Roles as a Coordination Construct: Introducing powerJava

Matteo Baldoni [1] Guido Boella [2]

Dipartimento di Informatica
Università degli Studi di Torino (Italy)

Leendert van der Torre [3]

CWI Amsterdam and Delft University of Technology (The Netherlands)

Abstract

In this paper we apply the role metaphor to coordination. Roles are used in sociology as a way to structure organizations and to coordinate their behavior. In our model, the features of roles are their dependence on an institution, and the powers they assign to players of roles. The institution represents an environment where the components interact with each other by using the powers attributed to them by the roles they play, even when they do not know each other. The interaction between a component playing a role and the role is performed via interfaces stating the requirements to play a role, and which powers are attributed by roles. Roles encapsulate their players' capabilities to interact with the institution and with the other roles, thus achieving *separation of concerns* between computation and coordination. The institution acts as a coordinator which manages the interactions among components by acting on the roles they play, thus achieving a form of *exogenous coordination*. As an example, we introduce the role construct in the Java programming language, providing a precompiler for it. In order to better explain the proposal, we show how to use the role construct as a coordination means by applying it to a dining philosophers problem extended with dynamic reconfiguration.

1 Introduction

Coordination, according to Malone and Crowston [16], is managing dependencies among independent activities. Coordination models and languages

[1] Email: baldoni@di.unito.it
[2] Email: guido@di.unito.it
[3] Email: torre@cwi.nl

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

all aim at providing frameworks for enhancing modularity, reuse of existing components, portability and language interoperability.

Papadopoulos and Arbab [23] distinguish two different approaches to coordination: the *data-driven* approach and the *control-driven* one. The difference rests in who drives the evolution of computation. In the former approach computation evolves driven by the data involved in the coordination, while in the latter one processes evolve according to events following state changes. Control-driven languages allow to achieve a more complete separation – even at the syntactic level – between coordination and computational concerns, which are dealt with by different processes. The state of the computation can, thus, be defined by the coordinated patterns that the processes involved in the computation adhere to. Moreover, control-driven approaches allow to treat the computational parts as black-boxes with clearly defined input/output interfaces. Finally, control-driven approaches allow a dynamic reconfiguration of the system, since the components specifying initial and evolving configurations are separated from the ones performing the actual computation.

The various coordination models and languages rely on distinct metaphors, like the shared dataspace, the blackboard model, the actor model, the chemical model, the channel model, *etc.* One reason for the diversity of metaphors is that coordination is an emerging area with an interdisciplinary focus going from economics to operational research, from organization theory to biology.

One basic metaphor in social theory and organization theory is the role metaphor. Roles are often defined as descriptions of expected behavior, and are used in organization theory to *distribute responsibilities*, *obligations* and *rights* among the agents working in an organization and, above all, to *distribute institutional powers* among them [22]. For example, an agent in the role of director of an organization has not only the right but also the power to sign buy orders on the behalf of the organization itself. Moreover, the agent playing the role of director has the power to commit the director to new responsibilities, as well as the power to commit to new duties the other members of the organization by ordering them. Finally, players of roles, exercising their powers, can change the structure itself of the organizations, by, e.g., merging departments, introducing new roles, hiring new employees, *etc.* Thus, roles, as entities endowed with powers, are used as a means to coordinate the behavior of an organization.

The research question of this paper is: "*How to introduce and use the role metaphor in control-driven coordination?*" To answer this question, our methodology is to start from our work on conceptual modelling [9], ontological analysis [7] and social reality of multiagent systems [8] to develop a new view on roles. Then, based on this model of roles we introduce a new role programming construct in a real programming language like Java; to prove its feasibility we translate it to pure Java by means of a precompilation phase. Finally, we show how the new language can be used for coordination purposes since it emphasises the separation of interactional aspects from the core

behavior of a class and it allows the exogenous coordination of components.

It is beyond the scope of this paper to provide a formal semantics of the new constructs or to define the associated type theory. Moreover we do not address in this paper other issues related to roles like the problem of method delegation or of roles playing other roles, but we leave them for future work.

The structure of this paper is as follows. In Section 2 we describe our definition of roles. In Section 3 we discuss how coordination can benefit from our definition of roles. In Section 4 we introduce the new language `powerJava` and in Section 5 we use it to model the dining philosophers example. Conclusion ends the paper.

2 Properties of roles

The characteristic features of roles in our model are their foundation, their definitional dependence from the institution they belong to, and the powers attributed to the role by the institution. To understand these issues we propose an example. Consider the roles student and teacher. A student and a teacher are always a student and a teacher of some school. Without the school the roles do not exist anymore: e.g., if the school goes bankrupt, the actors (persons) of the roles cannot be called teachers and students anymore. The institution (the school) also specifies the properties of the student, which extend the properties of the person playing the role of student: the school specifies its enrolment number, its email address, its scores at past examinations, but also how the student can behave. For example, the student can try an exam by submitting some written examination. A student can make the teacher evaluate its examination and register the mark because the school defines both the student role and the teacher's role: the school specifies how an examination is evaluated by a teacher, and maintains the official records of the examinations. Otherwise the student could not have any effect on the teacher. In defining such actions the school *empowers* the person who is playing the role of student: without being a student the person has no possibility to give an examination and to make the teacher evaluate it.

