Programmable Coordination Media

Enrico Denti, Antonio Natali, Andrea Omicini

LIA - DEIS - Università di Bologna
Viale Risorgimento, 2 – 40136, Bologna (Italy)
Ph.: +39 51 6443087 - Fax: +39 51 6443073
mailto:{edenti,anatali,aomicini}@deis.unibo.it
http://www-lia.deis.unibo.it/Staff/

Abstract. The design, development and maintenance of multi-component software systems often suffer from the lack of suitable coordination abstractions. The aim of this paper is to show the benefits of coordination models based on global communication abstractions whose behaviour is not fixed, but is extensible so as to accomplish the intended behaviour of the whole system. Accordingly, we propose the notion of *programmable coordination medium* as an abstraction provided by the coordination model around which the global behaviour of a coordination architecture can be designed. As an example, we show how a Linda-based approach can be empowered by exploiting the notion of programmable tuple space, as supported by the $\mathcal{ACLT}$ coordination model.

Keywords: Coordination Models, Programmable Coordination Media, Reactions, Tuple Spaces, Multi-Agent Systems

1 Introduction

Component technology is radically altering the way software systems are designed: for instance, today typical WWW servers are built by simply assembling and extending existing components like HTTP servers, DB managers and e-mail applications. Generally speaking, most current multi-component systems are designed by mapping each service to be provided into a single component, often adding new components as soon as new functionalities are needed: the whole system is conceived to be nothing more than the sum of its parts.

Correspondingly, current models for component interaction (like COM/OLE-ActiveX [3], CORBA [13], CGI interface protocol) are usually based on message passing and point-to-point communication, thus providing no real coordination. When designing composite software system, these models lack the adequate abstractions needed to achieve an intelligent and flexible behaviour of the overall system. Whenever a change to the *global* system behaviour is needed, for instance, they require *local* modifications to (possibly all) the components of the system. This is why these systems are usually designed around special components (monitors, coordinators) embodying the core of the global behaviour of the overall system, so that a change to one such a component affects the whole system.

Coordination models like the blackboard based ones, instead, provide for a global, explicit communication abstraction (a blackboard, a tuple space), around which a multi-component system is naturally to be built. However, how these coordination media work is usually set once and for all by the coordination model, and cannot be modified or extended according to the intended overall system's behaviour. Thus, coordination entities have to take charge of the interaction protocol, and cannot abstract from the coordination policy.

The main aim of this paper is to discuss the benefits of designing multi-component software systems around a global communication abstraction whose behaviour can be extended and tailored to the system needs. To this end, we suggest the notion of *programmable coordination medium* as a kernel for coordination models whose flexibility and expressive power lies in the extensibility of the coordination medium itself. In particular, we propose that extensibility be the result of embodying computational properties typically in charge of the components into the communication abstraction.

As a case study, Section 2 shows how a shared communication device à la Linda can work as the core of a flexible coordination architecture in the Linda-based $\mathcal{ACLT}$ [14] coordination model. $\mathcal{ACLT}$ tuple spaces are enhanced so as to be *reactive* to communication events, rather than to communication state changes only. Reactions to communication events can be defined through a logic-based specification language, making $\mathcal{ACLT}$ tuple spaces *programmable*.

As a typical example of a coordination architecture designed around a programmable coordination medium, we discuss a simple case of a multi-agent system based on the $\mathcal{ACLT}$ coordination model. As suggested in [8], it can be desirable to design the observable behaviour of agents of a multi-agent system according to a quite abstract and straightforward pattern, while most of the global properties of the system are naturally placed in the global communication abstraction. In this way, a change to the coordination policy requires no modifications to the interaction protocol of a single component of the system. Correspondingly, Section 3 shows how a multi-agent architecture can be built around a programmable coordination medium by extending the behaviour of $\mathcal{ACLT}$ tuple spaces through reaction programming.

2 Programmable tuple spaces

In order to show the effectiveness of an approach to coordination based on the notion of *programmable coordination medium*, this Section discusses some peculiar aspects of the Linda-based $\mathcal{ACLT}$ coordination model [14]. Linda [10] introduces the notion of *generative communication* and promotes the separation between the computation model and the coordination model [12], based on a shared memory communication abstraction called *tuple space*. In this paper, we will take the Linda coordination model as known, as well as its most common extensions (like the predicate, non-blocking `in_noblock` and `rd_noblock` primitives).

