Multi-paradigm Coordination for MAS: Integrating Heterogeneous Coordination Approaches in MAS Technologies

Stefano Mariani
DISI, Alma Mater Studiorum–Università di Bologna
via Sacchi 3, 47521 Cesena, Italy
Email: s.mariani@unibo.it

Andrea Omicini
DISI, Alma Mater Studiorum–Università di Bologna
via Sacchi 3, 47521 Cesena, Italy
Email: andrea.omicini@unibo.it

***Abstract*—Open distributed multi-agent systems featuring autonomous components demand coordination mechanisms for both functional and non-functional properties. Heterogeneity of requirements regarding interaction means and paradigms, stemming from the diverse nature of components, should not affect the effectiveness of coordination. Along this line, in this paper we share our pragmatical experience in the integration of objective and subjective, synchronous and asynchronous, reactive and pro-active coordination approaches within two widely-adopted agent-oriented technologies (JADE and *Jason*), enabling coordinating components to dynamically adapt their interaction means based on static preference or run-time contingencies.**

I. Introduction

Open and *distributed* multi-agent systems (MAS) featuring *autonomous* components demand efficient coordination mechanisms for both functional and non-functional properties. *Heterogeneity* of requirements regarding *interaction* means and paradigms – e.g. message-passing vs. shared space, synchronous vs. asynchronous, proactive vs. reactive, etc. – stemming from the heterogeneous nature of components – e.g. software agents vs. physical devices – implies that coordination should be effective despite heterogeneity.

Coordination approaches in the literature can be classified according to diverse criteria, such as the *means* through which interaction happens – e.g. messages vs. tuples –, *synchronism* of interaction primitives, enforced *programming style*—e.g. reactive, such as with actors, vs. pro-active, such as with agents. The most general classification distinguishes coordination approaches as either *objective* or *subjective* [1], depending on *who* is responsible for the "burden of coordination": the *coordinable*, or the *coordination medium*—following the meta-model and the terminology proposed in [2].

Objective coordination [1] refers to the approach – typically stemming from the software engineering research field – where coordination-related concerns are *encapsulated* within dedicated abstractions, such as *coordination artefacts* [3] (the medium), offering *coordination as a service* to coordinables [4]. Thus, coordination abstractions take responsibility for coordination, by steering agent *societies* – that is, ensemble of cooperating/competing agents – towards the achievement of social (shared) goals, through the suitable management of the *dependencies* between agents' activities [5], [6]—despite agent's individual goals, yet without harming their autonomy.

Subjective coordination [1] represents instead the dual approach – typically adopted in the (distributed) artificial intelligence field [7] – where coordination issues are *directly* tackled by the agents themselves, determining their best course of (inter-)action in the attempt to achieve their own goals. Thus, the burden of coordination is here distributed among coordinables, expressing their autonomy w.r.t. coordination commitments through their own *deliberation* process.

Clearly, respecting agents' *autonomy* is a foremost issue in the design of coordination models for MAS, be them objective or subjective. Also, the issue of autonomy emerges while designing other facets of coordination model and technologies, corresponding to the aforementioned classification criteria.

Enforcing a *reactive* programming style for handling interactions – as typically done in the context of event-based systems and actor-based frameworks – often leads to the problem of *inversion of control*—namely, the fact that the flow of interactions determines *when* the interacting components should act and *what* they should do, thus definitely wipe out *pro-activeness* of components. Also, on a lower level, the management of the component control flow is taken out from the programmers' hands, and left to the framework. In the overall, inversion of control results in a blatant example of violation of the component autonomy.

Adopting a *synchronous* semantics for interaction primitives invocation, without adequate mediating abstractions *uncoupling* the invoker flow of control from the (possibly) *suspensive* semantics of the primitive, is another way to violate the components autonomy. A typical example can be observed in any LINDA-like coordination model [8], where interacting components are usually *bound* to withstand the outcomes of the coordination process—e.g. getting stuck whenever a getter primitive finds no matching tuple in the shared space.

Accordingly, in this paper we share our pragmatical experience in integrating objective and subjective, synchronous and asynchronous, reactive and pro-active coordination approaches, enabling participating components to (deliberatively) dynamically adapt their interaction means based on static preference or run-time contingencies – while still having their autonomy preserved –, in the context of two widely adopted agent-oriented technologies, that is, JADE [9] and *Jason* [10], featuring their own subjective coordination mechanisms, and adopting the TuCSoN middleware as the provider of objective coordination. The goal is to provide MAS designers and programmers with a *flexible* development framework capable of seamlessly supporting *multi-paradigm coordination*, where agents deliberatively choose which coordination paradigm and interaction means to adopt for the current interactions—and, more importantly, where such a choice does not affect negatively the other interactive, or non-interactive, activities they are (possibly, concurrently) undertaking.

The remainder of the paper is organised as follows. Section II describes how the subjective coordination paradigm may be instantiated within a message-based setting, also wr.t. synchronism and programming style, by discussing the concrete cases of JADE (Subsection II-A) and *Jason* (Subsection II-B). Section III describes how the objective coordination paradigm may be implemented within a tuple-based setting, by discussing the TuCSoN approach (Subsection III-A). Section IV describes our solution for enabling multi-paradigm coordination, by briefly describing TuCSoN4JADE (Subsection IV-A) – already presented in [11], but useful here for comparison and generalisation purposes – and by thoroughly discussing TuCSoN4*Jason* (Subsection IV-B)—also, an example showcasing effectiveness of multi-paradigm coordination is provided in Subsection IV-C. Finally, Section V briefly reports on some similar works by fellow researchers, while Section VI summarises the work, and outlines future improvements.

II. Message-based Subjective Coordination

In this section we briefly describe the approach to subjective coordination adopted by two agent development frameworks well-known within the MAS community: JADE (Subsection II-A) and *Jason* (Subsection II-B).

A. *Subjective Coordination in* JADE

JADE (Java Agent DEvelopment Framework) [9] is a Java-based framework and infrastructure to develop and deploy open, distributed, agent-based applications in compliance with FIPA standard specifications for interoperable intelligent MAS. In JADE, autonomy of agents is supported by the *behaviour* abstraction, whereas coordination adopts the subjective paradigm thanks to an *asynchronous message-passing* layer – the *Agent Communication Channel* (ACC) – on top of which FIPA interaction protocols are implemented and provided to developers—as a *callbacks framework*.

