An Architecture for Tuple-based Coordination of Multi-agent Systems

ENRICO DENTI AND ANDREA OMICINI

LIA – Laboratorio di Informatica Avanzata, DEIS – Università di Bologna, Viale Risorgimento 2, I-40136, Bologna, Italy
(email: fedenti,aomicini@deis.unibo.it)

SUMMARY
Multi-agent system development calls for powerful and expressive coordination models and languages, as well as for an effective coordination technology. A good deal of the current research effort focuses on tuple-based coordination, exploiting its well-known advantages, such as agent uncoupling and associative access to information, and addressing its limitations in terms of flexibility and control capabilities. In particular, the behaviour of a Linda-like tuple space is fixed once and for all by the coordination model, and cannot be tailored to the specific application needs.

Tuple centres are tuple spaces whose behaviour can be programmed by defining transactional reactions to the basic communication events, allowing coordination laws to be explicitly defined and embedded into the coordination medium. This paper presents the architecture of a run-time system for tuple-based coordination, where tuple centres work as an extensible kernel, around which multi-agent systems can be designed and deployed. After sketching the implementation, the paper shows the advantages that can be achieved from both the design and the performance viewpoints. Copyright © 1999 John Wiley & Sons, Ltd.

KEY WORDS: Linda run-time system; tuple centres; coordination technology; multi-agent systems; extensible kernel

INTRODUCTION

Literature on coordination models introduced the idea that interaction and computation are two orthogonal dimensions, clearly separating the computational aspects, which concern a single execution flow, from component interaction aspects, which concern communication, synchronisation, etc. According to this idea, coordination is more than just simple communication or component interoperability, as supplied by technologies like CORBA [1] or DCOM [2]: actually, the purpose of coordination is not just to let components inter-operate, but also to help them to work together, ‘merging separate activities into an ensemble’ [3].

Most of the communication and synchronisation metaphors classically used for system design, like direct communication or the client/server design pattern, are weak from this viewpoint. Coordination entities (processes, objects, components, agents) are typically in charge of handling interaction protocols, so their design improperly merges computation and coordination aspects. In multi-agent systems, this problem makes the design of both the whole
system and the individual agents unnecessarily complex. Instead, research on coordination shows that an effective coordination model may substantially decrease such a complexity [4,5], by providing for suitable communication mechanisms and abstractions.

Coordination models like the blackboard-based ones [6] and Linda [7] provide for a global, explicit, shared communication abstraction, around which a multi-component system can be built in a natural way. In fact, a shared memory abstraction like a Linda tuple space not only supports the separation between the computation and the coordination models, but also features some fundamental properties such as generative communication, time and space uncoupling, and name independence [7].

Tuple-based communication abstractions have a clear impact on the way coordination entities and interactive systems are designed [8], as implicitly endorsed by the recent efforts on the JavaSpaces [9] and T Spaces [10] technologies. However, such abstractions are weak in flexibility and control capabilities, because the available primitives and their behaviour are fixed once and for all by the coordination model. So, the services provided by these coordination media cannot be modified or extended according to the specific application needs. This limitation restricts the benefit of using such a communication abstraction to very simple cases: since there is no way to change its behaviour, either its default one is adequate to the application needs, or components have to adapt themselves to the abstraction level provided by the communication abstraction, by taking charge of all the coordination policies not directly supported by the coordination media. Instead, the clear separation between computation and coordination issues is a requirement for the engineering of complex interactive systems, since it allows each component to be designed according to the most suitable abstraction level, independently of the other components.

The considerations above constituted the main motivations to introducing the notion of tuple centre as a communication abstraction that can work as a programmable coordination medium [11]. A tuple centre subsumes and extends a tuple space in that its behaviour in response to communication events can be programmed so as to embed the coordination laws. Thus, tuple centres do not provide just a support for inter-component communication or a set of services for component interoperability: instead, they have autonomous computational capabilities, which can be exploited to support component interaction at the desired abstraction level. This feature leads to a novel programming style [12], as components can be designed by focusing on their own specific tasks, delegating coordination to the coordination medium. Furthermore, the availability of a multiplicity of distributed coordination media makes it possible to encapsulate and spread coordination rules throughout the distributed system where needed.

Tuple centres also provide for the full observability of communication events, thus intrinsically supporting the dynamic view of communication that is needed in distributed programming for monitoring, tracing, and debugging purposes [13]. This feature makes it possible to seamlessly integrate application programming and system programming in a uniform framework [14].

The aim of this paper is to present the architecture of an extensible run-time kernel for the LuCe (Logic Tuple Centres) system, which exploits tuple centres for the coordination of multi-agent systems. From the conceptual design down to the implementation scheme, this approach (i) generalises the basic Linda model, (ii) supports open and heterogeneous multi-agent systems, (iii) promotes the design and the incremental development of flexible, easy-to-maintain applications, and (iv) provides for abstractions where to embody the laws for the coordination of a multi-agent system.

A **coordinated system** can be defined as a collection of **coordination entities** [5], which are independent computational elements whose mutual interaction determines the global behaviour of the coordinated system. In the context of this paper, coordination entities are referred to as **agents**, by implicitly assuming the broad notion of agent endorsed by the most recent research on Internet applications [15]. More precisely, an agent is a software component that interacts with the surrounding environment so as to pro-actively carry out its task.

**Coordination** is then aimed at making agent interaction result in a coherent application dynamics accomplishing global system requirements [16]. In tuple-based coordination models, agents interact by exchanging **tuples**, which are ordered collections of possibly heterogeneous information items. Inter-agent communication occurs indirectly through a multiplicity of **tuple spaces**, which are associative, stateful communication abstractions.