The $\mathcal{ACLT}$ coordination model (first presented in [14]) extends the basic Linda model with the notions of *logic tuple space* (see also [4,5]), *multiple tuple spaces*[1] [11], and *reactive tuple space* [7]. In the $\mathcal{ACLT}$ model, communication takes place through a multiplicity of named logic tuple spaces, which are collections of first-order unitary clauses, uniquely identified by a ground term. In particular, a logic tuple space may be given a twofold interpretation, either as a simple communication device, or or as a knowledge repository. According to the latter reading, a logic tuple space can be used as a logic theory, where deductive activities over the communication state can be performed. For this purpose, $\mathcal{ACLT}$ provides for a family of *demo* primitives, along with a coherent notion of logic consequence in a time-dependent environment [14].

What is relevant here is the $\mathcal{ACLT}$ notion of *programmable tuple space*. The first idea is to raise observability at the system level from tuples to (communication) operations over tuples. Correspondingly, $\mathcal{ACLT}$ tuple spaces are *reactive*, since they are provided with the capability to react to *communication events* rather than just to the *communication state changes* only, as in standard Linda [7].

A simple *specification language* allows communication events to be associated to *reactions*, which are sequences of operations executed atomically in response to specific communication primitives performed over tuple spaces. In other terms, a reaction can be thought as an event-handler catching communication events, and having full access to both the whole information concerning the specific communication event, and the current communication state, as represented by the tuple space.

By programming reactions, the effect of the execution of a communication operation can be extended as needed. Moreover, thanks to the reaction execution model, all the results due to a single communication operation (its own effect, and the effects of all the reactions associated to it) are made visible to the coordination entities as a single transition of the state of the communication abstraction. As a consequence, the behaviour of the coordination medium can be made as complex as desired at the component's perception level.

In principle, since the specification language is founded on the same communication pattern exploited for agent interaction (that is, logic tuples and basic operations over tuple spaces), components may be allowed to manipulate the communication abstraction behaviour. Along with the chance of performing deductions on the current coordination state provided by the $\mathcal{ACLT}$ notion of tuple space as a full-fledged logic theory, this opens the way to systems able to reason about themselves, and possibly self-modify their behaviour dynamically, according to the system goals.

[1] Although $\mathcal{ACLT}$ exploits multiple tuple spaces, we will henceforth leave this feature aside, since it is not relevant in the context of this work. Thus, we will always refer any communication primitive to a sort of "default tuple space", without specifying any tuple space name.

2.1 The reaction model

$ACLT$ (Linda-like) primitives have the same semantics as Linda ones. However, the behaviour of an $ACLT$ tuple space can be extended by exploiting reactions to add new effects to communication events.

The $ACLT$ reaction model is based on the idea of making communication events observable at the system level. For this purpose, any *physical* (communication) *event* can be associated with one or more *logical events*, each denoted by a unique name. Multiple logical events can be connected to the same physical event, as well as multiple physical events can correspond to the same logical event. The association between communication events and logical events is set by a special tuple of the form `map(Operation, Event)`, which captures the idea that each time `Operation` is performed on the tuple space, a logical `Event` occurs.

The $ACLT$ tuple space's programming model is based on the notion of *reaction*, triggered in response to logical events' occurrence, and specified through tuples of the form `react(Event,Body)`. The *reaction body* `Body` is the collection of the primitive operations to be executed when the logical `Event` occurs, and is syntactically defined as a conjunction of *reaction goals*. A reaction goal is either a state primitive (`current_agent/1`, `current_op/1`, ...), a term predicate (term equality/inequality, term unifiability/non-unifiability, ...), or a communication primitive (`out_r`, `in_r`, `rd_r`)[2]. In the special but frequent case that a single physical event is mapped onto a single logical event, a simplified shortcut syntax can be used, avoiding the `map/2` clause and expressing the reaction by means of a single `reaction(Operation,Body)` tuple.[3]

Reactions are executed only after the corresponding logical event has actually occurred: so, in particular, when a reaction to an `out(Tuple)` primitive is triggered, the emitted *Tuple* is already in the tuple space. Instead, `in` and `rd` communication primitives can be seen as made of two distinct communication events: the first query phase, when a tuple template is provided (the *pre* phase), and the subsequent answer phase, when a unifying tuple is eventually returned to the querying agent (the *post* phase).[4] According to that, $ACLT$ allows different reactions to be associated to each of these two phases [7], by means of the `pre/0` and `post/0` predicates, which succeed only in the corresponding phase. So, for instance, when the reactions possibly associated to the the *post* phase of an `in(Tuple)` primitive are executed, the returned tuple (i.e., the one uni-

[2] These are the only communication primitives which can occur inside a reaction: `out_r` works as a conventional `out`, while `in_r` and `rd_r` correspond to `in_noblock` and `rd_noblock`, respectively. Consequently, blocking primitives are not allowed inside reaction goals.

[3] Although in the following examples we will exploit only the simplified `reaction/2` syntax, the `map/2` + `react/2` syntax may come to be useful whenever the same reaction body ought to be repeated as a response to many different physical events, as in the case of the tracer presented in [7].