A behaviour can be logically interpreted as “an activity to perform with the goal of accomplishing a task”. Thus, different courses of actions of a JADE agent are encapsulated into distinct behaviours the agent executes simultaneously. More technically, JADE behaviours are Java objects, which are executed *pseudo-concurrently* within a single Java thread by a built-in *non-preemptive round-robin scheduler*. During JADE agent initialisation, behaviours are added to the *ready queue*, ready to be scheduled. Then, method `action()` of the first behaviour – implementing one of the agent’s course of action – is executed. Behaviours switch occurs only when the method returns; hence, in the meanwhile *no other behaviour can start execution* (non-preemptive scheduler). Behaviours removal from the ready queue occurs only when the `done()` method returns `true`; otherwise, the behaviour is re-scheduled at the end of the queue (round-robin scheduler). It is worth noting that method `action()` is executed *from the beginning every time*: there is no way to “stop-then-resume” a behaviour during its execution.

The ACC is the middleware service in charge of *asynchronous message passing* among agents: each agent has its own mailbox where incoming communications are silently put, waiting to be explictly (*pro-actively*) considered by the receiving agent—for the sake of preserving its autonomy. *When* and *how* to process the mailbox is up to the agent’s own deliberation:

- method `receive()` attempts to *asynchronously* retrieve the first message, or a message compliant with a given message template, from the mailbox—returning nothing in case no message is found
- method `blockingReceive()`, as the name suggests, attempts to *synchronously* retrieve the first message, or a message compliant with a given message template, from the mailbox—*waiting* until a message is eventually found

Clearly, the first method preserves the autonomy of the agent no matter what. The same does not hold for the second one, because deciding to wait for a specific interaction potentially *hinders* the ability of the agent to remain responsive to other interactions—although based on a deliberate choice of the agent itself.

The reason is that method `blockingReceive()` suspends *the agent as a whole*, not only the calling behaviour. Thus, the JADE Programmers Guide itself [12] suggests the adoption of the following programming pattern: call `receive()` instead of `blockingReceive()`, then call method `block()` – of the `Behaviour` class – if no message is found, so as to suspend *only the calling behaviour*—which will be automatically resumed by JADE upon reception of any message. Therefore, if programmers want their agents’ autonomy preserved while subjectively coordinating, they should abide to the “block-then-resume” pattern just described.

Besides plain message passing, JADE provides to programmers a number of ready-to-use FIPA interaction protocols, such as the Contract Net [13] and the general-purpose Achieve Rational Effect [14], in the form of a callbacks framework:

- programmers declare which FIPA protocol their interacting agents should commit to, by instantiating the corresponding Java object
- then, they implement the actual body of the callback methods corresponding to each interaction (message to be received or sent) as expected according to the specific FIPA protocol adopted

As any other framework adopting a callback scheme, JADE FIPA protocols promote a reactive programming style, where agents synchronously wait for messages to advance through the protocol stages. Agents’ pro-activeness – so, their autonomy – is thus limited. Nevertheless, JADE internally implements FIPA protocols abiding to the block-then-resume pattern, so that interacting agents stay responsive to concurrent interactions while subjectively coordinating—besides being able to carry out non-interactive activities through concurrent behaviours.

Summing up, JADE support to subjective coordination is provided through asynchronous message passing – respecting agents' autonomy thanks to the block-then-resume pattern – and built-in FIPA protocols—limiting instead autonomy due to reactiveness enforced by the callbacks framework.

B. *Subjective Coordination in Jason*

Jason [10] is both an agent language and an agent runtime system. As a language, it implements a dialect of AgentSpeak [15]; as a run-time system, it provides some services needed to execute a MAS—e.g. agents' reasoning engine and a non-distributed execution platform. Although *Jason* is entirely programmed in Java, it features BDI agents, so a higher-level language (the *Jason* language) is used to program *Jason* agents using BDI abstractions. In *Jason*, autonomy of agents is supported by *plan/intention* concurrent execution machinery, whereas coordination adopts the subjective paradigm[1] thanks to the asynchronous message passing layer.

Similarly to JADE behaviours, a *Jason plan* can be conceptually interpreted as a course of action to be performed to accomplish a task. Technically, a *Jason* plan differs considerably from JADE behaviours: *(i)* it is scheduled for execution as soon as a *triggering event* occurs, *(ii)* it is not directly executed "as is" (in general), but is instantiated as an *intention*, then executed, *(iii)* intentions are pseudo-concurrently executed *one action each*, according to a round-robin scheduler. Therefore, whereas in JADE the behaviour is the basic execution step, in *Jason* the same role is played by the single action, not by the plan/intention as a whole. Intentions may be *suspended* – either by the programmer or by the *Jason* reasoning engine – for a number of reasons—e.g. because the agent needs to wait for a message, or an environmental event.

Jason agents can in fact exchange beliefs (but also plans and goals), both synchronously and asynchronously through the same `.send()` internal action, in the form of messages encoded in a KQML-like language, with a very similar semantics. Depending on the *illocutionary force* determining the meaning of messages (e.g., `askOne` vs. `tell`), their reception causes different events (e.g., test-goal addition vs. belief addition) automatically triggering execution of the correspondent plans. It is worth noting that there is no explicit *receive* communication primitive involved, here, nor a mailbox to explicitly monitor[2]. The basic illocutionary forces concerned with exchanging beliefs are:

- `tell`, to share a (list of) belief(s) with a target (list of) agent(s). Addition of the communicated information to the belief base of the receiver agent, where it waits to be *pro-actively* considered, causes a *belief addition event*, handled by plans starting with the `+b` clause
- `askOne` to query another agent's belief base—sort of a "remote test-goal"[3]. The internal action `askOne` comes in three flavours:
    - *asynchronous*, where the caller intention never gets suspended, and the query result has to be handled reactively through a proper plan handling the belief addition event `+b`
    - *synchronous*, where the caller intention gets suspendend until the receiver's reply becomes available—that is, a matching belief appears in its belief base
    - *timed synchronous*, where the synchronism is limited by a finite upper bound on waiting time, whose expiration causes the intention to be resumed—and labelled with a `timeout` outcome

[1]While objective coordination could in principle be implemented on top of *Jason* `Environment` Java API for programming MAS (computational) environments, it is not considered here since no first-class support for environment abstractions is provided.

[2]Actually, a mailbox exists, but the *Jason* agent is completely unaware of it.