The above example highlights the following properties that roles have in our model of *normative* multiagent systems [9,7,8]:

Foundation: an instance of role must always be associated with an instance of the institution it belongs to (see Guarino and Welty [12]), 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.* [18], 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.

Finally, following Steimann [24]'s analysis of roles, in our approach a role can be played by different kinds of actors. For example, the role of customer can be played by instances both of person and of organization, i.e., two classes which do not have a common superclass. The role must specify how to deal with the different properties of the possible actors. This requirement is in line with UML, which relates roles and interfaces as partial descriptions of behavior. This requirement compels to avoid modelling roles as dynamic specializations as, e.g., [2,11] do. If customer were a subclass of person, it could not be at the same time a subclass of organization, since person and organization are disjoint classes. In the same way, person and organization cannot be subclass of customer, since a person can be a person without ever becoming a customer [12].

3 Roles and coordination

According to Arbab [3], extending the traditional Object Oriented model (OO) towards coordination of components presents some difficulties. They are related to the underlying notion of abstract data type with its idea of providing a set of operations in its interface while encapsulating data structures and the implementation of operations.

Components in OO are often seen as fortified collections of classes and objects with their interfaces. As a consequence the interactions among and the composition of components must use the same mechanisms as in the interaction among objects. The problem is that the method invocation semantics of the message passing metaphor in OO requires a very tight coupling between the caller and the callee objects. Amongst other things, the caller of a method of another object must know how to find this object and the syntax and semantics of the method. So, a component, to use another component, has to know it in advance, so that it can transfer control to it. The reason is that the operational interface of abstract data types induces an asymmetrical semantic dependency of consumers of operations on providers of operations: a consumer makes the decision on what operation to perform and it relies on the provider to carry out the operation. This reduces the "pluggability" of pre-existing components in a new system. Moreover, in inter-components interaction, method invocation does not allow to reach a minimum level of "control from the outside" of the participating components.

To resolve this problem, a different view of inter-component communication is advocated: *untargeted passive messages* exchanged with the environment that carry no control information in that they do not imply method invocation. For example, messages can be exchanged through channels. In contrast with targeted messages of method invocation, with untargeted messages the sender is not required to know who is the receiver of a message. In this way, a third party is given the possibility to set up the interaction

between the sender and a receiver of his choice. We believe that the notion of role we propose in this paper can contribute to the solution of these problems in the inter-components interaction in OO. The role metaphor *isolates the interaction between components at the level of the roles they play.*

It is possible to draw a comparison between roles and the IWIM (Idealized Worker Idealized Manager) model [10,4,23]. Components playing roles are the workers carrying out the computation, while the institutions are the managers coordinating the other processes. The major difference is that institutions (managers) do not directly manipulate components (workers), but they coordinate them through the roles they play, which represent the state of the component inside the institution. Symmetrically, the components do not interact with other components, but they interact with the institution and with the other roles only through the powers offered by the roles they play. Roles give to their players the powers to interact with the other roles and the institutions. At the same time these powers, which are modelled as methods, are inside and defined by the institution, so components are not required to know their implementation, while they can be invoked on them when acting in a role.

By means of roles it is possible to connect the output of a component to the input of another component without requiring them to be aware of the connection, and to encapsulate the modalities of this connections, like concurrency synchronization. In this way we achieve the separation of concerns: components which act as the primary unit of computation in a system, and institutions which specify interaction and communication patterns between the components by means of roles.

Since powers are methods inside and defined by the institution they have the possibility to access both the institution and the other roles in it: hence, they can also reconfigure the interaction between the components playing the role. In the same way, the institution itself can modify these interconnections, thus achieving an exogenous coordination of the components composing the system. The implementation of the powers in the institution contains the information necessary to interact with the other components which play roles in the institution. At this level the interaction can be carried out by the standard intra component method invocation, since all the roles belong to the same component as the institution. Results of invocation can be then delivered to the calling component according to the preferred protocol, e.g., synchronous or asynchronous.

Finally, the interaction between components and their roles is relieved from the asymmetry of method invocation. Even if roles are object instances as any other, they represent a perspective on the object playing them. To invoke a method of a role, i.e., a power, the player of the role does not need a reference to the role instance. Rather, it needs only to specify which roles it is playing and in which institution and, then, it can invoke the method. The association between the player instance and the related role instance is managed in a

transparent way by the framework we propose in the next section.

In this paper we illustrate the proposed `powerJava` language by means of the dining philosophers problem as a running example. It can be seen as a case of coordination between two types of components: resources to be concurrently shared by consumers, and consumers of resources which periodically need the output of a specific pair of resources to perform their computation. Resources provide data which are feeded as input to the consumers. Consumers do not know which resources are feeding them and how the concurrent access to the resources is synchronized. We assume that resources and consumers do not have any knowledge about the participants to the dinner, how the participants are disposed around the dinner table and how the resources have to be shared. Finally, consumers should not care about how to join and leave the dinner: they just exercise powers offered by the roles they play. All these coordination issues are dealt with by the roles and by the institution roles belong to, which constitutes at the same time the manager coordinating the workers via roles and the environment where the interaction takes place. Hence, the description of the coordination part of the system is given in a transparent and exogenous way and the components are plugged without requiring any knowledge of it.

4 Introducing roles in Java

In this section we discuss how our definition of roles can be introduced in Object Oriented programming languages. Since, as discussed, the role-as-specialization approach is not possible, we define roles as instances which can be associated runtime to objects which can play those roles. However, the extension of an object to roles must be transparent to the programmer.