[4] Correspondingly, the `current_tuple` primitive returns the tuple template in the first phase, and the unified tuple in the answer phase.

fying with *Tuple*) is no longer in the tuple space, while it is still there during the execution of reactions possibly associated to the the *pre* phase of the `in` primitive.

Because reaction goals are actually executed sequentially, their relative order may influence the result of the reaction [8]. Since multiple `react/2` tuples can be specified for a given logical event - as well as multiple `reaction/2` for the same communication primitive -, multiple reactions may be triggered at the same time: in principle, such reactions are executed as mutually-independent actions, in a non-deterministic order.

2.2 Reactions as transactions

A *successful reaction* is one whose reaction goals are all executed successfully. Instead, if even one reaction goal fails, the reaction aborts. Only successful reactions produce effects, while failed reactions yield no results at all. If, for instance, the default tuple space contains no `value/1` tuple, the reaction body

```
in_r(value(X)), out_r(value(1)), out_r(value(s(X)))
```

fails and produces no effect, while the reaction body

```
out_r(value(1)), in_r(value(X)), out_r(value(s(X)))
```

succeeds, and eventually adds the tuple `value(s(1))` to the default tuple space.

At the system level, $\mathcal{ACLT}$ reactions are executed atomically with a transaction semantics: the (potentially multiple) effects of a successful reaction are carried out in a single transition of the tuple space state. Consequently, outside a reaction there is no way to perceive the multiplicity of effects possibly produced by the sequential execution of the reaction itself. So, in the case the reaction succeeds, all its side-effect operations are realised simultaneously, leading to a single observable state transition. Instead, a failed reaction is virtually cancelled, as if it had never been executed, and yields no effect at all. Consider, for instance, the following reaction:

```
map(out, event).
react(event, ( current_tuple(p(_)),
        in_r(p(a)), in_r(p(X)), out_r(pp(a,X)) )).
```

which could have been expressed, more concisely, also as:

```
reaction( out(p(_)), ( in_r(p(a)), in_r(p(X)), out_r(pp(a,X)) ))
```

Each time a new tuple is inserted in the tuple space with an `out`, this reaction checks for the presence of two `p/1` tuples (whose one should be `p(a)`) and, in the case they are found, it replaces them with one single `pp/2` tuple. If some reaction goal fails (possibly because there is only one `p/1` tuple instead of the two required), no tuples are actually removed from the tuple space, nor are any other side-effects ever produced. If the reaction succeeds, instead, all its associated side-effects (removal of two tuples, and addition of a third) are realised

altogether: so, the simultaneous presence of the two `p/1` tuples is perceived by the system as a single `pp/2` tuple.

As shown in Subsection 2.1, a multiplicity of reactions can be triggered in response to the same communication event, both because the latter has been mapped onto multiple logical events, or because multiple reactions have been specified for the same logical event. In addition, further reactions may be triggered as a consequence of the successful completion of another reaction, as a reaction body can contain communication primitives in its turn. In order to guarantee the transaction semantics, all such further reactions are executed only *after* the triggering reaction has been successfully completed: accordingly, reaction nesting is not permitted.

In any case, all reactions following an agent-triggered communication event - both triggered *directly* by the event and *indirectly* by other reactions produced by the event - are actually executed *before* serving any other agent-triggered communication event. As a result, agents can only perceive the final result of the execution of the communication event and the set of all the reactions it triggered (directly and indirectly). For instance, if the following reaction has been defined:

```
reaction( out(p(s(X))), ( in_r(p(s(X))), out_r(p(X)) ))
```

a component suspended on an `in(p(s(0)))` operation will not be waked up, as normally expected, by an `out(p(s(0)))` operation, as this reaction will first replace the `p(s(0))` tuple with a `p(0)` tuple. As a result, only the `p(0)` tuple is visible at the component's perception level, while the `p(s(0))` tuple is not perceived.

This behaviour introduces a new kind of tuple space state transition at the component's perception level, and enhances the expressive power of the coordination model. In fact, thanks to the execution model of $\mathcal{ACLT}$ reactions described above, coordination entities still perceive the response of a tuple space to a communication event as a single-step transition of the tuple space state. However, such a transition is no longer bounded to be simple (adding/deleting one tuple) and fixed by the model, like in Linda, but can be made as complex as desired. ¿From a component perspective, for instance, the previous reaction

```
reaction( out(p(_)), ( in_r(p(a)), in_r(p(X)), out_r(pp(a,X)) ))
```

has the effect of making the simultaneous presence of the two `p/1` tuples unperceivable, and of leading a single `out` operation to result both in the removal of a tuple and in the insertion of another. In addition, the inserted tuple is not the one specified in the `out` operation, but is related both to that one and to the tuple space state.