[3]Actually, a test-goal is executed on the receiver agent side, thus, if plans starting with the `+?b` clause exist, they are triggered to handle the request.

It is worth noting that intentions only (not the whole *Jason* agent) are automatically suspended whenever they perform a (communication) action which cannot complete – e.g. the request for information following a test-goal attempt –, to be resumed as soon as the action obtains its "completion feedback" (see [10], page 86)—e.g. the message conveying the requested information. This mechanism may be interpreted as the equivalent of JADE block-then-resume pattern, which is automatically handled here by the *Jason* runtime.

Summing up, *Jason* support to subjective coordination is provided through both asynchronous and synchronous message passing, respecting agents' autonomy thanks to the intention suspension mechanism—unlike JADE, where `blockingReceive()` suspends the agent as a whole.

III. ARTEFACT-BASED OBJECTIVE COORDINATION

In this section we briefly describe the approach to objective coordination adopted by the TuCSoN coordination model and technology, taken as a reference for artefact-based coordination and exploited in Section IV as the provider of objective coordination services.

A. *Objective Coordination in* TuCSoN

TuCSoN [16] is a Java-based, (logic) tuple-based coordination model and infrastructure for open, distributed MAS. It provides *objective coordination as service* by implementing and extending the LINDA model [8] with ReSpecT *tuple centres* [17], the *coordination artefacts* [3] encapsulating coordination mechanisms and policies, distributed over a network of TuCSoN nodes.

TuCSoN preserves interacting agents' autonomy through the *Agent Coordination Context* (ACC) [18] abstraction—mainly; TuCSoN architecture also contributes, as clarified below by describing TuCSoN "two-steps" execution. ACC are assigned to agents as they enter a TuCSoN-coordinated MAS, with the responsibility, among the many others, to *uncouple* the *synchronism* of a coordination operation invocation from the *suspensive semantics* of the coordination primitive used – e.g. LINDA `in` –, by mapping operations to *events asynchronously* dispatched to the coordination media (ReSpecT tuple centres).

More precisely, the suspensive semantics of getter primitives, as inherited by the LINDA model, implies that if no matching tuple is found the operation waits indefinitely until it becomes available. This semantics does not depend on agent own deliberation: it is set by the coordination model at hand, and is what enables distributed synchronisation and coordination, in the very hand. The *synchronism* of invocation instead, couples (or uncouples) the fate of a coordination operation with that of the invoker agent, and is subject to the agent own deliberation. This means that:

- in case the agent chooses the synchronous invocation mode, it gets suspended, too, if the operation gets suspended
- if the asynchronous invocation mode is used, instead, suspension policy is confined to the operation, whereas the invoker agent can continue performing its activities, pro-actively inspecting the status of the operation when preferred—and eventually getting the result

In order to do so, the execution of any TuCSoN operation follows two steps:

- the *request* to carry out a given coordination operation is performed by the agent through its ACC, which then dispatches the corresponding operation event to the target tuple centre—*invocation* step
- the *response* to the coordination operation invoked is sent back to the requesting agent by the tuple centre through its ACC, as soon as the response is available—*completion* step

Put in other terms, any coordination service provided by TuCSoN is *asynchronous by default*; nevertheless, interacting agents may deliberately choose the invocation synchronism of each coordination operation—potentially limiting their autonomy: either synchronous or asynchronous.

Summing up, TuCSoN coordination model preserves agent autonomy by uncoupling the suspensive semantics of coordination operations from their invocation semantics (synchronism) – actually promoting pro-activeness over reactiveness – thanks to the ACC architectural abstraction.

IV. Enabling Multi-paradigm Coordination

In this section we describe two software modules enabling TuCSoN-based multi-paradigm coordination for JADE and *Jason* agents, namely, TuCSoN4JADE (Subsection IV-A) and TuCSoN4*Jason* (Subsection IV-B) respectively, emphasising the technical solutions adopted to preserve conceptual integrity of the resulting development framework—especially, w.r.t. autonomy of agents. Subsection IV-C provides an example about how a widely spread and well-known coordination scenario, that is, a Contract Net Protocol based interaction, may benefit from a multi-paradigm coordination approach.

A. *The case of* TuCSoN*4*JADE

An *autonomy preserving* integration of JADE and TuCSoN was already presented in [11], with an emphasis on the impact that technical choices regarding concurrency and communication aspects have on the opportunities for objective and subjective coordination integration—also accounting for the conceptual issues of autonomy of interacting agents. There, the main issue was making TuCSoN synchronous invocation semantics compatible with JADE concurrency model based on behaviours.

The simplest and most straightforward solution would be letting JADE agents directly access TuCSoN services, requiring nothing more than a correct usage of TuCSoN API—since both technologies are implemented on top of Java. Nevertheless, direct API access would inevitably hinder JADE agents autonomy while resorting to objective coordination synchronously: their control flow would be coupled to that of the coordination operation requested, so that in case of suspension the whole agent would get stuck, not just the invoker behaviour. This way, roughly speaking, agents' deliberate choice to exploit objective coordination could negatively affect their ongoing subjective coordination activities, as well as any other non-interactive activity in process. This would represent a glaring example of an *artificial dependency* (unintentionally) created by a "non-autonomy-preserving" approach to multi-paradigm coordination.

Instead, an appropriate integration between TuCSoN and JADE was achieved in [11] by developing a bridge component in charge of suitably suspending and resuming JADE behaviours depending on the outcome of the requested TuCSoN coordination operation—besides providing a suitable API wrapping TuCSoN services to JADE agents: the TuCSoN4JADE bridge[4].

In particular, the bridge is based on the fact that whenever any JADE behaviour is re-scheduled its `action()` method is re-started, thus the synchronous invocation of TuCSoN coordination operation is repeated. The whole TuCSoN4JADE machinery works because the invocation – through TuCSoN4JADE API method `synchronousInvocation()` – transparently checks whether the operation just invoked can be completed: in case the operation result is not available, yet, the request is actually dispatched by TuCSoN ACC to the target tuple centre, and the invoker behaviour is suspended by calling method `block()`. As soon as the operation result becomes available, the bridge component resumes the suspended behaviour and hands off the completion event—so that the second invocation, due to behaviour restart, immediately succeeds.

Summing up, TuCSoN4JADE lets JADE agents free to choose, for every single interaction event, which coordination paradigm suits their needs best, based on both design-time preference and run-time contingencies, without possibly hindering other ongoing coordination activities.

