# TuCSoN Coordination for MAS Situatedness: Towards a Methodology

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*—**Agent-based technologies embed solutions for critical issues in agent-oriented software engineering. In this paper we describe the coordination-based approach to MAS situatedness as promoted by the TuCSoN middleware, by sketching the steps of an agent-oriented methodology from the TuCSoN meta-model down to the TuCSoN programming environment.**

## I. Coordination and Situatedness in MAS

The need for *situatedness* in multi agent systems (MAS) is often translated into the requirement of being sensitive to *environment change* [1], possibly influencing the environment in turn. Such a requirement lays at the core of the notion of *situated action* – complementing that of *social action* [2] –, as those actions arising from strict interaction with the environment [3]. This leads to recognise *dependencies* among agents and the environment as one of the fundamental sources of complexity within a MAS—the other being dependencies between agents' activities [4]. Therefore, *coordination* – as the discipline of managing dependencies [4] – could be used to deal with both *social* and *situated interaction*, by exploiting *coordination artefacts* for handling both social and situated dependencies [5].

Accordingly, in this paper we introduce the *situated coordination* approach promoted by the TuCSoN model and technology for agent coordination [6] to handle situatedness in MAS as a coordination issue. In particular, we describe which support TuCSoN provides to MAS programmers in each macro-stage of a typical software engineering process applied to a MAS: the abstractions available for the requirement analysis (Section II), the run-time architecture to refer to during the design phase (Section III), the API provided to support implementation of the concept of situated coordination (Section IV).

## II. Requirement Analysis: The TuCSoN Meta-model

The availability of well-known and established development frameworks and middleware often lead to (implicit) methodologies which are essentially driven by the abstractions promoted and supported by the technology [7]. This typically happens when the maturity of technologies precedes that of methodologies—and actually happened for agent-oriented technologies in the last decade [8].

What influences the process of MAS engineering based on an agent-oriented framework is first of all the conceptual framework provided by the technology, and in particular the *meta-model* behind it, which fundamentally shapes the *space of the solutions*: the availability of different abstractions to elaborate over the application problem usually leads to different designs and implementations—and ultimately, to different solutions, too. This is why in the remainder of this section we describe the meta-model of the TuCSoN model and technology for agent coordination[6]; that is, the set of abstractions provided by TuCSoN in order to model application problems since the very beginning of the engineering stage.

Three are the TuCSoN core concepts for MAS engineering, which motivate the architecture described in Section III: *activities*, *environment change*, *dependencies*.

*Activities* are the *goal-directed/oriented* proceedings resulting into *actions* of any sort, which "make things happen" in a MAS. Through actions, activities in a MAS are *social* [2] and *situated* [3]. Activities are usually modelled through the *agent* abstraction: the reason for this choice is that, often, MAS designers are not merely interested in modelling an action "as is", but they also want / need to model also the *motivations* behind that action—namely, their *goal*. Thus, from the standpoint endorsed here, agents do not exist because they resemble some "real-world" entity; they exist as the means through which activities can be modelled in a MAS—as a way to model actions along with their driving goals.

*Environment change* represents the (possibly unpredictable) variations in the properties or structure of the world surrounding a MAS that affect it in any way—thus, which the MAS needs to account for. Such variations do not express any specific goal, either because this does not exist, or because it is not to be / cannot be modelled in the MAS. Also, these variations may not correspond to actual changes in the real-world properties or structure, but simply variations in the *perception* of the world the MAS has—in other words, what the MAS *observes* may vary independently of whether the environment actually changes too. Environment (change) is usually modelled through the *resource* abstraction, as a non-intelligent entity either continuously producing events or reactively waiting for requests to perform its function.

Finally, in any non-trivial MAS, activities *depend* on other activities (*social dependencies*), as well as on environment change (*situated dependencies*). Thus, *dependencies* motivate and cause *interaction*, both social and situated, based on the sort of dependency taking place.

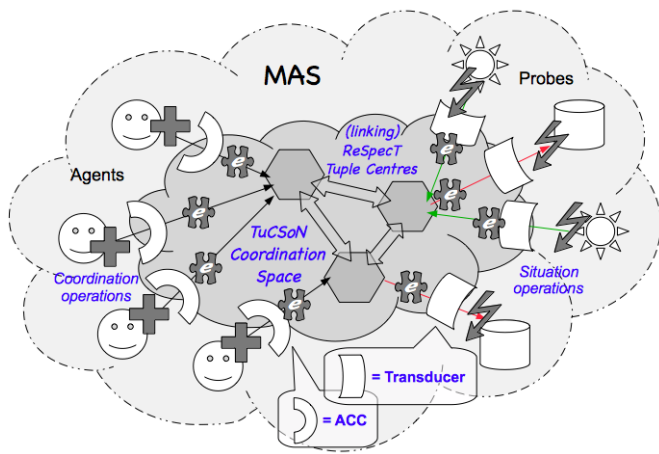Furthermore, the core notion linking the TuCSoN ar-

Fig. 1. TuCSoN architecture. ACC and transducers mediate the interactions between agents and the environment by translating activities and changes into events, which are then handled by tuple centres so as to support both coordination and situatedness. The "inner cloud" can be seen as a TuCSoN *node*, that is, a single "place" in which the coordination between MAS entities happens. Actually, such node can be distributed across networked devices, blurring the distinction between a TuCSoN node and a TuCSoN *system*.

chitecture to its meta-model is that of *event*. Despite their intrinsic diversity, activities and environment change constitute altogether the only sources of dynamics – thus complexity – in a MAS. In order to provide a uniform representation of MAS dynamics and to promote a coordination-oriented approach in modelling social as well as situated dependencies, both activities and environment changes are represented in TuCSoN as *events*. Therefore, in TuCSoN, events reify any social and any situated interaction taking place within the MAS, driving the coordination process.