Agents communicate, synchronise and cooperate by storing, reading and consuming tuples in tuple spaces. The **coordination language** provides agents with the primitives required to read, store and consume tuples: for instance, the Linda coordination language consists of the classical **out**, **rd** and **in** coordination primitives. The **communication language**, instead, is the language used by agents to talk to each other and exchange information via the coordination primitives, and essentially matches the set of the admissible tuples [17].

Though Linda originated in the field of parallel programming, with closed implementation and compiler-based optimisation techniques, the features of tuple-based models make them well-suited for the coordination of open, distributed, heterogeneous systems [18,19,20,21], with implementations based on **run-times** [22]. In particular, the clean separation between computation and coordination [3] is an essential requirement when building open and heterogeneous systems, while generative communication makes it possible to uncouple agents both spatially and temporally, which is particularly useful when mobile agents are concerned [23]. Furthermore, associative access to communication information makes tuple-based models well-suited for dealing with heterogeneous and dynamic information systems like the Internet-based ones [24].

Thus, tuple-based coordination models provide several advantages for application design and development. However, these models lack the flexibility and control required by today’s complex multi-component applications, because the dynamics of a tuple space as a coordination medium is defined once and for all by the coordination model, and cannot be tailored to the specific application needs. To overcome this limitation, two approaches may be followed in principle, which allow tuple spaces to be enhanced with new capabilities according to the application needs, by enabling either (i) new primitives to be defined, or (ii) the semantics of the existing primitives to be modified. As an example of the former approach, the York Kernel II [22] introduces new bulk primitives, like **copy**, **collect**, to address the ‘multiple rd problem’ [25]. Instead, the latter approach motivated the idea of a **programmable coordination medium** [11], that is, a communication abstraction whose behaviour can be charged of the full coordination load, and led to the notion of tuple centre.

**TUPLE CENTRES**

**Beyond tuple spaces**

One of the main advantages of interacting through a tuple space is that coordination is information-based: agents synchronise, cooperate and compete based on the information
available in the shared dataspace, by associatively accessing, consuming, and producing information. While this approach makes interaction protocols simple yet expressive, representing the information in forms of tuples is also the source of one of the main limits of tuple-based coordination, because there is no possible distinction between the information conveyed by tuples and its representation in a tuple space. So, there is no way to separate how information is represented from how it is perceived by agents.

Take for instance the well-known Dining Philosophers problem [26]: in a tuple space, chopsticks could be represented either singly, by means of a tuple like chop(\(i\)), or as pairs, using, for instance, a tuple such as chops(\(i, j\)) for the two adjacent chopsticks \(i\) and \(j\). The first choice is the most natural for the domain representation, but can easily lead to deadlock if a philosopher is not enabled to get atomically the two chopsticks he needs. The second choice solves the deadlock problem, but introduces a new one, that is, how to ensure a coherent representation of the domain, since it should be guaranteed, for instance, that tuples chops(2, 3) and chops(4, 5) are no longer available once chops(3, 4) has been taken.

As a result, since coordination is information-driven, there is no way to express coordination policies that are not directly supported by the standard tuple space mechanisms. Instead, the load of coordination has generally to be charged upon the coordinated entities, which have to be made ‘coordination–aware’. Therefore, agents cannot abstract from the coordination details, but have to embed in their code the interaction protocols required to implement the required coordination policies. Computation and coordination issues are then undesirably merged in the agent design, making it unnecessarily complex [11, 27]. Moreover, this approach does not cope well with system openness, since it requires agents to agree not only on the static format for knowledge interchange, but also on the dynamics of the knowledge access protocols.

In the Dining Philosophers example, for instance, avoiding deadlock when chopsticks are represented singly requires philosopher agents to agree on a tuple space locking protocol, such that a unique semaphore tuple is taken from the tuple space before asking for chopsticks, and is released just after (see Table IV). This approach makes the communication load grow, since further tuples have to be added and removed. Also, there is no means to ensure that a philosopher always adheres to the required locking protocol, since there is no way to enforce the laws of coordination.

Instead, a more satisfactory solution would be to keep information representation and perception separate, relating them according to the desired coordination laws. Such an approach would enable agent interaction protocols to be organised around the desired agent perception of the communication space, independently of its actual contents in terms of tuples. Moreover, by properly relating information representation and perception, the coordination media can be charged of the coordination load, embedding whichever coordination law is required – even though not supported directly by the coordination medium.

Referring again to the Dining Philosophers example, this approach would make it possible to retain the representation of chopsticks as single chop(\(i\)) tuples in the tuple space, while enabling philosopher agents to perceive chopsticks as pairs, to be acquired and released by means of a single tuple space operation. So, for instance, the agent at place 3 should ask for chopsticks 3 and 4 as a single chops(3, 4) tuple: its request would then be turned into the atomic removal of chop(3) and chop(4) from the tuple space if and only if both tuples are available.

This behaviour can be obtained by (i) maintaining the standard tuple space interface, while at the same time (ii) making it possible to enrich the behaviour of a tuple space in terms of state
transitions in response to standard communication events. This is the motivation behind the
very notion of tuple centre, which is a tuple space enriched with the notion of programmable
behaviour specification. A tuple centre has exactly the same interface as a standard tuple
space and is perceived by agents as such, but may behave in a completely different way, since
its behaviour encapsulates the coordination rules governing the interaction.

A tuple centre behaviour specification consists of a collection of reaction specifications,
associating any of the standard tuple space communication primitives to specific
computational activities, called reactions. A reaction is defined as a set of non-blocking
operations, and has a success/failure transactional semantics: successful reactions can
atomically produce effects on the tuple centre state, failed reactions yield no result at all.
Each reaction can freely access and modify the tuples in the tuple centre, and can access all
the information related to the triggering communication event, such as the performing agent,
the required operation, the involved tuple, etc.