An $\mathcal{ACLT}$ tuple space is then an example of a programmable coordination medium, since its observable behaviour in response to communication events can be modified through reaction programming. By freeing the components from the charge of explicitly handling a (possibly complex) interaction protocol, a programmable coordination medium allows coordination entities to be designed according to a straightforward communication protocol, while charging the medium

of most of the low-level coordination details. Next Section discusses the benefits of this approach in the context of multi-agent systems.

3 Building multi-agent systems around a programmable coordination medium

Being intrinsically interactive [15], multi-agent systems are naturally characterised by the model of component interaction, as well as by the observable behaviour of their components, rather than by the rules of agent inner computation [8]. As a result, agent architectures can be designed independently of agent internal models, focussing on agent observable behaviour. Due to this shifting focus from agents to agent interaction, the communication abstraction is asked to play a major role within the coordination model of choice. In particular, once the coordination model for the multi-agent system is given, the choice of the interaction policy should not affect the single agent architecture, which could then concentrate on agent observable behaviour and on its interface towards the outside. In fact, it seems desirable that agents are designed according to a quite abstract model, so as to delegate the required interaction protocol to the communication abstraction behaviour. From a conceptual viewpoint, this makes coordination media [6], where interaction actually takes place, be in charge of the interaction policy, instead of the single coordination entities, which are not required to have a view of the system as a whole. In practice, this is particularly useful because agents of a multi-agent system may often be difficult or even impossible to modify, especially when dealing with legacy software components, whose observable behaviour could not be easy to accommodate so as to accomplish the interaction strategy of choice.

In order to show how a programmable coordination medium could be exploited in a multi-agent system, in the rest of this Section we discuss three examples of simple multi-agent systems based on $\mathcal{ACLT}$. The first one (Subsection 3.1), the classical *dining philosopher* problem [9], shows how some global properties of a multi-agent system (like deadlock avoidance) can be embodied into the communication abstraction. The second one (Subsection 3.2), a slight variation of the previous problem, is meant to show how a more complex interaction policy can be achieved by simply re-defining how the communication device works, with no changes to the interaction protocols of the philosopher agents. The last one (Subsection 3.3) generalises the previous example, discussing how the interaction policy can be handled and modified by an agent, possibly as a result of an inferential process on the current coordination state of the multi-agent system.

3.1 The dining philosophers

As an example of the flexibility provided by the extensibility of the communication abstraction to the $\mathcal{ACLT}$ model, we discuss an implementation of the classical dining philosopher problem, based on reactions. A characteristic of this

problem is that, in order to avoid deadlock situations, a philosopher should either get the two forks he needs to eat, or get none. This means that the two forks should be obtained through a transaction. In order to ensure fairness, moreover, fork release should be performed atomically. In fact, if both the left and the right neighbour of the currently-eating philosopher are waiting to eat, releasing one fork before the other would result in privileging one philosopher with respect to its colleague, which should be avoided.

When trying to express the solution to this problem in Linda, the main problem is that the natural choice of modelling the fork acquisition as a sequence of two *in* operations is not transactional, thus yielding a potential risk of deadlock. Similarly, a sequence of two *out* operations would not be atomic, thus not ensuring fairness. In such a framework, a safe solution requires that the user explicitly handles a locking mechanism, thus affecting the agent behaviour. Using $\mathcal{ACLT}$ reactions, instead, transactionality is guaranteed by suitably programming the tuple space behaviour, with no need for a more complex agent protocol. Thus, deadlock avoidance and fairness are obtained through the programmable coordination medium.

Philosopher agents are designed according to a very straightforward interaction protocol: when a philosopher wants to eat, he tries to acquire the two forks through an `in(forks(F1,F2))` operation; when he is satiated, and wants to start thinking, he gives the forks back by means of an `out(forks(F1,F2))` operation. So, all the charge of the interaction policy is up to the communication abstraction.

While resources are actually available singly in the tuple space (each fork is represented by a `fork(F)` tuple), philosophers view resources as pairs of forks (tuples `forks(F1,F2)`): the tuple space is then programmed so as to bridge the two different perceptions - the system level (`fork/1` tuples, representing single forks) and the agent level (`forks/2` tuples, representing pairs of forks).