Given the above abstractions, MAS designers should think about the problem at hand in terms of *(i)* a bunch of goal-directed/oriented activities (agents) *(ii)* interacting with each other and influenced by / influencing some sort of *(iii)* changes in their environment (resources), therefore *(iv)* generating events which *(v)* have to be properly coordinated.

## III. MODEL & DESIGN: THE TuCSoN ARCHITECTURE

In this section, we first overview the TuCSoN architecture by describing its main components, directly stemming from the TuCSoN meta-model introduced in Section II. Then we sketch how such components collaborate to properly support the modelling of activities and environment changes in TuC-SoN (Subsection III-A and Subsection III-B), in particular focussing on agent-environment interactions—that is, situated dependencies (Subsection III-C).

TuCSoN[1] [6] is a Java-based, tuple-based coordination infrastructure for open distributed MAS. Its main architectural run-time components are—as depicted in Fig. 1:

agents — Any computational entity willing to exploit TuCSoN coordination services [9] is a TuCSoN *agent*. In order for agents to be recognised as *co-ordinables* [10] by TuCSoN, they need to obtain an ACC (see below), released by TuCSoN itself.

Agents (interaction-oriented) activities result into *coordination operations*, targeting the coordination media (a tuple centre, see below) actually handled by the ACC.

ACC — *Agent Coordination Contexts* [11] are TuC-SoN architectural components devoted to represent and mediate agents activities within the MAS. In particular, an ACC maps coordination operations (thus both social and situation actions) into events, dispatches them to tuple centres, waits for the outcome of dependency resolution (that is, coordination), then sends the operation result back to the agent. ACC are also the fundamental run-time entities that preserve agent *autonomy* [11]: in fact, while the ACC takes care of *asynchronously* dispatching events – consequence of agent's activity – to tuple centres, the agent is free to undertake other activities. This enables *uncoupling* in *control, reference, space* and *time*.

probes — Environmental resources in TuCSoN are called *probes*. They can be either sources of perceptions (like *sensors*), targets of actions (like *actuators*), or even both: TuCSoN models them in the same way, using transducers. In fact, actions over probes are carried out by *transducers*: as for agents with ACC, probes do not directly interact with the MAS, but through transducer mediation.

transducers — Analogously to ACC for agents, TuC-SoN *transducers* [12] are the architectural run-time components in charge of representing and mediating environment changes regarding probes. Each probe is assigned a transducer, which is specialised to handle events to/from that probe. So, in particular, transducers translate *(i)* probes properties changes into events, to be dispatched to tuple centres and properly coordinated, and *(ii)* MAS events into properties changes, to be sensed/effected on probes—ultimately, enacting situatedness.

events — TuCSoN adopts the ReSpecT [13] event model – adapted from [14] in TABLE I on page 8 –, representing any sort of event happening in the MAS in a uniform way—both the events generated from agents activities and those from changes in the environment. Events are the data structure reifying all the relevant information about the activity or change that generated them. In particular, TuCSoN events record: the *immediate* and *primary* cause of the event [15], its outcome, who is the source of the event, who is its target, when and where the event was generated. In this way, any event captured by TuCSoN – through ACC and transducers mediation – is situated both in space and time, as well as within its *execution context*. Such a feature lays at the core of the notion of *situated coordination*, meaning that tuple centres can effectively coordinate events (thus resolve dependencies) while accounting for the situated nature of interactions to its full extent.

tuple centres — ReSpecT tuple centres [16] are the TuC-SoN architectural component mediating all interactions happening in the MAS, thus in charge of
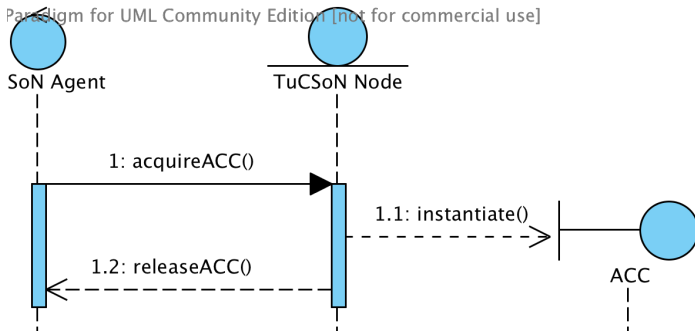
Fig. 2. ACC acquisition by TuCSoN agents. Nothing can be done by an agent with the TuCSoN middleware prior to ACC acquisition.

Fig. 5. ACC release by TuCSoN agents. Nothing can be done by an agent with the TuCSoN middleware after ACC release.

handling events in order to resolve dependencies. They are run by the TuCSoN middleware to rule and uncouple (in *control*, *reference*, *space*, and *time*) dependencies between agent activities as well as environment change—in other words, both social and situated interactions [5]. By adopting ReSpecT tuple centres, TuCSoN relies on *(i)* the ReSpecT language to program coordination laws, and *(ii)* the ReSpecT situated event model to implement events.