In principle, each communication event may trigger a multiplicity of reactions: however,
all the reactions executed as a consequence of a single communication event are carried
out in a single transition of the tuple centre state, before any other component-triggered
communication event is served. Therefore, from the agents’ viewpoint, the result of the
invocation of a communication primitive is the sum of the effects of the primitive itself and of
all the reactions it triggered, perceived altogether as a single-step transition of the tuple centre
state.

So, the semantics of communication events is no longer constrained to adding, reading, and
removing single tuples as in the Linda model, but can be made as complex as required by the
specific application needs [17]. This chance makes it possible to uncouple the agent’s view
of the tuple centre from the actual tuple centre state, and to relate them so as to embed the
coordination laws.

**Tuple centres in LuCe**

The LuCe coordination system exploits tuple centres as its coordination media. Its
communication language is based on first-order logic, so agents interact by exchanging logic
tuples [28,19]. In particular, a tuple is a ground fact, any unitary clause is an admissible tuple
template, and unification is the tuple matching mechanism.

Agents perceive a LuCe tuple centre as a logic tuple space, which can be accessed through
the standard Linda communication operations over logic tuples: out, in, rd, inp, rdp. These operations define the LuCe coordination language, and work in LuCe as expected by a
Linda-like model. In short, out puts a tuple in the tuple centre, while the query primitives
in, rd, inp, rdp first provide for a tuple template, and then expect a matching tuple as an
answer from the tuple centre. More in detail, in and inp delete the matching tuple from the
tuple centre, while rd and rdp leave it there; in and rd wait until a suited tuple is available
in the tuple centre, inp and rdp fail if no such a tuple is found.

What is relevant for our purposes is the relationship between the communication primitives
invoked by agents and the corresponding communication events crossing the tuple centre
boundaries. In particular, the query primitives actually result in two communication events
[29], while out results in a single event, given the asynchronous semantics adopted by LuCe
for this operation [30]. In the following, the expression pre phase uniformly denotes the single
event of the out primitive and the asking phase of the query primitives, whose answer phase
is called the post phase.
Table I. Main ReSpecT predicates for reactions

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>out_r(T)</code></td>
<td>Succeeds and inserts tuple T into the tuple centre</td>
</tr>
<tr>
<td><code>rd_r(T_T)</code></td>
<td>Succeeds, if a tuple T matching template T_T is found in the tuple centre, by unifying T with T_T; fails otherwise</td>
</tr>
<tr>
<td><code>in_r(T_T)</code></td>
<td>Succeeds, if a tuple T matching template T_T is found in the tuple centre, by unifying T with T_T and removing T from the tuple centre; fails otherwise</td>
</tr>
<tr>
<td><code>no_r(T_T)</code></td>
<td>Succeeds, if no tuple matching template T_T is found in the tuple centre; fails otherwise</td>
</tr>
</tbody>
</table>

**Communication event information**

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>current_tuple(T)</code></td>
<td>Succeeds, if T matches the tuple involved by the current communication event</td>
</tr>
<tr>
<td><code>current_agent(A)</code></td>
<td>Succeeds, if A matches the identifier of the agent that triggered the current communication event</td>
</tr>
<tr>
<td><code>current_op(Op)</code></td>
<td>Succeeds, if Op matches the operation that produced the current communication event</td>
</tr>
<tr>
<td><code>current_tc(N)</code></td>
<td>Succeeds, if N matches the identifier of the tuple centre performing the computation</td>
</tr>
<tr>
<td><code>pre</code></td>
<td>Succeeds in the pre phase of any operation</td>
</tr>
<tr>
<td><code>post</code></td>
<td>Succeeds in the post phase of any operation</td>
</tr>
<tr>
<td><code>success</code></td>
<td>Succeeds in the pre phase of any operation, and in the post phase of any successful operation</td>
</tr>
<tr>
<td><code>failure</code></td>
<td>Succeeds in the post phase of any failed operation</td>
</tr>
</tbody>
</table>

Each LuCe tuple centre is denoted by a unique global identifier, and is conceptually an ever-existing and ever-lasting abstraction. Thus, LuCe is by definition coordination transparent [17], since coordinated entities can exploit tuple centres without caring of their creation. For instance, an agent could ask tuple centre tc to perform an in operation with tuple template Temp1 by means of a tc?in(Temp1) call, with no need to worry about tc existence: so, there is no need for meta-coordination primitives like tsc [22] for tuple space creation.

The behaviour of a LuCe tuple centre is defined by means of the ReSpecT specification language [27], where behaviour specifications are expressed in terms of first-order logic tuples, called specification tuples. So, LuCe tuple centres are conceptually structured in two distinct sections, the tuple space – for ordinary data tuples – and the specification space – for specification tuples. In particular, a specification tuple reaction(Op, R) associates the event generated by an incoming communication operation Op to the reaction R. A ReSpecT reaction is then defined as a sequence of reaction goals, which may access properties of the occurred event, perform simple term tests, and operate on tuples in the tuple centre (see Table I). In particular, out_r, in_r, rd_r, no_r operations work on the tuple space similarly to communication operations, and can trigger further reactions in a chain.

Reaction goals are executed sequentially, each with a success/failure semantics. Correspondingly, the reaction as a whole is either a successful or a failed one, depending on whether all its reaction goals succeeded or not, as stated by the required transactional semantics. Moreover, all the reactions triggered by a given communication event are executed before serving any other communication event, so as to ensure that agents perceive only the final result of the execution of the communication event and the set of all the triggered reactions. This is what makes it possible to embody a new observable behaviour in a tuple.
centre, transparently to agents. Since ReSpecT is Turing-equivalent [27], any computable coordination law can be encapsulated into the coordination medium.

