and F. Dignum. Role-assignment in open agent societies. In *Procs. of AAMAS'03*, pages 489–496, New York (NJ), 2003. ACM Press.
14. J. Ferber, O. Gutknecht, and F. Michel. From agents to organizations: an organizational view of multiagent systems. In *LNCS n. 2935: Procs. of AOSE'03*, pages 214–230. Springer Verlag, 2003.
15. J. Gibson. *The Ecological Approach to Visual Perception.* Lawrence Erlabum Associates, New Jersey, 1979.
16. N. Guarino and C. Welty. Evaluating ontological decisions with ontoclean. *Communications of ACM*, 45(2):61–65, 2002.
17. S. Herrmann. Object teams: Improving modularity for crosscutting collaborations. In *Procs. of Net.ObjectDays*, 2002.
18. T. Juan and L. Sterling. Achieving dynamic interfaces with agents concepts. In *Procs. of AAMAS'04*, 2004.
19. C. Masolo, L. Vieu, E. Bottazzi, C. Catenacci, R. Ferrario, A. Gangemi, and N. Guarino. Social roles and their descriptions. In *Procs. of KR'04*, pages 267–277. AAAI Press, 2004.
20. M. Mezini and K. Ostermann. Conquering aspects with caesar. In *Procs. of the 2nd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 90–100. ACM Press, 2004.
21. M.P. Papazoglou and B.J. Kramer. A database model for object dynamics. *The VLDB Journal*, 6(2):73–96, 1997.
22. N. Scharli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In Springer Verlag, editor, *LNCS, vol. 2743: Procs. of ECOOP'03*, pages 248–274, Berlin, 2003.

23. F. Steimann. On the representation of roles in object-oriented and conceptual modelling. *Data and Knowledge Engineering*, 35:83–848, 2000.
24. M. Winikoff. JACK - intelligent agents: An industrial strength platform. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming*, pages 175–193. Springer Verlag, Berlin, 2005.
25. M. J. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2):115–152, 1995.
26. F. Zambonelli, N.R. Jennings, and M. Wooldridge. Developing multiagent systems: The Gaia methodology. *IEEE Transactions of Software Engineering and Methodology*, 12(3):317–370, 2003.