Summing up, MAS designers willing to benefit of TuCSoN coordination services should: *(i)* rely on ACC and therein defined primitives to interact with other TuCSoN-coordinated entities, *(ii)* define suitable transducers to represent the relevant portions of MAS environment, *(iii)* program TuCSoN tuple centres through ReSpecT specifications to effectively handle TuCSoN events—therefore, to effectively coordinate the MAS.

### A. Agent side

The *agent side* of a TuCSoN-coordinated MAS is basically represented by the run-time relationships between agents, ACC, and tuple centres.

First of all, as depicted in Fig. 2, TuCSoN agents have to acquire an ACC before issuing any sort of coordination operation towards the TuCSoN infrastructure. They do so by asking the TuCSoN middleware to release an ACC. Whether an ACC is actually released, and which one among those available[2] is dynamically determined by the TuCSoN middleware itself, based upon the agent request and its expected *role* inside the MAS [17].

Once a TuCSoN agent obtains an ACC,, all its interactions are mediated by the ACC itself, with no role for the TuCSoN node. In particular, as depicted in Fig. 3, in the case a coordination operation is requested through a *synchronous invocation*:

*(i)* first of all (messages $2 - 2.1.2$), the target tuple centre associated to the ACC is dynamically instantiated by the TuCSoN run-time infrastructure, and its network address given to the ACC for further reference

---

[2]See the TuCSoN official guide at http://www.slideshare.net/andreaomicini/the-tucson-coordination-model-technology-a-guide.
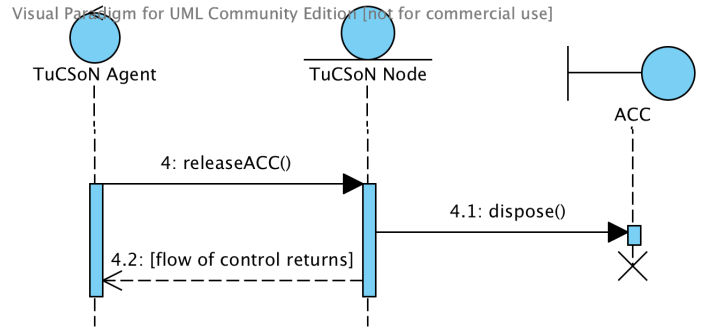
*(ii)* then (messages 2.2), the ACC takes charge of building the corresponding event and of dispatching it to the tuple centre target of the interaction

*(iii)* finally (messages $2.2.1 - 2.2.2.1$), the ACC is notified when the outcome of the coordination operation requested is available – after a proper coordination stage possibly involving other events from other entities – so that it can send the operation result back to the agent

Only the coordination operation request from the agent to its ACC is a *synchronous method call*: any other interaction is *asynchronous* as well as *event-driven*. This is necessary in every open and distributed scenario, and enables the already mentioned uncoupling in control, reference, space, and time. Nevertheless, in such a scenario – synchronous operation invocation – the control flow of the caller agent is retained by the ACC as long as the operation result is not available (message 2.2.2.1).

Conversely, Fig. 4 depicts the *asynchronous invocation* scenario: the only difference w.r.t. the synchronous one lays in the fact that the control flow is given back to the caller agent as soon as the corresponding event is dispatched to the target tuple centre (message 3.4). The actual result of the requested coordination operation is dispatched to the agent as soon as it becomes available, asynchronously (message 3.3.2.1). TuCSoN lets client agents determine which semantics to use for their coordination operations invocation, either synchronous or asynchronous.

Furthermore, the scenario depicted in Fig. 4 assumes that the target tuple centre is already up and running – e.g., as a consequence of a previous operation invocation – thus, the TuCSoN node simply retrieves its reference, and passes it to the ACC.

Whenever an agent no longer needs TuCSoN coordination services, it should release its ACC back to TuCSoN middleware, which promptly destroys it in order to prevent resources leakage—as depicted in Fig. 5. It should be noticed that there is no way to re-acquire an already-released ACC – e.g., to restore interactions history –, since whenever an ACC is requested a new one is created and assigned. Since ACC are used to represent and identify agents within a TuCSoN-coordinated MAS, an agent obtaining an ACC multiple times is recognised every time as a new agent by the TuCSoN middleware.