Since powers are a distinguishing feature of roles, we call our language for coordination `powerJava`. To introduce roles in `powerJava` we have to address the following issues:

(i) The construct defining roles.

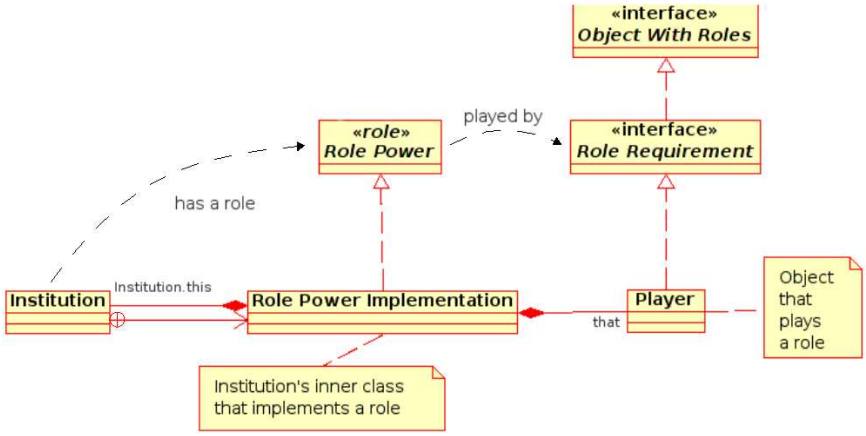

Fig. 1. Roles and Institutions.

(ii) The implementation of a role inside an institution according to its definition.

(iii) A way to invoke the powers, which a role has in an institution, from an object playing that role.

To achieve our goals, we need very limited modifications of the Java syntax (see Figure 3). In Figure 1 we summarize the overall model using a UML diagram. 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 distinct classes, so that roles can be specified independently of the particular class playing the role. This is a form of polymorphism. For example a role customer can be played both by a person and by an organization.

Our proposal is to define the role construct as a sort of "*double-sided*" interface which allows the connection of a player to an institution. The interface is double in that it specifies:

The methods that a class must offer for playing the role (requirements). In order to play a role, a class must offer some methods. These are specified in the role by an interface.

The methods offered to objects playing the role (powers). An object of a class, offering the required methods, plays the role: it is empowered with new methods as specified by this part of the role definition.

This "double face" pervades the life of a role: first, a role is defined with its requirements and powers, then its powers are implemented in a class which connects a role with a player satisfying its requirements, and, finally, the class implementing the role is instantiated passing to the constructor an instance of an object satisfying the requirements.

Requirements of a role in Java correspond to the notion of interface, specifying which methods must be defined in a class playing the role. As for interfaces, this mechanism of partial descriptions allows the polymorphism necessary for a role to be played by different classes.

The definition of a role (`roledef` in Figure 3) using the keyword `role` is similar to the definition of an interface: it is in fact the specification of the powers acquired by the role in the form of abstract methods signatures (`interfacebody`). The only difference is that the role definition does not allow even static variables to be declared, and it refers also to another interface, that in turn gives the requirements which an object, willing to play that role, must conform to. Such an interface is identified by the keyword `playedby`. In the example of Figure 4, the `role` definition `Philosopher` specifies the powers (`eat`, `think`, `start`, and `leaveTable`), whilst the *playedby* interface `PhilosopherReq` specifies its requirements (`putData` and `processData`).

This "double face" of a role definition captures the idea of Guillen-Scholten

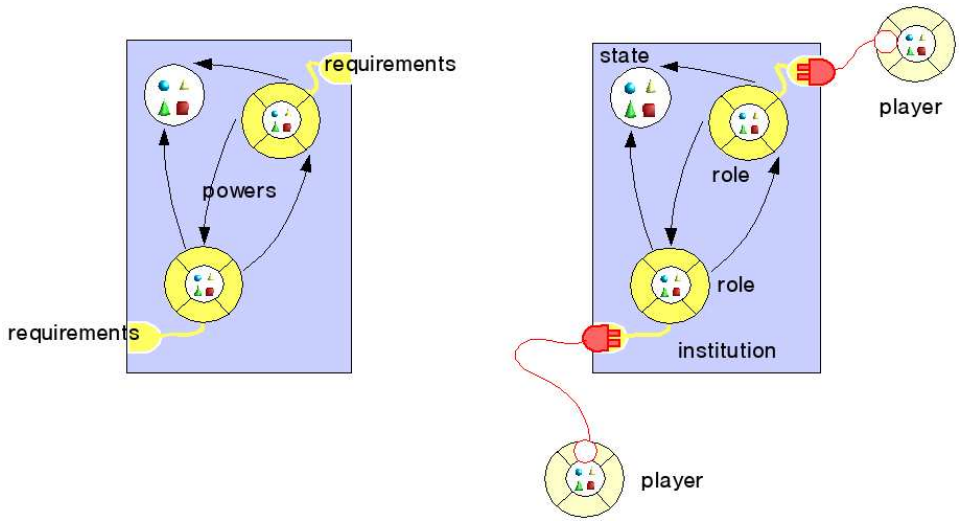

Fig. 2. The players will interact according to the acquired powers (they will follow the *protocol* implemented by the institution and its roles).

```
roledef ::= "role" identifier ["extends" identifier*]
   "playedby" identifier interfacebody

roleimplementation ::=
   [public | private | ...] [static] "class" identifier
   ["realizes" identifier] ["extends" identifier]
   ["implements" identifier*]  classbody

keyword ::= that | ...

rcast ::= (expr "." identifier) expr
```