Coherently with the distributed nature of component-based systems, each tuple centre stores both its data tuples and its behaviour specification tuples locally, making it possible to distribute coordination rules throughout the system where needed. So, coordination laws are implemented locally to a tuple centre, but affect a multi-agent system globally, since all the agents interacting through the same tuple centre are subject to the same coordination laws, independently of their current location on the distributed system.

Furthermore, thanks to their uniform representation, agents can manipulate data tuples and behaviour specification tuples adopting the same conceptual protocol. So, in principle, agents may modify/integrate the behaviour of an interactive system in the same straightforward way as they communicate with other agents, that is, by adding, removing, and reading specification tuples. To this end, LuCe provides a special set of communication primitives: outSpec, inSpec, rdSpec, inpSpec, and rdpSpec.

The distinction between the tuple space and the specification space suggests that two levels of abstraction may be adopted over the space of interaction: the communication viewpoint – represented by the state of the tuple space – and the coordination viewpoint – represented by the specification space. In fact, since the state of the space of tuples represents at any time the current state of agent interaction, each tuple space represents a portion of the global interaction space, where inter-agent communication takes place. On the other hand, specification tuples actually define the agent coordination rules. So, the agent’s capability to handle both the tuple space and the specification space of a tuple centre actually enables agents to adopt both the communication and the coordination abstraction levels over the coordinated system, acting as either a coordinated or a coordinating entity, as needed.

Examples

A bulk primitive. Some Linda-based systems provide a bulk primitive like collect [22], which takes a tuple template and returns the list of all the tuples matching the template, removing them from the tuple space. In LuCe, an equivalent behaviour can be achieved by properly programming the tuple centre in an agent-transparent way.

For this purpose, let us suppose that agents ask for the list \( L \) of all the tuples matching the template \( T \) by means of an \( \text{in} (\text{all}(T, L)) \) operation: then, the tuple centre should intercept it and, before the \( \text{in} \) operation is actually served, create the corresponding \( \text{all} \) tuple. To this end, the \( \text{query phase} \) of the \( \text{in} \) operation should trigger the proper reactions. The first reaction in Table II just adds two tuples to the tuple centre: \( \text{list} \) stores the matching tuples, while \( \text{inall} \) triggers further reactions in a ‘tail-recursive’ reaction chain. The two further reactions handle the reaction chain, respectively when there are other matching tuples, and when there are no matching tuples left. In the first case, the matching tuple is removed and added to the list, and the reaction is re-triggered; in the second case, the computation just terminates producing the required \( \text{all} \) tuple.

Dining Philosophers. In this classical problem, \( N \) philosopher agents share \( N \) chopsticks and a spaghetti bowl. Each philosopher needs two chopsticks to eat, but each chopstick is shared by two adjacent philosophers: so, the two chopsticks have to be acquired atomically to avoid deadlock, and released atomically to ensure fairness. This problem has been chosen as one of the simplest coordinated systems which overcomes the Linda coordination capabilities. More realistic examples of coordinated systems, dealing for instance with distributed information
Table II. Implementation of a bulk primitive

reaction( in(all(T)), ( pre, out_r(list([])), out_r(inall(T)) )).
reaction( out_r(inall(T)), ( in_r(inall(T)), out_r(inall(T)),
in_r(T), in_r(list(L)), out_r(list([T|L]))) )).
reaction(out_r(inall(T)), ( in_r(inall(T)),
no_r(T), in_r(list(L)), out_r(all(T,L)) )).

Table III. Coordination of the dining philosophers

reaction( out(chops(C1,C2)), ( in_r(chops(C1,C2)),
out_r(chop(C1)), out_r(chop(C2)) )).
reaction( in(chops(C1,C2)), ( pre, out_r(required(C1,C2)) )).
reaction( in(chops(C1,C2)), ( post, in_r(required(C1,C2)) )).
reaction( out_r(required(C1,C2)), ( in_r(chop(C1)), in_r(chop(C2)), out_r(chops(C1,C2)) )).
reaction( out_r(chop(C)), ( rd_r(required(C1,C)),
in_r(chop(C1)), in_r(chop(C)), out_r(chops(C1,C)) )).
reaction( out_r(chop(C)), ( rd_r(required(C,C2)),
in_r(chop(C)), in_r(chop(C2)), out_r(chops(C,C2)) )).

retrieval or workflow management [31], would have been unnecessarily complex for the purposes of this paper.

As discussed above, while the most natural choice for the domain is to represent chopsticks singly, agents are naturally led to handle chopstick pairs: the key benefit of tuple centre is that it allows these two perceptions to coexist. The availability of the i-th chopstick can then be represented as a logic tuple chop(i), while philosophers can accomplish a very simple interaction protocol based on acquiring/releasing chopstick pairs by means of an in(chops(C1,C2)) / out(chops(C1,C2)), with no need to be aware of the actual representation of the single chopsticks in the tuple space.

These two perceptions are then bridged by the reactions in Table III, which handle chop/1 and chops/2 tuples consistently. So, for instance, if one agent performs an out(chops(1,2)) and another one an out(chops(3,4)), an agent waiting for a chops(2,3) tuple is provided with such a tuple by reactions and can therefore resume execution, while tuples chop(1) and chop(4) become available in the tuple centre. The result is a multi-agent system where all the charge of the interaction policy is up to the coordination medium.

In principle, this approach allows coordination policies to be changed, possibly in a dynamic way, without affecting the agent interaction protocols. For instance, a variant of the Dining Philosophers‘ problem that adopts multiple sets of chopsticks for breakfast, lunch, and dinner [11] can be implemented using the same acquire/release protocol discussed here, by just re-programming the tuple centre so as to reflect the new coordination policy.