Summing up, designers of agents exploiting TuCSoN should make their agents: *(i)* acquire an ACC; *(ii)* choose
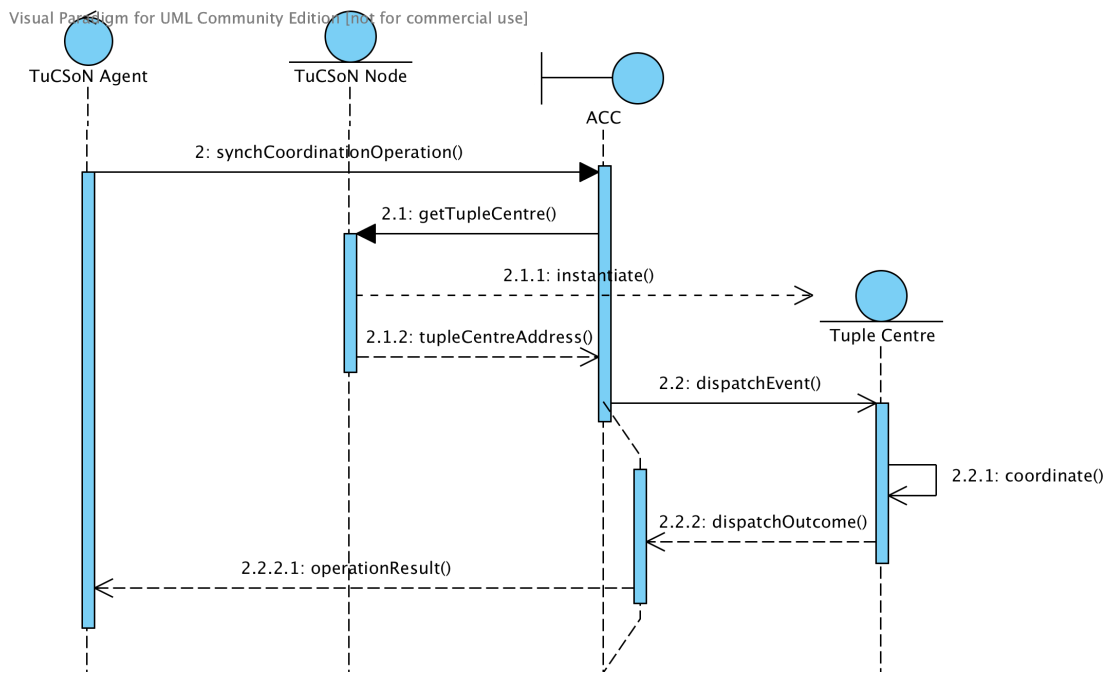
Fig. 3. Synchronous operation invocation. The control flow is released back to the agent only when the operation result is available—thus, only when the coordination process ends.
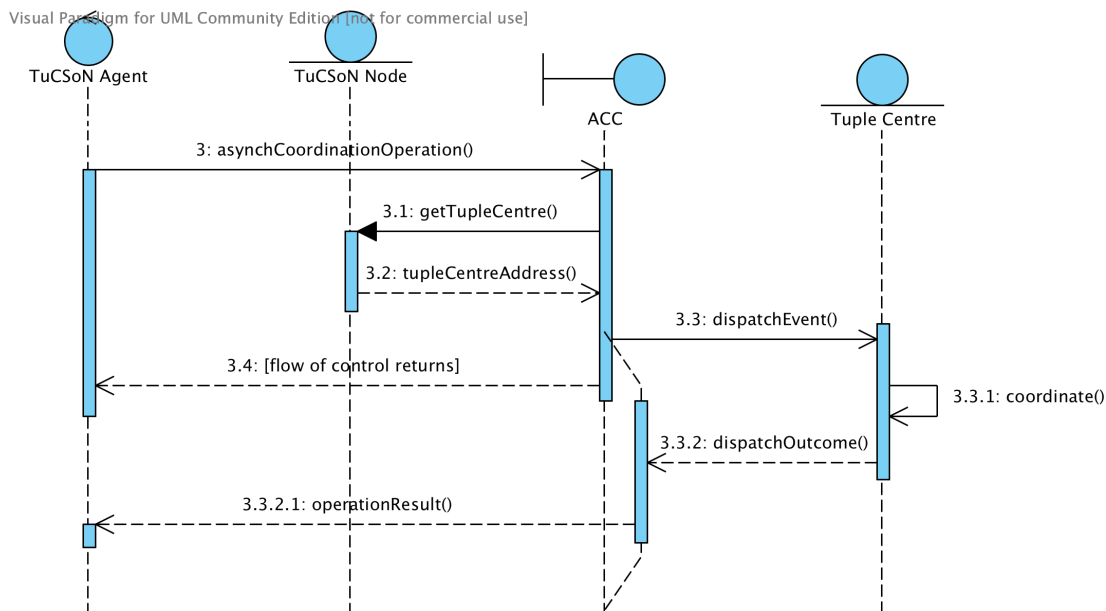


Fig. 4. Asynchronous operation invocation. The control flow is released back to the agent as soon as the event related to the request is generated and dispatched by the ACC.

each operation invocation semantics, and *(iii)* expect operations result to be available accordingly; *(iv)* release their ACC when TuCSoN service are no longer needed.

### B. Environment side

On the *environment side* of the TuCSoN architecture, agents and ACC are replaced by probes and transducers, respectively—as depicted by Fig. 6. Thus, first of all, probes should register to the TuCSoN middleware in order to get their

transducer and interact. After probe registration, any interaction resulting from environmental property change affecting the MAS are mediated by the transducer. Fig. 7 depicts the interaction among TuCSoN run-time entities in the case of a sensor probe, thus a sensor transducer, whereas Fig. 8 shows the case of an actuator probe. By comparing the two pictures, the flow of interactions is almost the same, except for the first invocation, which depends on nature of the probe—either sensor (Fig. 7) or actuator (Fig. 8).