For instance, the `forks(F1,F2)` tuple emitted by a philosopher agent when releasing forks is not perceived by the other agents, as it is immediately replaced with the two `fork(F1)`, `fork(F2)` tuples by the following reaction:

```
reaction( out(forks(F1,F2)), (                                    (1)
    in_r(forks(F1,F2)), out_r(fork(F1)), out_r(fork(F2)) )).
```

Handling fork requests, instead, is more complex, because the desired forks may not be immediately available. In this case, the `in(forks(F1,F2))` operation suspends, and the request should be recorded in the tuple space, so that it may be served later. Consequently, fork requests will be recorded in the tuple space by means of a reaction associated to the *pre* phase of the `in(forks(F1,F2))` operation. Such a tuple will then be retracted, by means of another appropriate reaction associated to the *post* phase of the same operation, when the philosopher has been served (after the proper forks have become available) and can start eating:

```
reaction( in(forks(F1,F2)), ( pre, out_r(required(F1,F2)) )).   (2)
reaction( in(forks(F1,F2)), ( post, in_r(required(F1,F2)) )).
```

Whenever a new fork request is recorded as a `required(F1,F2)` tuple, the tuple space is programmed so as to check whether the desired forks are immediately available, in what case it conquers the pair of resources by replacing the two `fork(F1)`, `fork(F2)` tuples with one single `forks(F1,F2)` tuple:

```
reaction( out_r(required(F1,F2)), (                                    (3)
    in_r(fork(F1)), in_r(fork(F2)), out_r(forks(F1,F2)) )).
```

Obviously, if the two forks are not available, this reaction fails, the philosopher agent stays suspended, and the `required/2` tuple remains in the tuple space.

However, waiting philosophers will be served later, when new single forks are made available as a consequence of a fork release performed by other agents. For this reason, the fork-release event has to be intercepted, and handled trying to group the pairs of forks needed by still-waiting philosophers. Since each fork may be requested, in principle, by two distinct philosophers, two reactions are needed - one for the left agent, and another one for the right agent. Each reaction checks whether there is a corresponding dangling fork request, and tries to serve it if this is the case:

```
reaction( out_r(fork(F)), ( rd_r(required(F1,F)),
    in_r(fork(F1)), in_r(fork(F)), out_r(forks(F1,F)) )).          (4)
reaction( out_r(fork(F)), ( rd_r(required(F,F2)),
    in_r(fork(F)), in_r(fork(F2)), out_r(forks(F,F2)) )).
```

Since reactions are executed transactionally, forks are reserved only in pairs when they are both available and needed by some agent: so, no deadlock can occur.

Notice that the agent model does not need to be specialised in order to accomplish the competition protocol: a philosopher simply asks for forks when hungry, and sets them free when satiated. Agent design can then concentrate on modelling agent internal architecture, while agent interaction model results quite simple and intuitive. A good deal of the intelligence of the system lays then in the interaction protocol, which is only of little concern for the single agent. Thus, the communication abstraction is specialised through suitable reaction programming so that it makes the system behave correctly, independently of the agent internal model: the only requirement is that the emerging behaviour of philosopher agents (their interaction model) accomplishes the very straightforward *acquire/release* protocol.

3.2 Philosophers dining with labelled forks

In order to show how an interaction policy can be modified and made more complex by changing the behaviour of the coordination medium, without affecting the interaction protocol of the coordination entities, we discuss a slight variation of the Dining Philosopher example. The basic problem is changed in that now there are three forks for each position on the table, labelled differently according to the kind of meal for which it has to be used: breakfast, lunch, or dinner. At any time in the multi-agent system, it is either breakfast, lunch, or dinner time.

When it is lunch time, for instance, only lunch forks can be used to start eating: however, a slowly-eating philosopher is allowed to keep on having his meal as long as he needs. So, if he starts eating at dinner time, he will be given dinner forks, and will be allowed to keep them for eating even when breakfast time comes around.

Since philosophers are supposed to be totally unaware of this enhancement, the philosopher protocol is exactly the same as in the previous example: in particular, they still try to get their pair of forks through an `in(forks(F1,F2))` operation, and still give them back by means of an `out(forks(F1,F2))` operation. Here, however, two contiguous philosophers, sharing a fork position, can eat at the same time (using, obviously, different forks obtained at different meal times), thus exploiting the extra resources - three forks instead of one. Take for instance the case of a two-philosopher system, where both agents get hungry at breakfast time. Only one of them (the *lucky philosopher*) will be assigned the breakfast forks, while the other (the *unlucky philosopher*) will be forced to wait. But when lunch time comes, the unlucky philosopher may be allowed to start eating even though the lucky one is still eating, because the lucky philosopher is using breakfast forks, and the lunch forks are free.

In order to achieve this new behaviour, we just have to slightly modify the internal representation of forks in the tuple space, adding a representation of the meal time concept, and updating the reactions of Subsection 3.1 accordingly. More precisely, the tuple space representation of the forks is changed from `fork(Fork)` to `fork(Meal,Fork)`, representing the fork *Fork* which can be used at *Meal* time. Moreover, a `timefor(Meal)` tuple is assumed to be always in the tuple space, indicating which forks to allocate to hungry philosophers at any time. Reactions, in their turn, should be modified so that:

- the reaction handling fork requests as `required(F1,F2)` tuples takes the meal time into account;
- the two reactions serving dangling fork requests take the meal time into account, too;
- an extra reaction takes care of serving dangling fork requests when the meal time changes and, therefore, new forks can be used.