SYSTEM ARCHITECTURE

A LuCe coordinated system consists of a multiplicity of nodes hosting agents and tuple centres. The collection of all the tuple centres of a LuCe system constitutes a single global
space for agent interaction. As a fundamental design choice, LuCe is coordination-transparent [17], in that agents do not have to care about the creation, existence, location, or deletion of tuple centres. So, agents do not perceive any detail of the physical distribution of tuple centres over the network. Moreover, tuple centre creation, when needed, is delegated to the underlying software layers.

The LuCe system provides for two transparency layers: the Network Transparency Layer (NTL) and the Coordination Transparency Layer (CTL). The NTL enables inter-node network transparency, by hiding the physical location of tuple centres over the network and performing the necessary message routing between the network nodes. The CTL, instead, operates on the single network nodes, handling intra-node communications and implementing the illusion of ever-existing tuple centres by locally creating tuple centres when needed.

Since LuCe is meant for open, distributed systems, when an agent accesses a LuCe tuple centre the network latency time is predominant with respect to the tuple centre’s response time. This is why concurrency inside a tuple centre is not a major issue in LuCe. Concurrency is instead achieved within a single Tuple Centre Handler (TCH) process, which may host a multiplicity of tuple centres, each one implemented as a single thread. While multiple tuple centres may well be distributed over the network, a single LuCe tuple centre can not: this choice simplifies the implementation of the reaction execution model, since all the reactions triggered by a given communication event can be executed locally to the node hosting the involved TCH. Furthermore, each node of a LuCe system possibly hosts one TCH process, so that several TCH processes may run in parallel in the same LuCe system. Obviously, any running LuCe system always includes at least one node that actually hosts a TCH. What is relevant here is the inner architecture and the behaviour of the TCH process, with special regard to the support of the reaction model: in particular, the key point is what happens when a communication event occurs.

Critical issues

Most tuple space managers reported in the literature handle each communication event in an event-specific way, so as to serve each message properly and efficiently. Examples of this approach are the SICStus Library Linda server [32], the Linda Interactor server [33], and the York Kernel II [22].

In particular, most of these Linda run-time systems optimise the handling of the \texttt{out} operations in case there is a pending agent request for such a tuple, by sending the emitted tuple directly to that agent – without inserting the tuple into the tuple space at all. Another very frequent optimisation is that an \texttt{in} \,(t) \,or \texttt{rd}(t) \,primitive does not always cause the agent request to be queued: instead, if a matching tuple is already in the tuple space, the tuple is immediately removed and given to the agent, avoiding its query to be enqueued at all.

The above optimisations are natural and effective in the parallel programming field where
Linda originated, because they embed the fixed behaviour of the tuple space abstraction within very efficient closed implementations. However, they are less useful in distributed programming, where openness is a major requirement and focus is on expressiveness rather than on efficiency.

In particular, their main drawback is that they couple distinct communication events together, actually cancelling communication events. This behaviour should be avoided in a tuple centre server, where the observability of communication events constitutes the basic means for coordination medium programming. Actually, since reactions may change the observable effects of the communication operations, some classical cause/effect relationships may no longer hold: so, no assumptions should be made on the observable effects of a communication primitive.

For instance, an \texttt{out(p(a))} operation is no longer guaranteed to actually wake up a suspended \texttt{in(p(a))} operation, because a reaction may change the \texttt{p(a)} tuple to, say, \texttt{q(a)} or even remove it before it can be perceived:

\[
\text{reaction}\left(\texttt{out(p(X))}, \texttt{(in\_r(p(X)), out\_r(q(X)))}\right).
\]

If this is the case, the \texttt{p(a)} tuple, according to the agents’ perception, never appears in the tuple centre. In the same way, an agent performing an \texttt{in(p(X))} operation when there is one matching tuple in the tuple centre may have its request suspended anyway, if a reaction removes the matching tuple before it can be delivered to the agent. Similarly, an \texttt{in(p(X))} operation destined to be suspended because there are no matching tuples may not be actually suspended, if a reaction adds a matching tuple ‘just in time’ for it to be found and returned to the agent.

In short, a TCH must not take for granted that the observable effects of a standard communication event are known \textit{a priori}, because an intervening reaction may change them completely. So, each time a communication event occurs, the TCH must first serve it, and immediately after check the existing specifications and execute the associated reactions.

The TCH architecture

As a result of the above discussion, the architecture of the tuple centre handler can be specified according to the following three design constraints:

(I) (insertion constraint) an \texttt{out(t)} primitive must always result in tuple \texttt{t} being added to the tuple space;

(II) (query constraint) a \texttt{rd(t)}, \texttt{in(t)}, \texttt{rdp(t)} or \texttt{inp(t)} primitive must always cause the agent request to be queued;

(III) (reaction constraint) once served one communication event, the TCH must immediately execute all the associated reactions afterwards, before serving any other communication event.

These choices simplify the structure of the handler process, as handling the \texttt{out} primitive is now straightforward, while all the query primitives \texttt{rd}, \texttt{in}, \texttt{rdp} and \texttt{inp} can be handled in a uniform way.

More precisely, according to Constraint I, the tuple centre handler serves an \texttt{out(t)} communication event by simply inserting tuple \texttt{t} in the tuple centre, regardless of any pending request. According to Constraint II, the \textit{pre} phase of a query primitive is served by just queuing the corresponding agent request. The subsequent \textit{post} phase is handled by the TCH...
according to Constraint III by checking the queue and serving all the requests that can or must 
be satisfied: in the following, this step is called the wake-up step.

The above scheme guarantees that all the communication events are perceived by the TCH 
engine, and, therefore, can cause reactions to be executed if this is the case. Moreover, the 
query phase and the answer phase are now two well-separate communication events: each 
may trigger its own reactions, to be executed within the same transition of the tuple centre 
state. So, a new step, which is not present in conventional tuple space managers, is required: 
the serve_reactions step. This is where a TCH conceptually differs from conventional Linda 
tuple space managers.