Fig. 3. The extension of Java syntax in `powerJava`.

et al. [13] that the concept of interface for an object is different from the corresponding notion for a component. An interface of an object involves only a one-way flow of dependencies from the object providing a service to its clients. In contrast, the interface of a component involves a two way reciprocal interaction between the component and its environment. Analogously, roles in our model allow the usability of components by specifying two interfaces: the interface required by a component to plug in the system in a role and the interface specifying which services it can provide to the system once it plays its role. The difference is that powers are implemented in the role and not in the component, since the role encapsulates how the component interacts with the rest of the system. In this way, the component can offer services integrated in the system while being developed independently without requiring any knowledge of it (see Figure 2).

The implementation of the requirements is given inside the class of the object playing the role. The implementation of the powers must be necessarily

```
interface ChopstickReq {
  Object getData();
}

role Chopstick playedby ChopstickReq {
  Object use();
}

interface PhilosopherReq {
  void putData(Object input1, Object input2);
  void processData();
}

role Philosopher extends Runnable playedby PhilosopherReq {
  void eat();
  void think();
  void start();
  void leaveTable();
}
```

Fig. 4. Role definitions.

given in the definition of the class defining the institution of the role.

To implement roles inside an institution we extend the notion of Java inner class, by specifying with the new keyword `realizes` the name of the role definition that the class is implementing (see Figure 3). In `powerJava`, an inner class that realizes a role must implement the corresponding role definition in the very same way as a class implements an interface. For example, in Figure 5 and 6, the inner class `PhilosopherImpl` of the institution `Table` realizes the role `Philosopher`.

As a Java inner class, a role implementation has access to the private fields and methods of the outer class (in our case the institution) 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 (see, e.g, the method `eat` in Figure 6). 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. Moreover, since an inner class is a class, it can extend other classes (unless they implement roles), it can be an institution itself and thus, have, its own role implementations, *etc.*

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. See, for example, the implementation of method `think` in `PhilosopherImpl`. The value of `that` is initialized when the constructor of the role implementation is

```
class Table {

  java.util.ArrayList phils = new java.util.ArrayList();
  Table(PhilosopherReq[] philsReq, ChopstickReq[] chopsReq) {
    ...
    ChopstickImpl res = this.new ChopstickImpl(chopsReq[i]);
    ...
    // the implicit first parameters passes the player of the role
    ...
    PhilosopherImpl phil =
      this.new PhilosopherImpl(philsReq[i], res, ...);
    ...
    phils.add(phil);
    ...
  }
  public void addPhilosopher(PhilosopherReq philReq,
                             ChopstickReq chopReq) {
    // Add a philosopher with its chopstick.
  }
  public void startDinner() {
    // This starts the philosophers' threads
    for(int i=0; i<phils.size(); i++)
      ((Philosopher)phils.get(i)).start();
  }

  class ChopstickImpl realizes Chopstick {
    [...]
  }

  class PhilosopherImpl realizes Philosopher {
    [...]
  }

}
```

Fig. 5. The definition of an institution and its roles.

invoked. The referred object has the type defined by the role requirements.

All the constructors of all roles have an implicit first parameter which must be passed as value the player of the role.[4] 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: e.g., the constructor `PhilosopherImpl` has a first parameter of type `PhilosopherReq`. At the moment it is not possible to create an instance of a role without any player associated with but this could be object of further investigation.

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 instanti-

[4] The parameter is added by the precompiler of `powerJava`. See the website `http://www.powerjava.org` for details.

```
class Table {
  [...]
  class ChopstickImpl realizes Chopstick {
    Object owner = null;
    private synchronized void grab(Object f) {
      try {
        while(owner != null) wait();
      } catch(InterruptedException e) {}
      owner = f;
    }
    private synchronized void release(Object f) {
      if (f == owner) owner = null;
      notify();
    }
    public Object use() {
      return that.getData();
    }
  }
  class PhilosopherImpl realizes Philosopher {
    private ChopstickImpl left, right;
    private boolean done = false;
    // the implicit first parameter is added by the precompiler
    public PhilosopherImpl(ChopstickImpl l, ChopstickImpl r) {
      left = l; right = r;
    }
    public void eat() {
      left.grab(this); right.grab(this);
      that.putData(left.use());
      right.use());
      left.release(this); right.release(this);
    }
    public void think() {
      that.processData();
    }
    public void start() {
      Thread thread = new Thread(this);
      thread.start();
    }
    public void run() {
      while(!done) {
        eat();
        think();
      }
    }
    public void leaveTable() {
      // The current philosopher is removed.
    }
  }
}
```

Fig. 6. The definition of an institution and its roles (continued).

```
class Consumer implements PhilosopherReq {
  // Fields and methods of Consumer ...
  public void putData(Object input1, Object input2) {
    // The consumer gets the data produced by the resources
  }
  public void processData() {
    // The consumer uses the data
  }
}

class Resource implements ChopstickReq {
  // Fields and methods of Resource ...
  public Object getData() {
    // The resource returns the data in order to send it to the consumer
    return ... ;
  }
}
```

Fig. 7. Classes playing roles.

ate. This is like it is done in Java for inner class instance creation. Note that it is not possible to directly instantiate a role: first of all, roles are used like interfaces, secondly, roles can be implemented in different ways in the same institution. For example, the class `Table` could contain different implementations the role `Philosopher`. In Figure 5, a philosopher is created by `this.new PhilosopherImpl(philsReq[i],...)`, where the parameter `philsReq[i]` is an object implementing the requirements `PhilosopherReq`.