So, reactions *(2)* remain untouched,

```
reaction( in(forks(F1,F2)), ( pre, out_r(required(F1,F2)) )).    (5)
reaction( in(forks(F1,F2)), ( post, in_r(required(F1,F2)) )).
```

while reaction *(1)* becomes

```
reaction( out(forks(F1,F2)), ( in_r(used(M,F1,F2)),                (6)
    in_r(forks(F1,F2)), out_r(fork(M,F1)), out_r(fork(M,F2)) )).
```

The `used/3` tuple is needed to track which forks are currently being used, given that, as discussed above, different types of forks may be used at the same time. Correspondingly, reaction *(3)* becomes

```
reaction( out_r(required(F1,F2)), ( rd_r(timefor(M)),
    in_r(fork(M,F1)), in_r(fork(M,F2)),                    (7)
    out_r(forks(F1,F2)), out_r(used(M,F1,F2)) )).
```

and reactions *(4)* become

```
reaction( out_r(fork(M,F)), ( rd_r(required(F1,F)),
    rd_r(timefor(M)), in_r(fork(M,F1)), in_r(fork(M,F)),
    out_r(used(M,F1,F)), out_r(forks(F1,F)) )).            (8)
reaction( out_r(fork(M,F)), ( rd_r(required(F,F2)),
    rd_r(timefor(M)), in_r(fork(M,F)), in_r(fork(M,F2)),
    out_r(used(M,F,F2)), out_r(forks(F,F2)) )).
```

The new reaction needed to handle meal time changes does basically the same, serving a dangling fork request when the new meal time allows new forks to be used:

```
reaction( out_r(timefor(M)), ( rd_r(required(F1,F2)),
    in_r(fork(M,F1)), in_r(fork(M,F2)),                    (9)
    out_r(forks(F1,F2)), out_r(used(M,F1,F2)) )).
```

As a result, new notions (like meal time and meal forks) are introduced in the system, new resources are made available (more forks), a new policy for resource assignment is adopted, but the philosopher agents can keep on using the same straightforward *acquire/release forks* protocol defined in the example of Subsection 3.1. So, reaction programming makes it possible for agents to maintain the same perception of the resource space as in the previous example, even though such a space has changed and made more complex. This feature is achieved by properly programming the communication abstraction so as to encapsulate changes and hide them from agents, actually embodying the new coordination policy into the coordination medium.

3.3 Agents requiring labelled resources

In the previous example, a new interaction policy is achieved by properly re-programming the communication abstraction. How such a re-programming is performed is not specified, so one could think that the designer has to deal with this. However, $\mathcal{ACLT}$ support for logic agents with inferential capabilities suggests that a specific agent may be charged of such a task. This example shows how the modification of the interaction policy can be achieved dynamically, likely as a result of a reasoning over the current state of the coordination medium performed by a logic agent, working as a meta-level "supervisor".

For this purpose, the example of Subsection 3.2 is generalised by replacing the notion of meal time (and meal-labelled forks) with a general resource labelling scheme. Resources are now grouped in classes, and represented by tuples of the form `res(`*`Type`*`,`*`Name`*`)`, where *`Type`* represents the resource class and *`Name`* the resource name. Generalising the meal time notion of the previous example,

represented there by the tuple `timefor(Meal)`, we no longer suppose that only one class of resources is made available at one given time. So, the tuple space may contain more than one `class(Type)` tuple at the same time, each representing one class *Type* of available resources.

Agents require groups of n homogeneous (i.e., of the same class) resources through `in(resources(A,R1,...,Rn))` operations, then release them by means of `out(resources(A,R1,...,Rn))` operations, where *A* is the agent identifier. Agents are free to ask for as many resources as they need ($\leq max$), so that agent `a1` may ask for two resources `r1`, `r2` through an `in(a1,r1,r2)`, while agent `a2` may ask for three resources `r1`, `r3`, `r4` through an `in(a2,r1,r3,r4)`.