If the TCH is built around a get_request/serve_request main loop, as in most of the Linda 
systems quoted, the insertion of the serve_reactions step is straightforward, and results in a 
pseudo-code like the following:

```c
while(true){
    get_request(R),
    serve_request(R),
    serve_reactions()
}
```

Here, get_request waits for and captures an incoming operation request $R$ – that is, a 
communication event crossing the TCH’s boundary inwards – along with all its properties. 
Then, serve_request serves the request as discussed above, just adding the tuple to the tuple 
space in the case of an out primitive (Constraint I), or queuing it in the case of a query 
primitive (Constraint II).

The wake-up step, instead, is subsumed by the new serve_reactions step, thus 
accomplishing Constraint III, and defaults to it in a standard tuple space manager [17]. This 
step implements the answer phase of all the query primitives in an enhanced way, serving 
all the reactions defined for the occurred communication event and then waking up agent 
requests according to the new situation.

Embedding wake-up within serve_reactions is essential because new reactions may be 
triggered within the wake-up step, too, and they must all be served before serving the next 
operation request. So, just adding the serve_reactions step before or after the wake-up step 
would be incorrect.

The event registry

The reaction model requires that the communication events are made observable. To do 
so, the TCH must be provided with an Event Registry, which records all the occurred 
communication events. Each communication event should cause a new entry to be added 
to the Registry, recording the operation and all its properties. Such properties include the 
operation name, the performing agent, the involved tuple, the involved tuple space, the 
operation phase (query phase or answer phase), its result, and the current ‘event time’ of 
the TCH. Since there may not be any global time notion in a distributed environment, the 
event time is obviously a local time, which is incremented each time a communication event 
occurs.

Accordingly, the in, rd, inp and rdp primitives cause this clock to be incremented twice – 
once in the query phase and once more in the answer phase of the operation – while out.
which is a single-phase primitive, increments it only once. Meanwhile, obviously, the TCH’s
clock may have been incremented other times, depending on the event history.

So, for instance, a \texttt{tc\_out(p(1))} operation performed by agent A causes an entry like
\texttt{(out,'A',p(1),tc,single,4)} to be added to this Registry, representing the only
phase of the \texttt{out} operation. In a similar way, a \texttt{tc\_in(p(X))} operation first causes an
entry like \texttt{(in,'A',p(X),tc,query,7)} to be added to the Registry, representing the
query phase of the \texttt{in} operation. Then, when the answer tuple (e.g. \texttt{p(6)}) is delivered to
agent A, a new entry like \texttt{(in,'A',p(6),tc,answer,success,9)} is added to the
Registry, modelling the answer phase of the \texttt{in} operation.

Tuple space operations performed inside reactions, that is \texttt{rd\_r}, \texttt{no\_r}, \texttt{in\_r} and \texttt{out\_r},
are handled in the same way, as they can trigger further reactions, too. However, since nothing
is crossing the TCH’s boundary, they do not correspond to actual communication events: so,
they do not cause the TCH’s inner time to be incremented, either. As a result, each transition
of the tuple centre state corresponds to one ‘macro-event’ and implies exactly one increment
of the tuple centre’s clock, according to the agents’ perception of the tuple centre evolution.

The \texttt{serve\_reactions} step

Reactions are handled in the \texttt{serve\_reactions} step of the TCH’s main loop. Conceptually,
this step is a loop made of two phases:

\begin{verbatim}
while ( \langle there are entries in the Registry \rangle ) {
  \langle choose one entry non-deterministically \rangle
  \langle execute all the reactions associated to that entry one by one, in a non-deterministic order \rangle
}
\end{verbatim}

If these reactions, in turn, access the tuple space, new entries may be generated: in any case,
the above loop continues until the Registry contains no entries. This condition states that the
transition of the tuple centre state triggered by the operation request is over. After that, the
\texttt{wake-up} step is executed.

The main issue with reaction execution is how to ensure that each reaction is handled as
a single transaction, with the typical ‘all-or-nothing’ semantics. Transactionality is achieved
by first executing the whole reaction body in a virtual environment, accepting the sequence
for real execution only if such a simulation succeeds. To do so, the \texttt{ReSpecT} virtual machine
operates on an image of the real tuple centre. If and only if virtual execution succeeds, the
reaction is actually carried out, executing its body goals as required by the reaction model
specifications [27], and recording the corresponding events in the Registry.

The \texttt{wake-up} step

According to the server architecture defined above, the \texttt{wake-up} step is carried out in two
phases:

1. if a matching tuple exists for one request, dequeue and serve this request;
2. if no matching tuple exists for any of the pending requests, dequeue one of the remaining
   \texttt{rd\_r/inp} requests, if any, and return a failure message to the agent.

In Phase 1, the TCH serves the \texttt{successful} requests, i.e. all the \texttt{rd/ln/rdp/inp} requests
for which a matching tuple is currently available. Then, Phase 2 takes care of serving the
failed \texttt{rdp} / \texttt{inp} requests. The \texttt{rd} / \texttt{in} requests for which no matching tuples are currently available remain pending in the request queue, according to the suspensive semantics of these primitives.

To carry out Phase 1, the TCH inspects one item in the request queue and checks for the availability of a matching tuple. If it finds one, it:

(a) sends such a tuple back to the agent,
(b) registers the occurred event in the Event Registry, thus manifesting the answer phase of the communication operation, and
(c) removes the tuple, if the served event was a tuple-consuming operation, i.e. \texttt{in} or \texttt{inp}, or leaves it there otherwise.