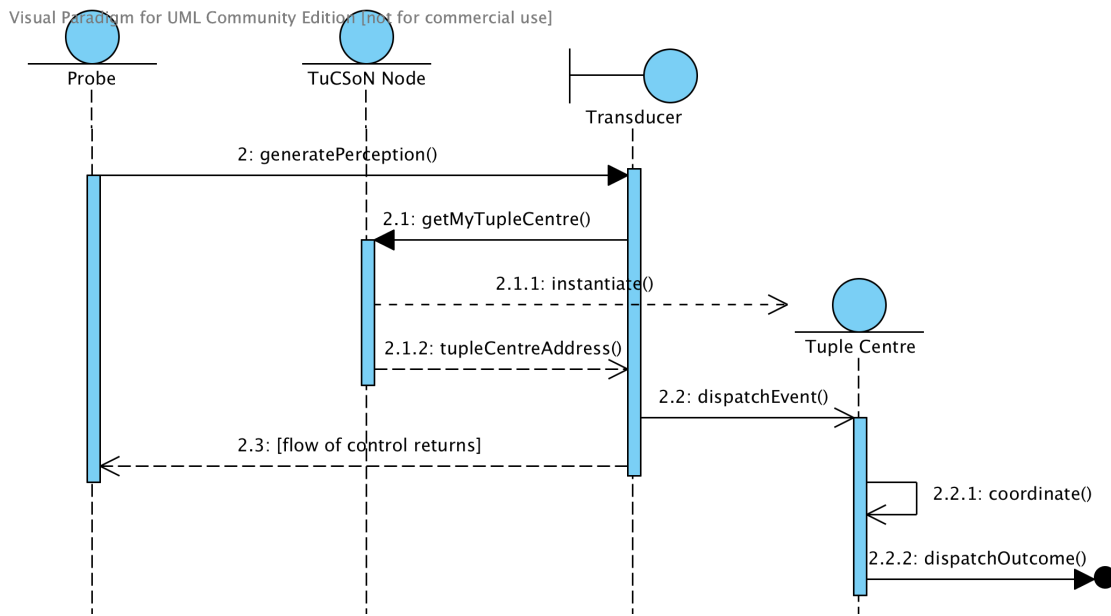In particular, a perception by a sensor probe works as

Fig. 7. Sensor probe interaction. The control flow returns to the probe as soon as the environmental event is generated and dispatched by the transducer, thus, everything happens asynchronously.
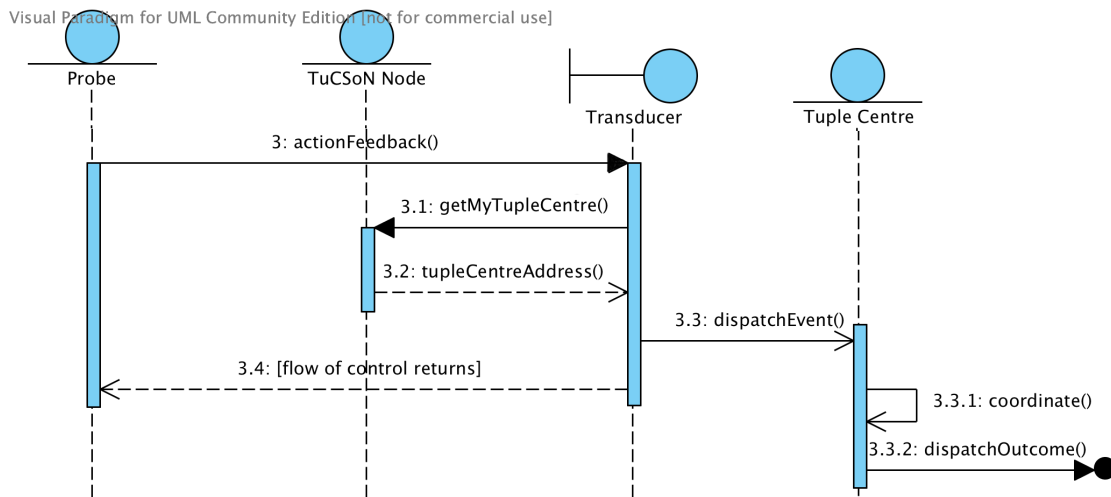


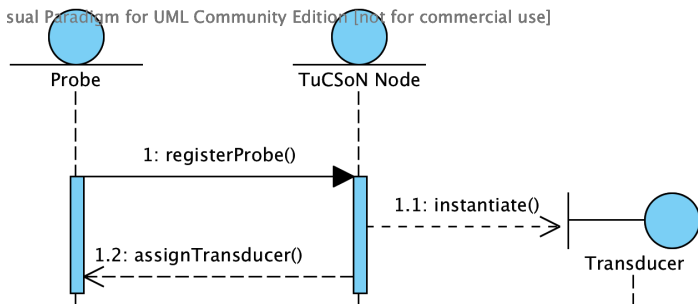Fig. 8. Actuator probe interaction. Again, everything happens asynchronously.