B. *The case of* TuCSoN*4Jason*

In the case of *Jason*, the main integration issue is to make TuCSoN synchronous invocation semantics compatible with *Jason* concurrency model based on intentions. In fact, simply wrapping the TuCSoN API into suitable *Jason* abstractions, such as internal actions, would inevitably hinder *Jason* agents autonomy while resorting to objective coordination synchronously: their control flow would be coupled to that of the coordination operation requested, thus in case of suspension the whole agent would get stuck, not only the invoker intention.

Yet again, agent deliberate choice to exploit objective coordination negatively affects their ongoing subjective coordination activities, thus resulting in another failed attempt

[4]http://apice.unibo.it/xwiki/bin/view/TuCSoN/4JADE

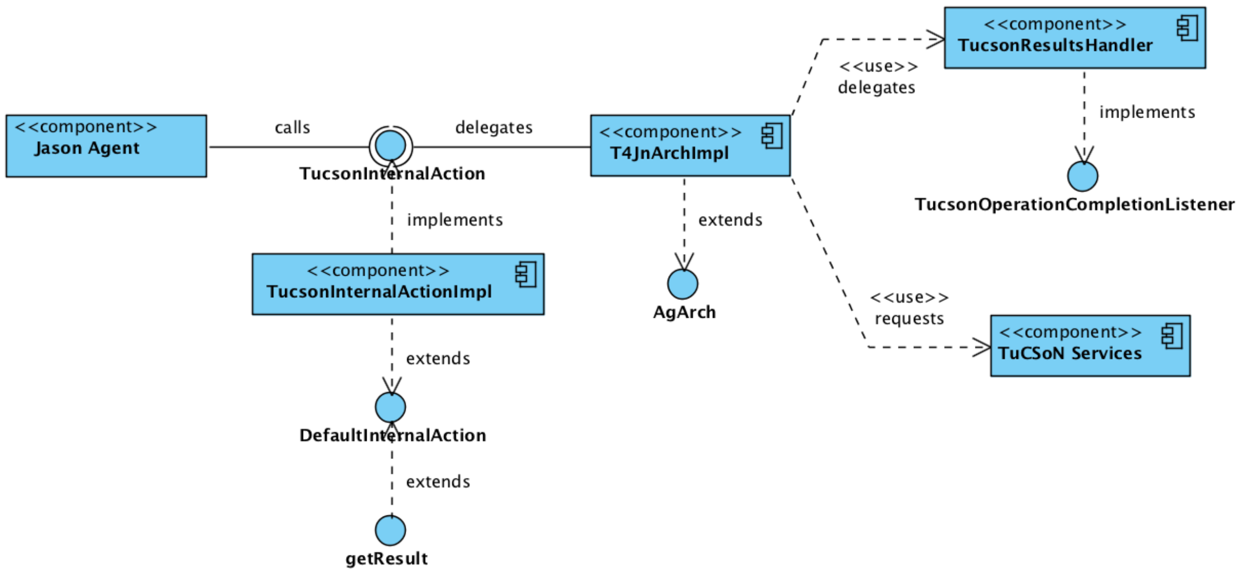

Figure 1: TuCSoN4*Jason* architecture

to fully enable multi-paradigm coordination. Thus, an adequate integration between TuCSoN and *Jason* requires a fine-grained integration of TuCSoN two-steps execution mechanism with *Jason* intention suspension mechanism, properly suspending/resuming intentions based on coordination operations outcome: the TuCSoN4*Jason* bridge is the Java library providing such integration[5].

First of all, we analyse the integration solution provided by TuCSoN4*Jason* from the architectural point of view, highlighting its main components—Figure 1:

- a *custom agent architecture* enabling customisation of each *Jason* agent inner reasoning engine
- an *operation completion listener* enabling TuCSoN to couple *Jason* intention suspension mechanism with its invocation semantics
- a set of *custom internal actions* enabling *Jason* agents to request TuCSoN coordination services and inspect operations outcome

The custom agent architecture, implemented by class `T4JnArchImpl`, is responsible for dispatching coordination operation requests from *Jason* agents to the TuCSoN service, keeping track of pending as well as completed operations (in collaboration with the other components), and coordinating the other TuCSoN4*Jason* components.

The operation completion listener, implemented by class `TucsonResultsHandler`, is responsible for handling notifications of operations completion coming from TuCSoN, for tracking them within the custom agent architecture, (automatically) resuming suspended intentions when their result becomes available, and unifying TuCSoN tuples (the operation results) with *Jason* variables.

[5]http://apice.unibo.it/xwiki/bin/view/TuCSoN/4Jason

The set of internal actions, implemented by the public classes within package `t4jn.api` extending `TucsonInternalActionImpl`, is responsible for providing *Jason* agents with the means to request TuCSoN coordination services, that is, basically, the set of available coordination primitives. Besides the usual tuple space primitives such as `out`, `rd`, and `in`, primitive `getResult` deserves special attention: it is responsible for uncoupling the request of a coordination operation from its completion, handling (automatic) intention suspension when the result of a getter primitive (such as `in`) is not yet available.

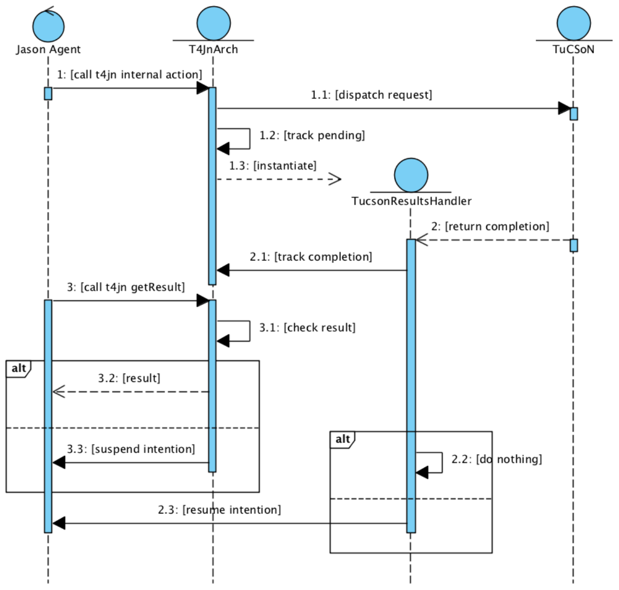

Figure 2: TuCSoN4*Jason* flow of interactions

Collaboration between components – as depicted in Figure 2 – makes TuCSoN4*Jason* bridge capable of enabling autonomy-preserving multi-paradigm coordination for *Jason* agents, by carefully coupling *Jason* intention suspension mechanism with TuCSoN two-step execution of suspensive primitives.