In order for an object to play a role it is sufficient that it conforms to the role requirements. Since the role requirements are implemented as a Java interface, it is sufficient that the class of the object implements the methods of such an interface. In Figure 7, the class `Consumer` can play the role `Philosopher`, because it conforms to the interface `PhilospherReq` by implementing its methods.

Differently than other objects, role instances do not exist by themselves and are always associated to their players: when it is necessary to invoke a method of a philosopher it is sufficient to have a reference to its player object; it is not necessary to know which is the role instance played by the object. Methods can be invoked from the players, given that the player is seen in its role (e.g., `PhilosopherImpl`). To do this, we use the Java idea of casting with a difference: the object is not casted to a type. Casting is done in `powerJava` by casting the player of the role to the role implementation we want to refer to. It is not possible, however, to cast to a role (e.g., `Philosopher` since it could have been implemented by different role implementations in the same institution: hence, the casting would have been ambiguous.

However, since roles do not exist outside an instance of the institution defining them, in order to specify a role, it is necessary to specify the institution it belongs to. In the syntax of `powerJava` the structure of a role casting is captured by `rcast` in Figure 3. For instance, in Figure 8

```
class Test {
  public static void main(String[] args) {
    Consumer[] consumers = new Consumer[] {
      new Consumer(), new Consumer(), ...
    };
    Resource[] resources = new Resource[] {
      new Resource(), new Resource(), ...
    };
    Table table = new Table(consumers, resources);
    table.startDinner();
    // ...
    Consumer newConsumer = new Consumer();
    Resource newResource = new Resource();
    table.addPhilosopher(newConsumer, newResource);
    // Role cast
    ((table.PhilosopherImpl)newConsumer).start();
    ((table.PhilosopherImpl)newConsumer).eat();
    ((table.PhilosopherImpl)newConsumer).think();
    // ...
    ((table.PhilosopherImpl)newConsumer).leaveTable();
  }
}
```

Fig. 8. How the `main` sets up the dinner.

```
((table.PhilosopherImpl) newConsumer).eat()
```

takes the `Philosopher` role played by the `newConsumer` in the institution `table` and invokes on it methods like `eat` which are powers of the role. In this way, the `Consumer` component is relieved from the burden of having an explicit reference to the role instance it is playing: the role is directly accessed via its player.

We call this *role casting*. Type casting in Java allows to see the same object under different perspectives while maintaining the same structure and state. In contrast, role casting views an object as having a different, even if related, state and different behaviors. This is because, it conceals a *delegation* mechanism: the player instance hiddenly delegates the role instance the execution of the method. The delegated object can only act as allowed by the powers of the role, but it can access the state of the institution via its powers.

Finally `powerJava` allows the definition of roles which can be further articulated into other roles. For example, a school can be articulated in teaching classes (another social entity) which are, in turn, articulated into student roles. In this way, it is possible to create a hierarchy of managers and workers as suggested by the IWIM model [10,4,23]: at the bottom level are component workers which do no coordinate any other component, while at the upper levels there are components managing other components (either workers or other managers).

5 Example

In order to illustrate how roles can be used for coordination purposes, we use, as well as Arbab did [3], the dining philosophers example, presenting it in `powerJava`; we do this by introducing roles in the implementation that is proposed in the Java tutorial [26]. To fully explain the potential of roles we extend the dining philosophers introducing some reconfiguration issues. In particular, we assume that new components playing the role of philosophers and new resources playing the role of chopsticks can join the table, as well as that philosophers can leave at any moment. The example is modelled by means of five kinds of objects:

(i) The dinner `Table`, which constitutes the environment where the components interact, and which coordinates them. The table is an institution which is organized in two types of roles: philosophers and chopsticks.

(ii) `Philosopher`s offer to the consumers playing them four powers: the method for eating after grabbing the chopsticks (`eat`), the method for thinking (`think`), the method for starting the dinner intermixing thinking and eating (`start`), and the method for leaving the table (`leaveTable`).

(iii) `Chopstick`s offer the resources playing them a method for being used by other components (`use`) to get the data from the resource playing the role.

(iv) The `Resources` are components, whose interface only offers a method for getting data from them (`getData`).

(v) The `Consumers` are components, which must offer a way to pass the data to them (`putData(Object input1, Object input2)`) and to perform a computation on them (`processData`).

The `Table` is the coordination environment maintaining an ordered list of the philosophers sitting at it with their chopsticks and the implementation of the two roles. Its constructor takes two arrays, one containing objects that can play the philosopher role, the other containing objects that can play the chopstick role (in our example, a `Consumer` and a `Resource` which implement respectively the `PhilosopherReq` and `ChopstickReq` interfaces). Using these objects, the table creates the instances of the required roles, puts the philosophers around the table and connects them with their chopsticks (see Figure 9).

The implementations of `Chopstick`s and `Philosopher`s encapsulate the fields relating them to each other (`left` and `right`) and the methods for accessing in a concurrent way to the shared resources: this information is hidden to the components playing them. However, their powers ensure that the components are made interact with each other in a way which prevents deadlocks and interferences.