Fig. 6. Probes registration and transducers association. No events can be perceived nor actions undertaken on a probe prior to transducer association.

follows—Fig. 7:

*(i)* first of all (messages $2 - 2.1.2$), the target tuple centre associated to the transducer is dynamically instantiated by the TuCSoN run-time infrastructure, and its network address passed to the transducer for further reference

*(i)* then (messages 2.2), the transducer builds the event corresponding to the perception operation, and dispatches it to the tuple centre target of the interaction

*(i)* finally (messages $2.2.1 - 2.2.2$), the tuple centre enacts the coordination process triggered by such event (if any), properly dispatching its outcome

As far as probe interaction is concerned, there is no distinction between synchronous or asynchronous semantics—regardless if it involves either a sensor or an actuator. In fact, being representations of environmental resources, probes are not supposed to expect any feedback from the MAS: they simply cause / undergo changes that are relevant to the MAS. For this reason, the semantics of situation operations invocation
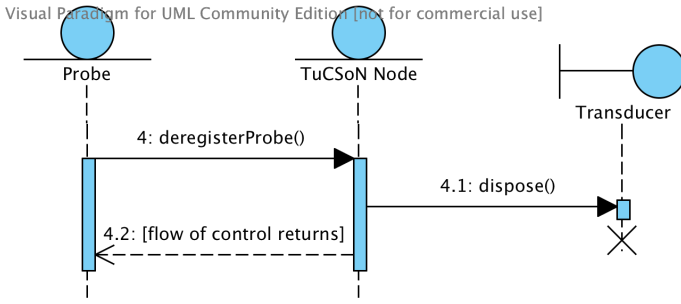
Fig. 9. Probe deregistration. Nothing can be either sensed or effected by the MAS upon the deregistered probe, since the mediating transducer is no longer running.

on probes is always asynchronous—as depicted in Fig. 7 and Fig. 8: the control flow is always returned to the probe as soon as the corresponding event is generated.

When a probe is no longer needed, it should be deregistered from TuCSoN, which subsequently destroys the associated transducer—as depicted in Fig. 9.

Wrapping up, TuCSoN situatedness services require MAS designers to: *(i)* always register probes causing their transducer instantiation; *(ii)* be aware that environmental events are always generated asynchronously; *(iv)* deregister probes when they are no longer needed—no automatic deregistration is performed by the TuCSoN middleware.

Finally, there is one last fundamental aspect of TuCSoN situatedness that MAS designers should be aware of—which will be clarified in next section.

### C. Between agents and environment: Situated coordination

Putting together the agent and the environment side of the TuCSoN event-driven architecture, Fig. 10 and Fig. 11 depict a synchronous interaction of an agent with a sensor, and an asynchronous interaction of an agent with an actuator, respectively. By inspecting the whole interaction sequence, one could see how *(i)* TuCSoN ACC and transducers play a central role in supporting distribution and uncoupling of agents and probes within the MAS, and *(ii)* how TuCSoN tuple centres and the ReSpecT language are fundamental to support both situatedness and *objective* coordination [18], [19].

In particular, in Fig. 10 the agent is issuing a synchronous coordination operation request involving a given tuple (`sense(temp(T))`)—message 1. After event dispatching (all the dynamic instantiation interactions were left out for the sake of clarity), the tuple centre target of the operation reacts to such invocation by triggering the ReSpecT reaction in annotation 1.1.1, which generates a situated event (step 1.1.2) aimed at executing a situation operation (`getEnv(temp, T)`) on the probe (`sensor`). The transducer associated to the tuple centre and responsible for the target probe intercepts such an event and takes care of actually executing the operation on the probe (message 1.1.2.1). The sensor probe reply (message 1.1.2.2) generates a sequence of events propagation terminating in the response to the original coordination operation issued by the agent (message 1.1.2.3.2.1).

It is worth noticing the role of the tuple centre in supporting situatedness: in fact, step 1.1.2.3.1 properly reacts to the completion of situation operation `getEnv(temp, T)` by the sensor probe, emitting exactly the tuple originally requested by the agent (`sense(temp(T))`).

In Fig. 11 the sequence of interactions as well as the annotations are very similar to those in Fig. 10. In particular, annotation 2.1.1 shows how the ReSpecT reaction triggering event matches the event raised as a consequence of agent coordination operation request (`act(temp(T))`), while annotation 2.1.2.3.1 highlights how the tuple centre maps the situation operation outcome (`setEnv(temp, T)`) in the originally requested tuple (`act(temp(T))`) through a proper ReSpecT reaction. The only differences w.r.t. Fig. 10 are the asynchronous invocation semantics used by the agent and the actuator nature of the interacting probe—thus, the name of the messages 2.1.2.1 and 2.1.2.2.

It is now evident that the missing aspect not considered in Subsection III-B involves the ReSpecT language: in fact, as shown by Fig. 10 and Fig. 11, ReSpecT is fundamental to program TuCSoN tuple centres so as to correctly bind coordination operations with situation operations, ultimately supporting agent-environment interactions—thus, situatedness.

## IV. IMPLEMENTATION: ReSpecT API

This section focusses on ReSpecT programming for situatedness and events handling, by discussing the ReSpecT language API. In particular, it is meant to explain what programmers can do in the implementation stage of TuCSoN-coordinated MAS by exploiting ReSpecT situated event model to its full extent.

Starting from the reaction annotating message 1.1.1 in Fig. 10, and according to ReSpecT formal syntax & semantics [13], we can distinguish:

`in(sense(temp(T)))` — the *triggering event*. As soon as the operation invocation event generated by the ACC arrives to the target tuple centre (message 1.1), it scans its ReSpecT program searching for any reactions whose triggering event *matches* the one received—where matching means unification in the first-order logic ReSpecT language. Any reaction found is collected and candidate for execution. In this case, the triggering event corresponds to an *Activity*, using the terminology of ReSpecT event model in TABLE I—that is, something coming from an agent.