Whenever a TuCSoN operation is requested (message 1), through *Jason* custom internal actions, the bridge component – in particular, the custom agent architecture represented by interface `T4JnArch` – takes care of asynchronously dispatching the request to TuCSoN (message 1.1) by exploiting TuCSoN services devoted to asynchronous operation invocation—not depicted to keep the picture simple. Then it tracks the pending state of the requested operation (message 1.2) and spawns a dedicated handler for its completion (message 1.3)—`TucsonResultsHandler`. As soon as the operation is ready to be completed, the handler is notified by TuCSoN with the result (message 2), so that the state of the corresponding operation can be changed from "pending" to "completed" (message 2.1), and, if needed, resume the suspended intention within which TuCSoN4*Jason* `getResult()` internal action was called (message 2.3, Figure 4).

It is worth noting that the notification may happen at anytime, without prior knowledge on whether `getResult()` has either been called or not. Dually, from the *Jason* agent standpoint, there is no prior (design-time) knowledge on whether the operation succeeded until `getResult()` is (synchronously) called.

In fact, any *Jason* custom internal action has an asynchronous invocation semantics, that is, it does not wait for operation completion, so that the agent is free to choose when to ask for it whenever it pleases (message 3)—deliberatively. When doing so, the TuCSoN4*Jason* bridge checks if the operation result is available (message 3.1): if it is, the agent gets it and the current intention can proceed (message 3.2); otherwise, the calling intention gets suspended (message 3.3, Figure 3), until the result becomes available.

Figure 3 shows a code excerpt taken from the `getResult()` custom internal action implementation. There, TuCSoN4*Jason* manipulates the state of the *Jason* reasoner behind any *Jason* agent – thanks to `T4JnArchImpl` extending *Jason* `AgArch` – so as to make it suspend the intention within which `getResult()` was called, in case no result is available yet. In particular, line 4 retrieves from the reasoner the current execution context (`Circumstance`), line 6 gets the current intention (the one where `getResult()` was called), lines 8 – 13 actually suspend the intention.

Figure 4, instead, shows a code excerpt taken from the `TucsonResultsHandler()` component implementation. There, TuCSoN4*Jason* manipulates the state of the *Jason* reasoner so as to make it resume the intention within which `getResult()` was called, as soon as its result becomes available. In particular, line 4 tracks that such an intention is no longer pending, line 8 retrieves from the reasoner the current execution context, lines 10 – 15 iterate over pending intentions according to the reasoner state, line 17 finds the intention to be resumed among the ones being iterated over, lines 18 – 21 actually resume the intention.

Summing up, TuCSoN4*Jason* lets *Jason* agents free to choose, for every single interaction event, the coordination paradigm that best suits their needs, based on both design-time preference and run-time contingencies, without hindering any other ongoing coordination activities.

```
1  ...
2  // result NOT available
3  if (!(results.containsKey(actionId))) {
4      Circumstance c = ts.getC();
5      // suspend calling intention
6      Intention i = c.getSelectedIntention();
7      this.suspendIntention = true;
8      i.setSuspended(true);
9      c.addPendingIntention(
10         jason.stdlib.suspend.SELF_SUSPENDED_INT
11         + i.getId(), i);
12     // track suspended intention
13     arch.getSuspendedIntentions().put(actionId, i);
14     mutex.unlock(); // thread-safety
15     return true;
16 }
17 ...
```

Figure 3: Suspension of a *Jason* intention, due to `getResult()` being called with no result available yet

```
1  ...
2  // corresponding action caused suspension
3  if (this.suspendedIntentions.containsKey(this.actionId)) {
4      Intention suspendedIntention = this.suspendedIntentions
5          .remove(this.actionId);
6      this.mutex.unlock(); // thread-safety
7      ... // parse result
8      Circumstance c = this.ts.getC();
9      // scan pending intentions
10     Iterator<String> ik = c.getPendingIntentions().keySet()
11         .iterator();
12     while (ik.hasNext()) {
13         String k = ik.next();
14         if (k.startsWith(suspend.SUSPENDED_INT)) {
15             Intention i = c.getPendingIntentions().get(k);
16             // find corresponding intention
17             if (i.equals(suspendedIntention)) {
18                 i.setSuspended(false);
19                 ik.remove();
20                 ... // other reasoner-related stuff
21                 c.resumeIntention(i);
22             }
23         }
24     }
25 }
26 ...
```

Figure 4: Resuming a previously suspended intention to due its result becoming available

Subjective coordination based on asynchronous message passing, handled with a reactive programming style based on belief addition events ($+b$), can be seamlessly and simultaneously exploited by *Jason* agents while they are objectively coordinating based on either

- synchronous TuCSoN operations, emulated by following an operation invocation – asynchronous by deafult – with an immediate call to `getResult()`
- asynchronous TuCSoN operations handled adopting a pro-active programming style, e.g. calling `getResult()` as part of the *Jason* plan when the information is actually needed

- asynchronous TuCSoN operations handled adopting a reactive programming style, e.g. calling `getResult()` in a separate parallel plan dedicated solely to handling the operation result—e.g. storing it as a belief, thus triggering belief addition plans, or explicitly triggering dedicated plans processing the result

C. Multi-paradigm CNP with TuCSoN4*Jason*

As a simple yet expressive example of multi-paradigm coordination, we describe a book trading scenario implemented as a *Jason* MAS—the example is included in TuCSoN4*Jason* distribution. In particular, we re-interpret the well-known Contract Net Protocol (CNP) [13] in a multi-paradigm coordination setting, where part of the protocol relies on synchronous tuple-based objective coordination – delivered by TuCSoN – and part on asynchronous message-based subjective coordination—built-in in *Jason*.

In short, n seller agents advertise their catalogue of books, whereas m buyer agents browse such catalogues looking for books. The whole interaction chain naturally follows FIPA Contract Net Interaction Protocol [20]: buyers start a call-for-proposals, sellers reply with actual proposals, buyers choose which one to accept, the purchase is carried out. A fundamental requirement – called *concurrency property* in the following – is that sellers should stay reactive to call-for-proposals even in the middle of a purchase transaction—otherwise they could lose potential revenues. The book trading scenario is thus taken here as a paradigmatic example of the practical relevance of preserving autonomy while enabling multi-paradigm coordination.