While the agents perceive resources as groups of unlabelled items *R1*, ..., *Rn*, the system handles single labelled items in form of `res(Type,Res)` tuples. For this purpose, reactions *(10-11)* are defined, handling requests for groups of resources, and recording them as `req(A,R1,...,Rn)` tuples. In particular, the agent protocol allows now agents to ask for resources in either a blocking (reaction *(10)*) or a non-blocking way (reaction *(11)*).

```
reaction( in(resources(A,R1)), ( pre,
    out_r(req(A,R1)) )).
...                                                          (10)
reaction( in(resources(A,R1,...,Rn)), ( pre,
    out_r(req(A,R1,...,Rn)) )).
```

```
reaction( in_noblock(resources(A,R1)), ( pre,
    out_r(req(A,R1)) )).
reaction( in_noblock(resources(A,R1)), ( post, failure,
    in_r(req(A,R1)) )).
...                                                          (11)
reaction( in_noblock(resources(A,R1,...,Rn)), ( pre,
    out_r(req(A,R1,...,Rn)) )).
reaction( in_noblock(resources(A,R1,...,Rn)), ( post, failure
    in_r(req(A,R1,...,Rn)) )).
```

Resource release is handled by reaction *(12)*, which is in charge of making resources released (as a group) by one agent available as single resources to all agents.

```
reaction( out(resources(A,R1)), (
    out_r(res(T,R1)),
    in_r(used(A,T,R1)), in_r(resources(A,R1)) )).
...                                                          (12)
reaction( out(resources(A,R1,...,Rn)), (
    out_r(res(T,R1)), ..., out_r(res(T,Rn)),
    in_r(used(A,T,R1,...,Rn)), in_r(resources(A,R1,...,Rn)) )).
```

A new agent request can be served when either a new request is performed and all needed resources are free (reaction *(13)*), or one resource is released (in which case – reaction *(14)* – all permutations are to be considered), or a new class of resources is made available (in which case – reaction *(15)* – all pending requests are to be checked).

```
reaction( out_r(req(A,R1)), ( rd_all_r(class(T),TL),
    out_r(lreserve(req(A,R1),TL)) )).
...                                                          (13)
reaction( out_r(req(R1,...,Rn)), ( rd_all_r(class(T),TL),
    out_r(lreserve(req(A,R1,...,Rn),TL)) )).


reaction( out_r(res(T,R)), ( rd_r(class(T)),
    rd_all_r(req(A,R),ReqL),
    out_r(lreserve(ReqL,class(T))) )).
...
reaction( out_r(res(T,R)), ( rd_r(class(T)),
    rd_all_r(req(A,R,...,Rn),ReqL),
    out_r(lreserve(ReqL,class(T))) )).
...                                                          (14)
reaction( out_r(res(T,R)), ( rd_r(class(T)),
    rd_all_r(req(A,R1,...,R,...,Rn),ReqL),
    out_r(lreserve(ReqL,class(T))) )).
...
reaction( out_r(res(T,R)), ( rd_r(class(T)),
    rd_all_r(req(A,R1,...,R),ReqL),
    out_r(lreserve(ReqL,class(T))) )).


reaction( out(class(T)), ( rd_all_r(req(A,R1),ReqL),
    out_r(lreserve(ReqL,class(T))) )).
...                                                          (15)
reaction( out(class(T)), ( rd_all_r(req(A,R1,...,Rn),ReqL),
    out_r(lreserve(ReqL,class(T))) )).
```

In any case, a `reserve(`*`Req`*`,`*`Type`*`)` tuple is produced through reaction *(16)* for any possible match between the available resources and the pending requests, and is then handled by reaction *(17)*).

```
reaction( out_r(lreserve(Req,T)), ( in_r(lreserve(Req,T)) )).
reaction( out_r(lreserve(Req,[T|TL])), ( out_r(reserve(Req,T))
    out_r(lreserve(Req,TL)) )). ...                          (16)
reaction( out_r(lreserve([Req|ReqL],T)), ( out_r(reserve(Req,T))
    out_r(lreserve(ReqL,T)) )).


reaction( out_r(reserve(Req,T)), ( in_r(reserve(Req,T)) )).
reaction( out_r(reserve(req(A,R1),class(T))), (
    in_r(res(T,R1)),
    out_r(resources(A,R1))
    out_r(used(A,T,R1)), in_r(req(A,R1)) )).
...                                                          (17)
reaction( out_r(reserve(req(A,R1,...,Rn),class(T))), (
    in_r(res(T,R1)), ..., in_r(res(T,Rn)),
    out_r(resources(A,R1))
    out_r(used(A,T,R1,...,Rn)), in_r(req(A,R1,...,Rn)) )).
```

Finally, reactions *(18–20)* handle the deletion of a class of resources, performed through either a blocking or a non-blocking *in* operation.

```
reaction( in(class(T)), ( post,                              (18)
    out_r(noclass(T)) )).
```

```
reaction( in_noblock(class(T)), ( post, success,                    (19)
    out_r(noclass(T)) )).
```

```
reaction( out_r(noclass(T)), ( in_r(noclass(T)) )).
...                                                                 (20)
reaction( out_r(noclass(T)), ( in_r(class(T)),
    out_r(noclass(T)) )).
```