`(operation, invocation)` — the *guard predicate*. Triggered reactions are further filtered based upon evaluation of their guards, that is, logic predicates allowing fine-grained control over reaction triggering, which can evaluate to either `true` or `false`. In this case, the `operation` guard filters coordination operation events whereas the `invocation` guard filters events from the ACC to the tuple centre. If all the guards of the reaction are evaluated to `true`, such reaction is scheduled for execution.

`[...] ? getEnv(temp, T)` — the actual *reaction*. After the guard-based filtering phase, the tuple centre non deterministically selects one reaction
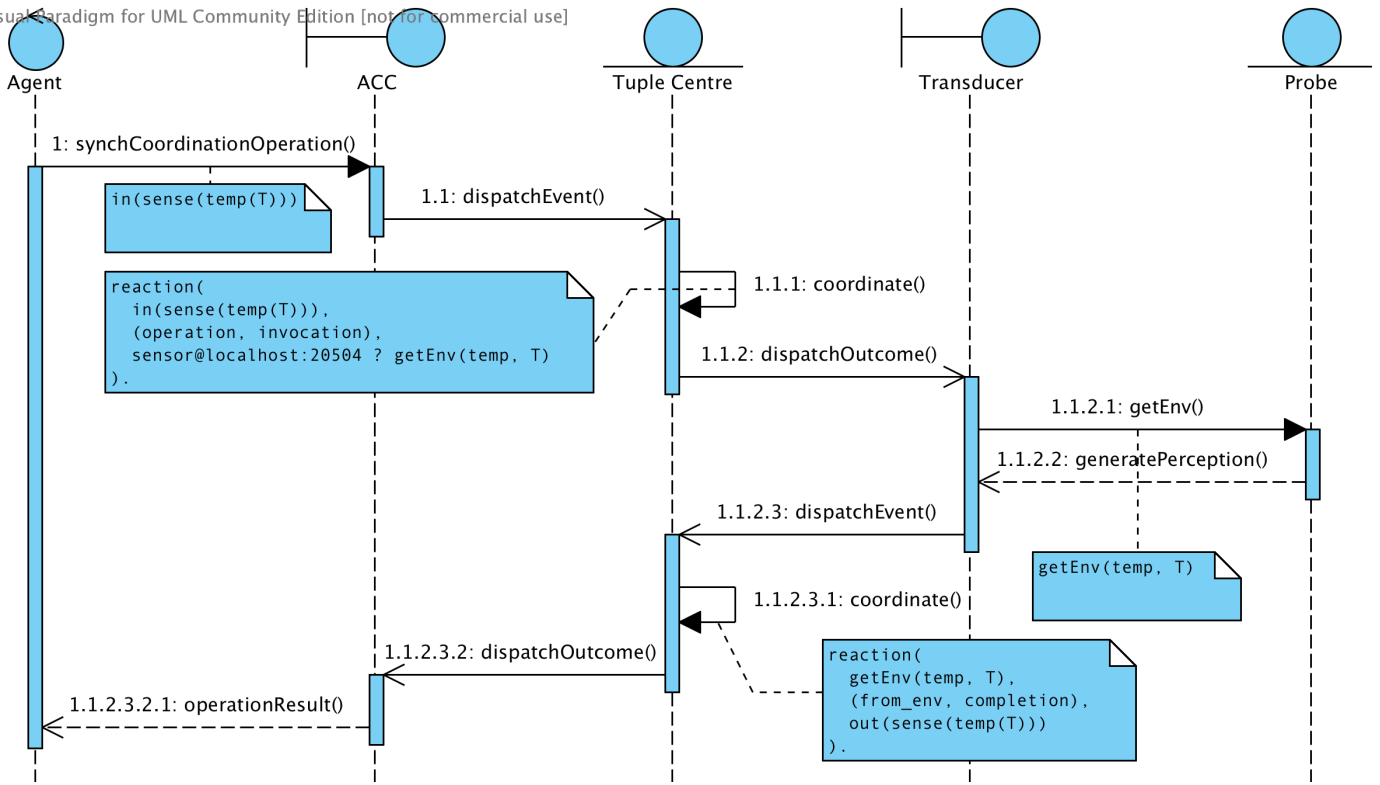
Fig. 10. Synchronous situation operation querying a sensor. ReSpecT plays a fundamental role in binding both the agent coordination operation to its corresponding situation operation (annotation in step 1.1.1) and the probe response back to the agent original request (annotation in step 1.1.2.3.1).
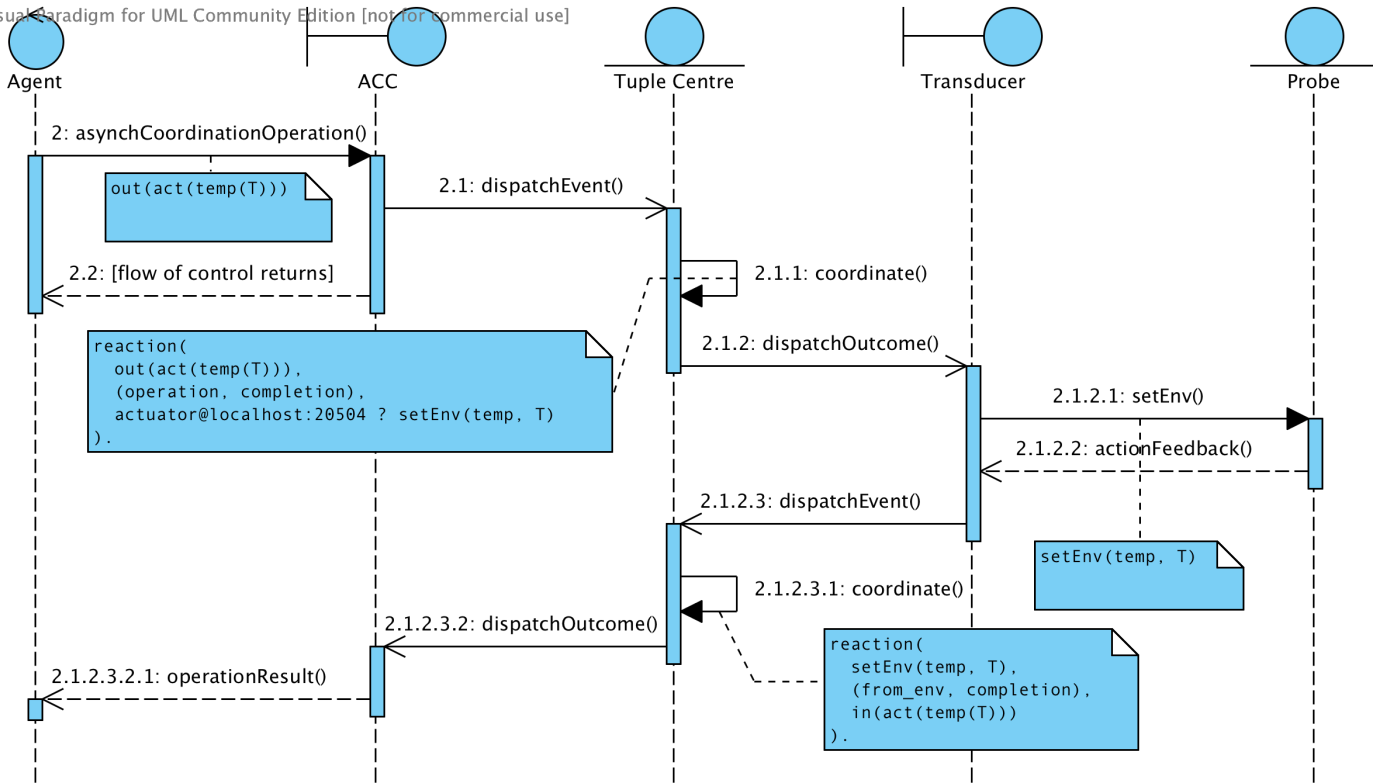