Accordingly, we re-think the CNP by integrating objective and subjective coordination approaches: tuple-based call-for-proposals with message-based purchase. In fact, since the call-for-proposals should reach all the sellers, putting a single call-for-proposals tuple in a shared contract-net space is more efficient than messaging each seller individually. On the contrary, since the purchase is typically a 1-to-1 interaction, messaging can efficiently do the job. As a result, this approach is not just conceptually correct, but also more efficient—less messages, less network operations, etc.

For the purpose of comparison, we first attempt to implement the MAS without exploiting TuCSoN4JADE, just directly calling TuCSoN API from within *Jason* with *Jason* internal actions without further concerns (Figure 5); then we repeat the same exercise by relying on TuCSoN4*Jason* (Figure 6). As one could expect, the result is that in the former case the concurrency property – thus, agent autonomy – is lost, whereas it is preserved as expected in the latter.

Figure 5 depicts one possible instance of the run-time interactions between a given seller and a given buyer. In particular, the seller is replying to a previous call-for-proposals (message 3b). Meanwhile, it is also ready to serve new incoming call-for-proposals (3a). Here is the problem: the suspensive coordination operation `rd` gets stuck until a call-for-proposals is issued by a buyer. This is fine: it is exactly for this suspensive semantics that the LINDA model works. What is not-so-fine is that the `rd` is stuck on a network-level call and no "defensive" programming mechanism has been implemented to shield the caller intention. Thus it is stuck too, hindering the caller agent from scheduling other intentions in the meanwhile—in particular, the "purchase" interaction chain (4b-5b) cannot carry on until a new call-for-proposals is issued.

Figure 6 depicts the same scenario, now programmed upon the TuCSoN4*Jason* bridge, thus preserving autonomy. Since the `rd` call is shielded by a proper mechanism within the bridge, the suspensive semantics is confined to the caller intention. This means that only the caller intention is suspended – using the proper mechanisms provided by *Jason* (see Figure 3) – whereas other activities can carry on concurrently—e.g., the purchase transaction already in place (4b-5b).

The wide applicability of the CNP, as well as its suitability for implementation as a multi-paradigm coordination protocol – drawing from both objective and subjective, synchronous, and asynchronous approaches – makes the concept of multi-paradigm coordination quite relevant in the context of agent development frameworks and coordination technologies—also making the need for suitable middleware components such as TuCSoN4*Jason* quite apparent.

V. DISCUSSION & RELATED WORKS

In [21], CArtAgO [22] is integrated with three different agent development frameworks: *Jason*, 2APL [23], and simpA [24]. The approach taken is an example of *autonomy-preserving integration*: e.g., in the case of *Jason*-CArtAgO, *Jason* intentions suspension mechanism is successfully integrated with CArtAgO artefacts by exploiting CArtAgO agent body abstraction. In particular, whenever a *Jason* agent requests execution of an operation on a CArtAgO artefact, the caller intention is automatically suspended until the "effector feedback" is received. Thus, nothing can hinder *Jason* agent autonomy if they *simultaneously* operate on artefacts while exchanging messages with other agents.

In [19], integration between JADE and TuCSoN technologies is successfully achieved, allowing JADE agents to exploit TuCSoN coordination services as part of the JADE platform. However, without preserving autonomy. JADE model of autonomy and TuCSoN model of coordination [11] were not considered: in fact, if a coordination operation gets suspended, the caller behaviour is unavoidably suspended, too, because of its single thread of control being stuck waiting for operation completion. This inevitably leads to the suspension of all other behaviours the agent is (possibly) concurrently executing. Roughly speaking, the agent choice to rely on objective coordination may affect its ongoing subjective coordination activities. This is a clear example of an artificial dependency (unintentionally) created by a "non autonomy-preserving" approach.

VI. CONCLUSION & OUTLOOK

Enabling *multi-paradigm coordination* along a number of orthogonal dimensions of coordination – such as objective vs. subjective stance, synchronism of coordination primitive invocation, reactiveness of the programming style promoted by the API – potentially brings about a wide range of communication, synchronisation, namely, *coordination-related* requirements to be satisfied, whenever the coordinables are provided with the ability to change the way they exploit

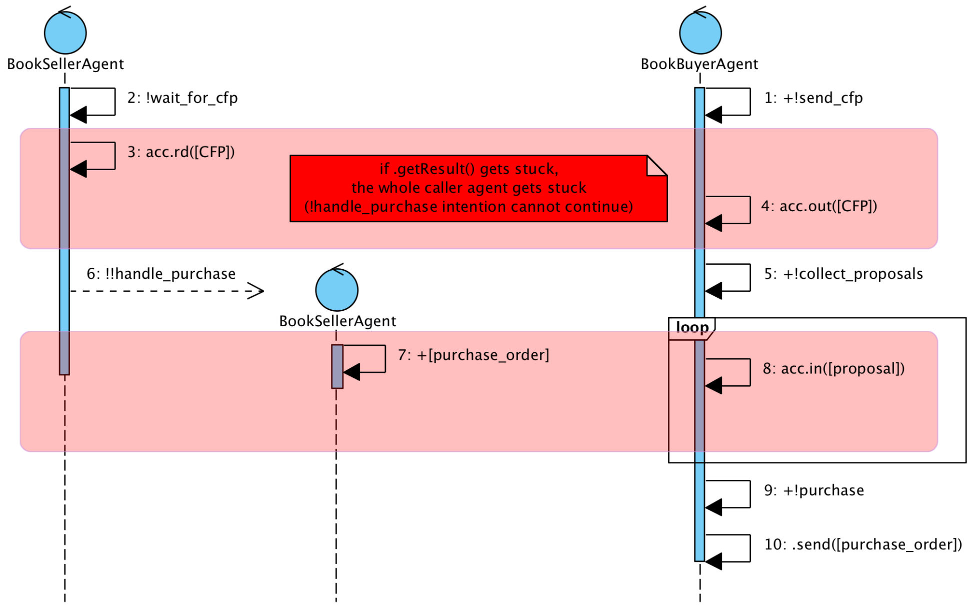

Figure 5: Non autonomy-preserving approach taken in [19]: `rd` suspensive semantics extends to the caller behaviour, then to the caller agent, blocking all its activities.