In particular, the implementation `ChopstickImpl` of the role `Chopstick` implements not only the powers of that role, but also suitable private meth-

ods for grabbing (`grab`) and releasing (`release`) chopsticks in a synchronized way. These methods block and reactivate processes accessing chopsticks which are already used (`owner`) by a philosopher or by the table. Note that these methods are private, but, since they are defined in an inner class, they are still visible to the other roles (indeed, they are used in the implementation of the powers of a `Philosopher`, see, e.g., the method `eat`).

The implementation `PhilosopherImpl` of the role `Philosopher` implements the powers `eat`, `think`, `start`, and `leaveTable`. The method `eat` allows a consumer, playing the role `Philosopher`, to participate to the dinner. The method first gets the chopsticks, it takes the expected data by using the chopsticks (`left.use()` and `right.use()`), and then invokes `putData` with the obtained information as parameters, thus passing the results to the consumer. Observe that the consumer will get an outcome without being aware of the information source (a sort of channel of information). Finally, it releases the chopsticks. Note that this method must be defined inside the role and not in its player since the player component is not required to know how to get, use, and release the chopstick; the component uses the data while the data source management is encapsulated in the role. In other words, the two classes representing the players of the roles need only implementing the methods specified by the interfaces as their requirements. These classes do not make any reference to each other, nor to the coordination structure of the environment they interact into: all these aspects are dealt with by the

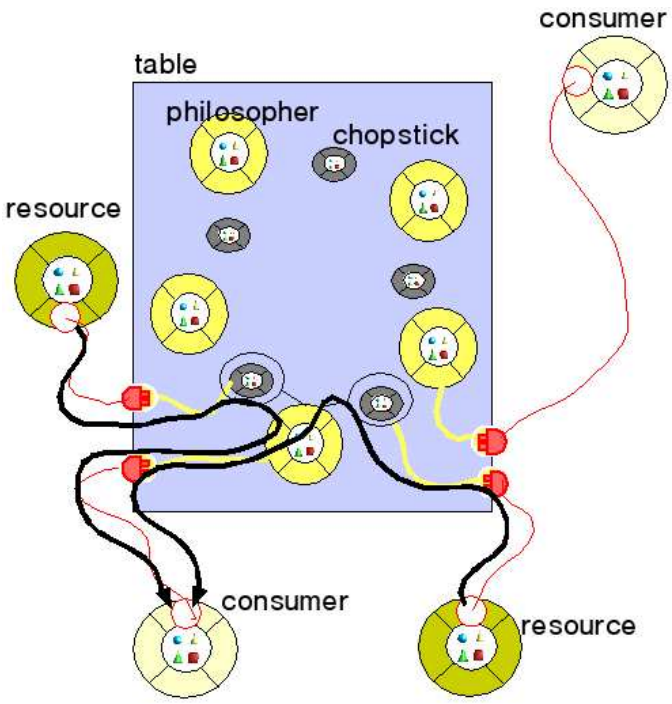

Fig. 9. The table is the coordination environment, coordination between resources and consumers is carried on through the roles.

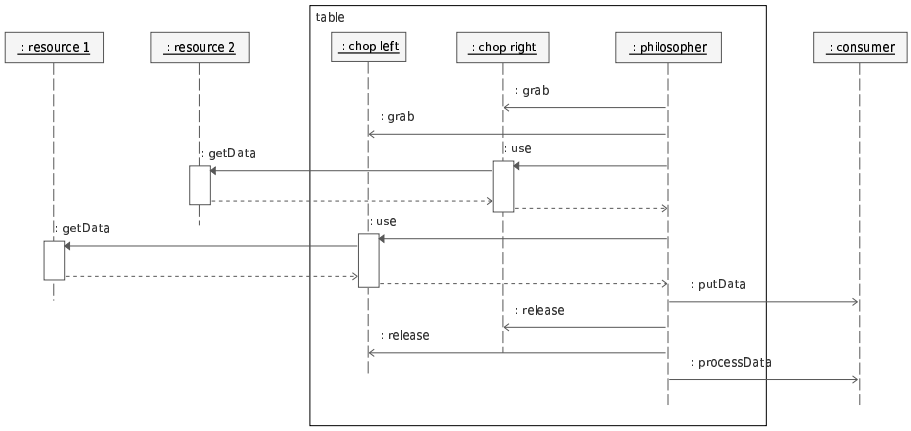

Fig. 10. Roles, institution and coordination.

```
public void addPhilosopher(PhilosopherReq philReq,
                           ChopstickReq chopReq) {
  // ...
  PhilosopherImpl phil0 = (PhilosopherImpl)phils.get(0);
  ChopstickImpl chop = phil0.left;
  chop.grab(this);
  phils.add(new PhilosopherImpl(philReq, phil0.left,
                                new ChopstickImpl(chopReq)));
  phil0.left = ((PhilosopherImpl) phils.get(phils.size()-1)).left;
  chop.release(this);
  // ...
}
```

Fig. 11. Sketch of reconfiguring the dinner table by adding a philosopher.

roles they play thanks to the precompiling phase (see the underlying model in Figure 1). Figure 10 visualizes the sequence diagram of the example.