To carry out Phase 2, the TCH:

(a) registers the occurred event in the Event Registry, reporting the failure result of the operation, and
(b) sends the failure message to the performing agent as required.

Finally, if new entries have been added to the Registry as a result of performing the \textit{wake-up} step, the \textit{serve\_reactions} step is re-invoked recursively, so as to ensure that the corresponding reactions, if any, are triggered and executed before serving the next communication event. This is why the \textit{wake-up} step must be embedded within the \textit{serve\_reactions} step, and cannot simply be added before or after it in the server’s main loop.

**TCH implementation**

Our prototype implementation \cite{34} is built around SICStus Prolog \cite{35}, used as the kernel for the TCH. The vanilla SICStus system is empowered with an extra library to provide a Prolog interface to UDP sockets, network information and asynchronous UNIX signals.

Each tuple centre is mapped onto a separate SICStus Prolog module: this choice improves the system efficiency in locating predicates, as SICStus automatically introduces one hash table for each module. Modules have the same name as the mapped tuple centres, and include both the tuple space and the specification space of each tuple centre. The choice of representing tuple centres as SICStus modules makes the overhead of tuple centre creation almost negligible, as all that is required is to allocate a new Prolog module exploiting the SICStus efficient module implementation.

The tuple space and the specification space are kept logically distinct by storing the corresponding tuples under two different functors: \texttt{tuple(tuple,time)} for the tuple space, and \texttt{spec(tuple,time)} for the specification space, where \texttt{time} is the TCH’s local time when the tuple is stored.

The Event Registry is implemented using tuple-based technology, too: in particular, Registry entries are represented as \texttt{service/7} logic tuples in the built-in Prolog database. Obviously, the Registry is not a tuple centre, but just a simple tuple space: otherwise, each operation on the Registry would require another entry to be added to the Registry, and a loop would occur.

Finally, it is worth noting that tuple-based technology also makes it possible to check the pending request queue \textit{versus} the currently available tuples efficiently, by allowing the queue to be accessed associatively. To do so, the tuples involved in agent operations are recorded in another support registry, the \textit{Verify Registry}, implemented as \texttt{verify/7} logic tuples in the Prolog database. So, the pending request queue can be accessed based on the tuple recorded in the Verify Registry, improving efficiency with respect to a linear search.
PERFORMANCE EVALUATION

Since tuple-based coordination is founded on the assumption that inter-agent communication occurs only via tuple repositories, the communication load is a relevant efficiency parameter. The tuple centre concept can give a relevant contribution from this viewpoint, because a tuple centre intrinsically requires fewer messages than a conventional tuple space to achieve a similar behaviour. So, our claim is that the gain brought by tuple centres in reducing the communication load overcomes and compensates the extra computational load of reaction handling.

To put this statement to the test, the Dining Philosopher problem discussed in the previous sections was used to perform three classes of experiments: (a) standard tuple space coordination, (b) tuple space coordination with a deadlock-free agent interaction protocol, and (c) tuple centre coordination. The first class is just a mere reference, since the resulting system does not feature the required properties. The second and the third class, instead, represent the two different approaches to be actually compared: that is, adding global properties to a multi-agent system by either changing the agents’ interaction protocol (b), or changing the behaviour of the coordination medium (c).

Design advantages

From the design viewpoint, the advantages of tuple centre coordination clearly emerge from the simple comparison of the two chunks of pseudo-code shown in Table IV, abstracted from the actual code of C agents. Since the tuple space cannot provide for deadlock avoidance in chopstick acquisition and fairness in chopstick release, philosophers are required to adopt a locking protocol, using for instance a semaphore tuple to be acquired before actually trying to get chopsticks, and released when done. This requirement actually prevents the separation between computation and coordination, promoted by tuple-based coordination models at the language level, from being lifted up to the design level, where the two issues are improperly merged (see Table IVb). Obviously, hiding the locking protocol into a procedure would not solve the problem, since computation and coordination would still be improperly merged in the agent’s procedure code, and agents would still have to take care of coordination themselves.

Instead, tuple centres can be charged of the coordination load, letting agents free to interact at the most suitable abstraction level, by embedding coordination laws into the behaviour specification (see Table III). Here, the most suitable agents’ viewpoint is to view chopsticks as pairs to be acquired/released, while still representing chopsticks singly. Computation and coordination issues are then neatly separated in the design of the philosopher agent, as shown in Table IVc.

Estimate of communication cost

Tuple centre coordination also improves the overall system performance, by reducing the global communication load. In fact, in the first test (Table IVa) a philosopher agent willing to eat just gets the two chopsticks it needs, which requires two in operations. After eating, such chopsticks are released by means of two out operations. So, each think/eat cycle costs four tuple space operations. In the second test (Table IVb), which takes deadlock avoidance into account, the chopstick acquisition/release phases consist of three operations each, since the semaphore tuple must be handled, too. As a result, a think/eat cycle costs six tuple space operations.
Table IV. Tuple space vs. tuple centre: pseudo-code for a C philosopher

<table>
<thead>
<tr>
<th>Tuple space</th>
<th>Tuple space (deadlock-free)</th>
<th>Tuple centre</th>
</tr>
</thead>
<tbody>
<tr>
<td>rd(philo(me,C1,C2)); while (true) { /* main cycle <em>/ think(); /</em> acquisition <em>/ in(chop(C1)); in(chop(C2)); eat(); /</em> release */ out(chop(C1)); out(chop(C2)); }</td>
<td>rd(philo(me,C1,C2)); while (true) { /* main cycle <em>/ think(); /</em> acquisition <em>/ iCanEat=false; while (!iCanEat) { in(token); if (!inp(chop(C1))) if (!inp(chop(C2))) out(chop(C1)); else iCanEat=true; out(token); } eat(); /</em> release */ out(chop(C1)); out(chop(C2)); out(token); }</td>
<td>rd(philo(me,C1,C2)); while (true) { /* main cycle <em>/ think(); /</em> acquisition */ in(chops(C1,C2)); }</td>
</tr>
</tbody>
</table>

operations at least, that is, without taking into account philosopher’s failures in getting both chopsticks after obtaining the semaphore tuple.