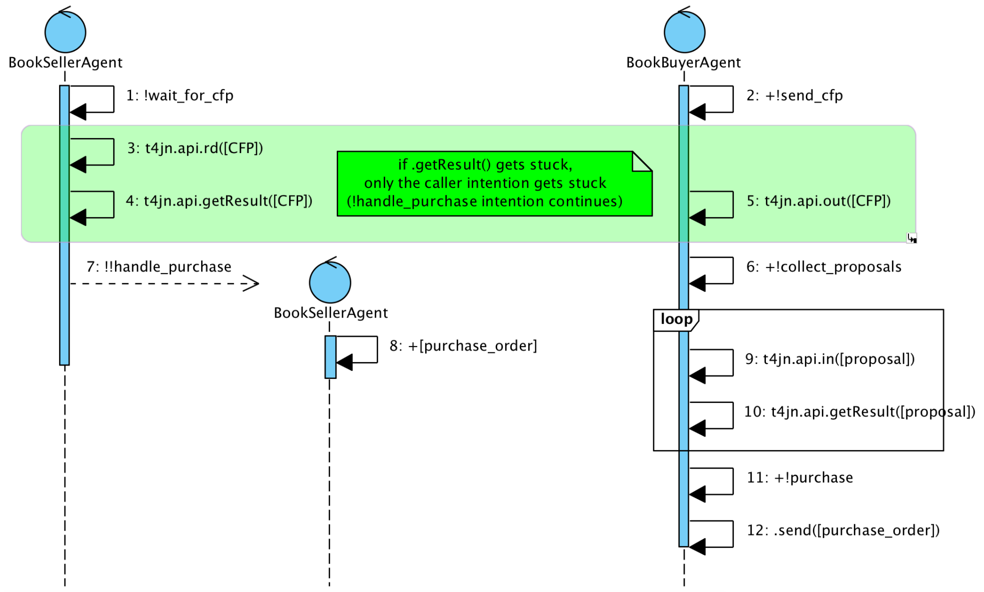
BookSellerAgent
BookBuyerAgent
1: !wait_for_cfp
2: +!send_cfp
3: t4jn.api.rd([CFP])
if .getResult() gets stuck,
only the caller intention gets stuck
(!handle_purchase intention continues)
4: t4jn.api.getResult([CFP])
5: t4jn.api.out([CFP])
7: !!handle_purchase
6: +!collect_proposals
BookSellerAgent
loop
8: +[purchase_order]
9: t4jn.api.in([proposal])
10: t4jn.api.getResult([proposal])
11: +!purchase
12: .send([purchase_order])

Figure 6: TuCSoN4*Jason* autonomy-preserving approach to multi-paradigm coordination: `rd` suspensive semantics is confined to the caller intention only, thus the caller agent can carry on its other concurrent intentions.

coordination services *at run-time*, depending on preference or ever-changing, unpredictable contingencies.

For these reasons, technologies like TuCSoN4*Jason* and TuCSoN4JADE may be *key-enablers* in all those application scenarios featuring distributed and heterogeneous components which, on the one hand, heavily depends on communication and coordination to carry out their duties, on the other hand, may need to adopt diverse coordination paradigms depending on both their own capabilities and run-time contingencies. Such a sort of scenarios are usually dealt with by *pervasive* MAS [25], where, besides distribution and heterogeneity, also *autonomy* and *(situated) intelligence* typically play a key role: autonomy, for infusing the system with non-functional properties such as tolerance to failures, situated intelligence for improving the system effectiveness by leveraging reasoning over, e.g., contextual information.

The fact that *Jason* adopts the BDI model for agents, coupled with TuCSoN choice of working with first-order logic tuples, makes TuCSoN4*Jason* even more relevant for the aforementioned class of systems.

In the near future, we plan to further strengthen integration between TuCSoN and both *Jason* and JADE: for the former, private tuple spaces as belief bases and shared tuple spaces as environmental resources are both directions we wish to explore; for the latter, we envision automatic translation back and forth FIPA ACL messages and first-order logic tuples, as well as support for tuple-based FIPA interaction protocols.

REFERENCES

[1] A. Omicini and S. Ossowski, "Objective versus subjective coordination in the engineering of agent systems," in *Intelligent Information Agents: An AgentLink Perspective*, ser. Lecture Notes in Computer Science, M. Klusch, S. Bergamaschi, P. Edwards, and P. Petta, Eds. Springer Berlin Heidelberg, 2003, vol. 2586, pp. 179–202. [Online]. Available: http://link.springer.com/10.1007/3-540-36561-3_9

[2] P. Ciancarini, "Coordination models and languages as software integrators," *ACM Computing Surveys*, vol. 28, no. 2, pp. 300–302, Jun. 1996. [Online]. Available: http://portal.acm.org/citation.cfm?id=234732

[3] A. Omicini, A. Ricci, and M. Viroli, "Coordination artifacts as first-class abstractions for MAS engineering: State of the research," in *Software Engineering for Multi-Agent Systems IV: Research Issues and Practical Applications*, ser. Lecture Notes in Computer Science, A. F. Garcia, R. Choren, C. Lucena, P. Giorgini, T. Holvoet, and A. Romanovsky, Eds. Springer Berlin Heidelberg, Apr. 2006, vol. 3914, pp. 71–90, invited Paper. [Online]. Available: http://link.springer.com/10.1007/11738817_5

[4] M. Viroli and A. Omicini, "Coordination as a service," *Fundamenta Informaticae*, vol. 73, no. 4, pp. 507–534, 2006, Special Issue: Best papers of FOCLASA 2002. [Online]. Available: http://content.iospress.com/articles/fundamenta-informaticae/fi73-4-04

[5] C. Castelfranchi, "Modelling social action for AI agents," *Artificial Intelligence*, vol. 103, no. 1-2, pp. 157–182, Aug. 1998. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0004370298000563

[6] T. W. Malone and K. Crowston, "The interdisciplinary study of coordination," *ACM Computing Surveys*, vol. 26, no. 1, pp. 87–119, 1994. [Online]. Available: http://dl.acm.org/citation.cfm?doid=174668

[7] G. M. O'Hare and N. R. Jennings, Eds., *Foundations of Distributed Artificial Intelligence*, ser. Sixth-Generation Computer Technology. John Wiley & Sons, Apr. 1996. [Online]. Available: http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0471006750.html

[8] D. Gelernter, "Generative communication in Linda," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, pp. 80–112, Jan. 1985. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2433