The class `Resource` implements the interface `ChopstickReq`, which contains the method for getting the data: `getData`. The class `Consumer` implements the interface `PhilosopherReq`, which allows it to play the role of `Philosopher`. It implements `putData`, to read data produced by the information sources and the method to process them (`processData`). We implemented here synchronous communication for simplicity, but nothing prevents implementing an asynchronous version, e.g., by adding a buffer to the role's state. A `Consumer` accesses its role of `PhilosopherImpl` by casting itself to that role in the table, e.g.: `((table.PhilosopherImpl)newConsumer).start()` in Figure 8.

Our philosophers are able to reconfigure the system using the method `addPhilosopher(PhilosopherReq philReq, ChopstickReq chopReq)` of `Table` and the method `leaveTable()` of `Philosopher`. `addPhilosopher` has two objects as parameters, one able to play the `Philosopher` (Figure 11) role and the other one able to play the `Chopstick` role and adds them at the end of the table. To do so, essentially the table grab the chopstick of the first

philosopher (not to use them, but to prevent others from using them), creates a new philosopher and a new chopstick role and adjusts the connections between the philosophers and their chopsticks. Finally it releases the reserved chopstick (which is now distributed among the remaining philosophers). In contrast with the former method, `leaveTable` (Figure 12) is the method directly invoked on a `Consumer` in its role of `Philosopher` (see `main` in Figure 8) in order to leave the table.

The `main` in Figure 8 simply constructs the required arrays of `Consumers` and `Resources`, invokes the constructor of the `Table` which creates the coordination environment, and starts the dinner of the philosophers. Finally, it adds a further philosopher invoking the method `addPhilosopher` on the table. The computation of the corresponding role played by the new consumer is started after the proper role cast, and, after a while, it leaves the table by invoking `leaveTable` on its role:

```
((table.PhilosopherImpl)newConsumer).leaveTable()
```

In summary, the interaction between the `Consumer` and `Resource` components is realized without requesting them to know each other to be able to invoke their methods, to know the configuration they are involved into, and the connections among the components (i.e., which the chopsticks of each philosopher are and how they are distributed around the table), and how to reconfigure the system. Roles are the connectors which relate the components to the system and among each other. The interconnections among the components can be changed by the institution or by the roles themselves, while the components remain unaware of the reconfiguration. In our model we can replace any component with another version of it without having to change any other component or the coordination scheme of the system. Finally, the coordination scheme of the system that is independent of the computation parts of components can also be updated without the necessity to change the components of the system.

```
public void leaveTable() {
  // ...
  int i = phils.lastIndexOf(this);
  PhilosopherImpl next =
    (PhilosopherImpl)phils.get( (i+1) % (phils.size() - 1));
  right.grab(this);
  phils.remove(i);
  next.left=this.left;
  right.release(this);
  done = true;
  // ...
}
```

Fig. 12. Sketch of how a philosopher leaves the table.

6 Conclusion and related work

In this paper we show how to introduce the role metaphor in Object Oriented languages and how to use it in control-driven coordination. Component objects interact with each other only via the roles they play in an institution which constitutes the interaction environment, thus achieving the separation of concerns, and exogenously coordinates the behavior of the system. We show how Java can be extended with roles, and how the resulting language `powerJava` can be used to model the dining philosophers problem enriched with reconfiguration issues. Many characteristics of this proposal of role definition have origin in the multi-agent systems research area. In [5] details about this relationship are discussed.

As Guillen-Scholten *et al.* [13] do for channels, we extend Java to show how it is possible to model components in a self-contained way in a widely used Object Oriented language. We implement a precompiler to pure Java for the language `powerJava`, using the tool javaCC, provided by Sun Microsystems [1]. The precompiler together with the complete example, more information on the language and on its translation to Java can be found at `http://www.powerjava.org`. Briefly, each role specification is translated into couple of interfaces, while role implementations are translated into inner classes, whose constructors are extended appropriately. Players are modified in order to manage a list of roles and role casting is translated into an instruction that allows finding the corresponding roles inside these lists (using the name of the role and the instance of the institution), then delegating this object for the execution of the power.

Roles definition has a strong relationship with the specification of a *communication protocol* [17,15]. Indeed, roles (as entities endowed with powers) are a means to coordinate the behavior within an organization. Playing a role means acquiring specific powers (given by the organization); the players interact according to the acquired powers. In other words, the players follow the protocol implemented by the institution and its roles in order to interact with each other. The institution itself is an abstraction of the protocol. Protocols as institutions can be collected together in order to constitute a library of protocols (coordination patterns). The designers must verify and prove properties of their coordination pattern just once [27], specifying which requirements should be satisfied in order to play the role involved in the protocol (to be "plugged" in the pattern, see Figure 2).

The notion of role used in `powerJava` has some similarities with the notion of agent coordination context developed by Omicini [21]. Agent coordination contexts are based on the control room metaphor. According to this metaphor, an agent entering a new environment is assigned its own control room, which is the only way in which it can perceive the environment, as well as the only way in which it can interact. In our terminology, an environment is an institution and a control room defines the powers of an agent working in an environment.

Our approach shares the idea of gathering roles inside wider entities with languages like Object Teams [14] and Ceasar [19]. 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`. By using role casting, it is possible to play a role at any point of the code and not only inside specific methods, as instead in [14]. Therefore, flexibility is increased.