Things change completely in the third case, where the separation between information representation of single resources and information perception as resource pairs enables agents to be designed according to a very simple two-phase protocol. Owing to the transactional semantics of the reaction model, atomicity is guaranteed: deadlock is prevented a priori, and no extra protocols are needed. So, a think/eat cycle costs just two tuple centre operations – the very minimum (see Table IVc).

In a distributed environment, the agent execution time is due, to a large extent, to network communication: the cost of additional computation is typically negligible. In fact, reactions are executed locally, and do not imply any communication operation over the network, since out_r, in_r, etc., are not communication operations. So, reaction execution time should not be relevant when compared to network latency time. For this reason, tuple centre coordination can be expected to notably improve the agent and system performance with respect to simple tuple space coordination.

Test of communication cost

To put this statement to the test, the philosopher agents were implemented both in C and in Java, exploiting the C libraries and Java packages developed ad hoc for the LuCe system. The experimental environment consisted of a set of heterogeneous philosopher agents, three C agents and three Java agents, eating at random times for 50 times each before leaving the system: the test was repeated 20 times, taking the average values. How long each agent thinks
or *eats* (see Table IV) is set every time randomly, taking care to keep the agent execution
time still negligible with respect to communication time: so, the global multi-agent system
execution time should mainly measure the cost of inter-agent communication. Results are
reported in Figure 1, where values are normalised to 1.0 for the first test. These figures
follow, with good approximation, the cost estimate made in the previous Subsection: in
particular, the tuple centre approach shows a clear performance improvement over the second
test – the one ensuring deadlock avoidance, too – despite the additional computational charge
of reaction execution. So, the adoption of tuple centre coordination significantly improves
the performance of this coordinated system, by roughly halving the communication cost.
Furthermore, the results suggest that the cost of transactionality should not affect significantly
the global system performance in this case. Generally speaking, the adoption of a single,
possibly huge tuple centre for the coordination of a complex multi-agent system may actually
have a negative impact on the cost of communication. However, it seems reasonable to
expect that exploiting a multiplicity of reasonably-sized tuple centres should keep the cost
of transactionality low enough to be disregarded in practice.

These values are even more significant because all test were made on a single host, so
as to minimise the effect of the network load: in a truly distributed system, the cost of
communication operations is likely to be higher, making the tuple centre approach even more
effective. Moreover, it should be considered that the system used for the tests was developed
without having efficiency as the first goal in mind, so a re-engineered implementation is likely
to show better performance, too.

**RELATED WORKS AND CONCLUSIONS**

The effectiveness of tuple-based coordination led to the development of several coordination
architectures [22] in different contexts. While the Linda model [7] was originally conceived
for parallel architectures [36], tuple spaces proved to be effective to build distributed object-ored systems [18], in particular on the World Wide Web: see, among the others, PageSpace [37], WCL [38], and Lime [39]. On the same line, new industry standards are emerging, like JavaSpaces [9] and T Spaces [10].

Within this framework, this paper presented the LuCø architecture for open and
heterogeneous multi-agent systems, which subsumes the Linda model by defining a clean
relationship between agent requests and their intended effects over tuple spaces, as well as a
simple general scheme to handle such requests. In order to address the limits of tuple-based
coordination in terms of flexibility and control, LuCø tuple centres can be programmed so as to
rule agent interaction, while still being perceived by agents as standard tuple spaces. The LuCe run-time kernel makes all communication events observable and supports the execution of reactions associated to such events, by exploiting a uniform representation of communication data, reaction specifications, and communication events in the form of logic tuples.

Law-governed Linda [40], too, exploits programmable tuple spaces for security and efficiency in distributed systems, where programmable controllers are associated locally to each agent to intercept and modify the behaviour of any communication operation. However, the emphasis on locality makes it difficult to express global coordination policies. In T Spaces [10], agents can add new primitives or overload old ones to implement any kind of transaction they need on the data stored in the tuple space. However, adding new primitives seems a poorly suited approach for open systems, since it enforces the coupling between the interacting entities, which have to be made aware of the new primitives in order to exploit them. While overloading seems a better solution, the T Space approach is perhaps too coarse-grained when compared to the LuCe one. In LuCe, primitives are not simply re-defined, but can be extended selectively according to any property of the communication events triggered by any standard communication primitive, thus providing for a more fine-grained control.

The TuCSoN coordination model [41] relies on the same notion of tuple centre as LuCe, and extends it so as to address the issue of agent mobility, by enabling the definition of local interaction spaces. For this reason its implementation is based on the LuCe architecture presented in this paper, where the Network Transparency Layer is replaced by the support for remote communication.

The MARS coordination model [42], too, adopts programmable tuple spaces for mobile agent coordination. Unlike LuCe and TuCSoN, however, it exploits an object-oriented tuple space model that makes it more suitable to service and network management applications. Further work is being devoted to extend the LuCe architecture so as to integrate MARS’ programmable tuple spaces.

ACKNOWLEDGEMENTS

This work has been carried out under the financial support of the MURST (the Italian ‘Ministero dell’Università e della Ricerca Scientifica e Tecnologica’) in the framework of the MOSAICO Project (Design Methodologies and Tools of High Performance Systems for Distributed Applications).

The authors would like to thank Franco Zambonelli for his many remarks on an early version of this paper, and the anonymous referees for their helpful suggestions.

REFERENCES