[9] F. L. Bellifemine, A. Poggi, and G. Rimassa, "JADE–a FIPA-compliant agent framework," in *4th International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM-99)*. London, UK: The Practical Application Company Ltd., 19–21 Apr. 1999, pp. 97–108. [Online]. Available: http://jade.cselt.it/papers/PAAM.pdf

[10] R. H. Bordini, J. F. Hübner, and M. J. Wooldridge, *Programming Multi-Agent Systems in AgentSpeak using Jason*. John Wiley & Sons, Ltd, Oct. 2007. [Online]. Available: http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0470029005.html

[11] S. Mariani, A. Omicini, and L. Sangiorgi, "Models of autonomy and coordination: Integrating subjective & objective approaches in agent development frameworks," in *Intelligent Distributed Computing VIII*, ser. Studies in Computational Intelligence, L. Braubach, D. Camacho, and S. Venticinque, Eds., vol. 570. Springer, 2014, pp. 69–79, 8th International Symposium on Intelligent Distributed Computing (IDC 2014), Madrid, Spain, 3-5 Sep. 2014. Proceedings. [Online]. Available: http://link.springer.com/10.1007/978-3-319-10422-5_9

[12] F. Bellifemine, G. Caire, T. Trucco, and G. Rimassa, "JADE programmer's guide," http://jade.tilab.com/doc/programmersguide.pdf, Telecom Italia S.p.A., Apr. 2010.

[13] R. G. Smith, "The Contract Net Protocol: High-level communication and control in a distributed problem solver," *IEEE Transactions on Computers*, vol. C-29, no. 12, pp. 1104–1113, 1980. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1675516

[14] Foundation for Intelligent Physical Agents, "FIPA Communicative Act Library Specification," http://www.fipa.org/specs/fipa00037/, 6 Dec. 2002.

[15] A. S. Rao, "AgentSpeak(L): BDI agents speak out in a logical computable language," in *Agents Breaking Away*, ser. Lecture Notes in Computer Science, W. Van de Velde and J. W. Perram, Eds. Springer, 1996, vol. 1038, pp. 42–55, 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96), Eindhoven, The Netherlands, 22-25 Jan. 1996, Proceedings. [Online]. Available: http://link.springer.com/10.1007/BFb0031845

[16] A. Omicini and F. Zambonelli, "Coordination for Internet application development," *Autonomous Agents and Multi-Agent Systems*, vol. 2, no. 3, pp. 251–269, Sep. 1999, Special Issue: Coordination Mechanisms for Web Agents. [Online]. Available: http://link.springer.com/10.1023/A:1010060322135

[17] A. Omicini and E. Denti, "From tuple spaces to tuple centres," *Science of Computer Programming*, vol. 41, no. 3, pp. 277–294, Nov. 2001. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642301000119

[18] A. Omicini, "Towards a notion of agent coordination context," in *Process Coordination and Ubiquitous Computing*, D. C. Marinescu and C. Lee, Eds. Boca Raton, FL, USA: CRC Press, Oct. 2002, ch. 12, pp. 187–200.

[19] A. Omicini, A. Ricci, M. Viroli, M. Cioffi, and G. Rimassa, "Multi-agent infrastructures for objective and subjective coordination," *Applied Artificial Intelligence: An International Journal*, vol. 18, no. 9–10, pp. 815–831, Oct.–Dec. 2004, special Issue: Best papers from EUMAS 2003: The 1st European Workshop on Multi-agent Systems. [Online]. Available: http://www.tandfonline.com/doi/10.1080/08839510490509036

[20] Foundation for Intelligent Physical Agents, "FIPA Contract Net Interaction Protocol specification," http://www.fipa.org/specs/fipa00029/, 6 Dec. 2002.

[21] A. Ricci, M. Piunti, L. D. Acay, R. H. Bordini, J. Hübner, and M. Dastani, "Integrating heterogeneous agent programming platforms within artifact-based environments," in *7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-08)*, L. Padgham and D. C. Parkes, Eds. Estoril, Portugal: IFAAMAS, 12–16 May 2008, pp. 225–232. [Online]. Available: http://dl.acm.org/citation.cfm?id=1402419

[22] A. Ricci, M. Viroli, and A. Omicini, "CArtAgO: A framework for prototyping artifact-based environments in MAS," in *Environments for MultiAgent Systems III*, ser. Lecture Notes in Artificial Intelligence, D. Weyns, H. V. D. Parunak, and F. Michel, Eds. Springer, May 2007, vol. 4389, pp. 67–86, 3rd International Workshop (E4MAS 2006), Hakodate, Japan, 8 May 2006. Selected Revised and Invited Papers. [Online]. Available: http://link.springer.com/10.1007/978-3-540-71103-2_4

[23] M. Dastani and J.-J. C. Meyer, "A practical agent programming language," in *Programming Multi-Agent Systems*, ser. Lecture Notes in Computer Science, M. Dastani, A. El Fallah Seghrouchni, A. Ricci, and M. Winikoff, Eds. Springer, 2008, vol. 4908, pp. 107–123, 5th International Workshop, ProMAS 2007 Honolulu, HI, USA, May 15, 2007 Revised and Invited Papers. [Online]. Available: http://link.springer.com/10.1007/978-3-540-79043-3_7

[24] A. Ricci, M. Viroli, and G. Piancastelli, "simpA: A simple agent-oriented Java extension for developing concurrent applications," in *Languages, Methodologies and Development Tools for Multi-Agent Systems*, ser. Lecture Notes in Computer Science. Springer, Jul. 2008, vol. 5118, pp. 261–278, 1st International Workshop, LADS 2007, Durham, UK, September 4-6, 2007. Revised Selected Papers. [Online]. Available: http://link.springer.com/10.1007/978-3-540-85058-8_16

[25] F. Zambonelli, A. Omicini, B. Anzengruber, G. Castelli, F. L. DeAngelis, G. Di Marzo Serugendo, S. Dobson, J. L. Fernandez-Marquez, A. Ferscha, M. Mamei, S. Mariani, A. Molesini, S. Montagna, J. Nieminen, D. Pianini, M. Risoldi, A. Rosi, G. Stevenson, M. Viroli, and J. Ye, "Developing pervasive multi-agent systems with nature-inspired coordination," *Pervasive and Mobile Computing*, vol. 17, pp. 236–252, Feb. 2015, Special Issue "10 years of Pervasive Computing" In Honor of Chatschik Bisdikian. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1574119214001904