Our notion of role, as a double-sided interface, bears some similarities with Traits [20] and Mixins. However, the latter are distinguished because roles are used to extend instances and not classes, with a few exceptions, e.g., [6]. Another difference of our approach, with respect to others that we have mentioned and, in particular, [14], stands in the use of *interfaces*. There is a wide agreement that variables should be declared with interfaces as their types, not classes. In fact, in this way a greater modularity is obtained. This is particularly important in the development of frameworks where classes must fit in at various points in the design and in component-based programming [25].

References

[1] *Java compiler compiler [tm] (javaCC [tm]) - the java parser generator*, Sun Microsystems, https://javacc.dev.java.net/.

[2] Albano, A., R. Bergamini, G. Ghelli and R. Orsini, *An object data model with roles*, in: *Procs. of VLDB'93*, 1993, pp. 39–51.

[3] Arbab, F., *Abstract behavior types: A foundation model for components and their composition*, in: *Formal Methods for Components and Objects, LNCS 2852*, Springer Verlag, Berlin, 2003 pp. 33–70.

[4] Arbab, F., *The IWIM Model for Coordination of Concurrent Activities*, in: *Procs. of Coordination* (1996), pp. 34–56.

[5] Baldoni, M., G. Boella and L. van der Torre, *Bridging Agent Theory and Object Orientation: Importing Social Roles in Object Oriented Languages*, in: R. H. Bordini, M. Dastani, J. DIx and A. Seghrouchni, editors, *Proc. of the International Workshop on Programming Multi-Agent Systems, ProMAS 2005*, Utrecht, the Netherlands, 2005, to appear.

[6] Bettini, L., V. Bono and S. Likavec, *A core calculus of mixin-based incomplete objects*, in: *Procs. of FOOL Workshop*, 2002.

[7] Boella, G. and L. van der Torre, *An agent oriented ontology of social reality*, in: *Procs. of FOIS'04*, Torino, 2004, pp. 199–209.

[8] Boella, G. and L. van der Torre, *Attributing mental attitudes to roles: The agent metaphor applied to organizational design*, in: *Procs. of ICEC'04* (2004), pp. 130–137.

[9] Boella, G. and L. van der Torre, *Regulative and constitutive norms in normative multiagent systems*, in: *Procs. of KR'04* (2004), pp. 255–265.

[10] Ciancarini, P. and Hankin, C., editors, *iCoordination Languages and Models, First International Conference, Coordination'96* (1996), Cesena, Italy.

[11] Gottlob, G., M. Schrefl and B. Rock, *Extending object-oriented systems with roles*, ACM Transactions on Information Systems **14(3)** (1996), pp. 268 – 296.

[12] Guarino, N. and C. Welty, *Evaluating ontological decisions with ontoclean*, Communications of ACM **45(2)** (2002), pp. 61–65.

[13] Guillen-Scholten, J., F. Arbab, F. de Boer and M. Bonsangue, *A channel based coordination model for components*, ENTCS **68(3)** (2003).

[14] Herrmann, S., *Object teams: Improving modularity for crosscutting collaborations*, in: *Procs. of Net.ObjectDays*, 2002.

[15] Huget, M. P. and J. Koning, *Interaction Protocol Engineering*, in: H. Huget, editor, *Communication in Multiagent Systems*, LNAI **2650** (2003), pp. 179–193.

[16] Malone, T. and K. Crowston, *The interdisciplinary study of coordination*, ACM Computing Surveys **26** (1994), pp. 87–119.

[17] Mamdani, A. and J. Pitt, *Communication protocols in multi-agent systems: A development method and reference architecture*, in: *Issues in Agent Communication*, LNCS **1916** (2000), pp. 160–177.

[18] Masolo, C., L. Vieu, E. Bottazzi, C. Catenacci, R. Ferrario, A. Gangemi and N. Guarino, *Social roles and their descriptions*, in: *Procs. of KR'04*, 2004.

[19] Mezini, M. and K.Ostermann, *Conquering aspects with caesar*, in: *Procs. of the 2nd International Conference on Aspect-Oriented Software Development (AOSD)* (2004), pp. 90–100.

[20] N. Scharli, O. N., S. Ducasse and A. Black, *Traits: Composable units of behavior*, in: S. Verlag, editor, *LNCS, vol. 2743: Procs. of ECOOP'03*, Berlin, 2003, pp. 248–274.

[21] Omicini, A., *Towards a notion of agent coordination context*, in: D. C. Marinescu and C. Lee, editors, *Process Coordination and Ubiquitous Computing*, CRC Press, 2002 pp. 187–200.

[22] Ouchi, W., *A conceptual framework for the design of organizational control mechanisms*, Management Science **25(9)** (1979), pp. 833–848.

[23] Papadopoulos, G. and F. Arbab, *Coordination models and languages*, Advances in Computers **46** (1998), pp. 329–400.

[24] Steimann, F., *A radical revision of UML's role concept*, in: *Procs. of UML2000*, 2000, pp. 194–209.

[25] Steimann, F., W. Siberski and T. Kühne, *Towards the Systematic Use of Interface in Java Programming*, in: *Proc. of 2nd Int. Conference in Java Programming*, 2003, pp. 13–17.

[26] Sun Microsystems, "The Java Tutorial," Http://java.sun.com/docs/books/tutorial/.

[27] Walton, C., *Model checking agent dialogues*, in: *Int. Workshop on Declarative Agent Languages and Technology*, 2004, pp. 156–171.