In this example, many sets of resources can be made available at the same time through the simultaneous presence of many `class(Type)` tuples in the tuple space. Classes of resources can then be added and removed by inserting (reaction *(15)*) and removing (reaction *(18)*) `class(Type)` tuples. This allows resources to be allocated according to arbitrarily complex strategies, possibly driven by the reasoning of a logic agent on the content of the tuple space. For instance, the supervisor may realise that the system load is too heavy, checking the number of pending resource requests (represented by `req(A,R1,...,Rn)` tuples) by properly combining $\mathcal{ACLT}$ *demo* primitives with side-effect communication primitives. Thus, it may decide to add a new resource set to the system through a set of `res(NewType,Name)` tuples, and make them available to the agents by simply adding a single `class(NewType)` tuple.

4 Related works and conclusions

The particular instantiation of the notion of programmable coordination medium presented here (the $\mathcal{ACLT}$ programmable tuple space) deeply relies on the concept of reaction, like many other different coordination models. For instance, the chemical metaphor of Gamma [1] uses reactions to specify very general coordination laws in terms of *reaction conditions* and consequent *actions*, but no communication abstraction is provided, nor is any agent interaction protocol. As it can be argued from the Dining Philosopher example shown in [2], reactions are the only means for the evolution of a multi-agent system based on Gamma, since the model does not account for agent deliberative activity.

Also the ESP coordination language [5] is based on the notion of multiple logic tuple space, and exploits reactiveness of the tuple space. However, the computational shift from the agents to the communication abstraction is even stronger than in $\mathcal{ACLT}$, as ESP reduces the notion of agent to a purely reactive execution thread.

According to our perception, coordination architectures may actually take advantage from being based on a programmable coordination medium, that is a communication abstraction whose behaviour is not fixed, but can be extended and tailored to accomplish the overall system goals. In this work, we have explored the benefits of this approach in the context of the Linda-based $\mathcal{ACLT}$ coordination model, where the Linda tuple space communication abstraction is enhanced through reaction programming. Indeed, we suggest that the same approach may also be successfully exploited in typical multi-coordinated architectures based on message passing and peer-to-peer communication, like World

Wide Web servers, where no global communication abstraction is a-priori available. This is particularly true when considering the increasing request for value-added services, calling for more flexible and intelligent system behaviours. By combining a programmable coordination medium with the agent capability of performing inferential activities over the state of the coordination, one could build multi-component software systems able to intelligently drive their own evolution by dynamically self-modifying the communication abstraction behaviour.

References

1. J.-P. Banâtre and D. le Métayer. The Gamma model and its discipline of programming. *Science of Computer Programming*, 15(1):55–77, November 1990.
2. J.-P. Banâtre and D. le Métayer. Programming by multiset transformation. *Communications of the ACM*, 36(1):98–111, January 1993.
3. Kraig Brockschmidt. *Inside OLE*. Microsoft Press, 1995. 2nd ed.
4. A. Brogi and P. Ciancarini. The concurrent language, Shared Prolog. *ACM Transactions on Programming Languages and Systems*, 13(1), January 1991.
5. P. Ciancarini. Distributed programming with logic tuple spaces. *New Generation Computing*, 12, 1994.
6. P. Ciancarini. Coordination models and languages as software integrators. *ACM Computing Surveys*, 28(2), June 1996.
7. E. Denti, A. Natali, A. Omicini, and M. Venuti. An extensible framework for the development of coordinated applications, 1996. First International Conference, COORDINATION'96, Cesena, Italy, April 15–17, 1996.
8. E. Denti and A. Omicini. Designing multi-agent systems around an extensible communication abstraction. In A. Cesta and P.-Y. Schobbens, editors, *Proceedings of the 4th ModelAge Workshop on Formal Models of Agents, Certosa di Pontignano, Italy, January 15–18, 1997*, pages 87–97. National Research Council of Italy, 1997. To be published by Springer-Verlag in the LNAI Series.
9. E.W. Dijkstra. *Co-operating sequential processes*. Academic Press, London, 1965.
10. D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1), January 1985.
11. D. Gelernter. Multiple tuple spaces in Linda. In *Proceedings of PARLE*, volume 365 of *LNCS*, 1989.
12. D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, February 1992.
13. Object Management Group. The common object request broker: Architecture and specification. Technical report, OMG, July 1995. Rev. 2.0.
14. A. Omicini, E. Denti, and A. Natali. Agent coordination and control through logic theories. In *Topics in Artificial Intelligence - 4th Congress of the Italian Association for Artificial Intelligence, AI*IA '95*, volume 992 of *LNAI*, pages 439–450, Firenze, Italy, October 11–13 1995. Springer-Verlag.
15. P. Wegner. Interactive foundations of computing. Technical report, Brown University, Providence (RI), August 1996.