Fig. 11. Asynchronous situation operation commanding an actuator. As in Fig. 10, ReSpecT role in enabling situatedness is visible in annotations 2.1.1 and 2.1.2.3.1.

| ⟨*Event*⟩ | ::= | ⟨*StartCause*⟩ , ⟨*Cause*⟩ , ⟨*Evaluation*⟩ |
|---|---|---|
| ⟨*StartCause*⟩ , ⟨*Cause*⟩ | ::= | ⟨*Activity*⟩ \| ⟨*Change*⟩ , ⟨*Source*⟩ , ⟨*Target*⟩ , ⟨*Time*⟩ , ⟨*Space:Place*⟩ |
| ⟨*Source*⟩ , ⟨*Target*⟩ | ::= | ⟨*AgentId*⟩ \| ⟨*TCId*⟩ \| ⟨*ProbeId*⟩ \| ⊥ |
| ⟨*Evaluation*⟩ | ::= | ⊥ \| { ⟨*Result*⟩ } |

TABLE I.    ReSpecT SITUATED *event model*.

from the pool of those scheduled and starts executing it. In this case, the only computation to carry out is a change, again, using the terminology of TABLE I. In particular, the situation operation (`getEnv(...)`) on probe `sensor` – whose full name is the ⟨*ProbeId*⟩ in TABLE I –, which causes a situation operation event to be generated and dispatched first to the associated transducer, then to the actual probe.

After the request has been served, the field ⟨*Evaluation*⟩ from TABLE I is still empty—waiting for the completion to be carried on. In fact, due to the asynchronous nature of events dispatching in TuCSoN, the tuple centre itself does not suspend execution waiting for a response from the probe. This is necessary to face the issues of network communications in a distributed scenario.

For these reasons, the ReSpecT reaction annotating message 1.1.2.3.1 in Fig. 10 is complementary and necessary to complete the situated interaction meaningfully. In such reaction, the triggering event, the guards, and the reaction are, respectively:

getEnv(temp, T) — the situation operation event corresponding to the change execution by the tuple centre (through the probe transducer) in the first reaction (messages 1.1.2 - 1.1.2.3). The first reaction, in fact, requests the operation execution to the probe transducer, whereas this reaction manages such request reply.

(from_env, completion) — filtering situation operation events (from_env) representing the outcome of an execution (completion). Using these guards, MAS programmers are guaranteed to make the tuple centre react only when the requested situation operation has been actually executed on the target probe.

out(sense(temp(T))) — the computation emitting in the tuple centre the tuple reifying the information perceived by the sensor probe. Such a tuple perfectly matches the one used as argument of the coordination operation issued by the interacting agent: in fact, coupled with the synchronous invocation semantics chosen, this ensures MAS programmers that their agent will resume its execution only when the perception operation has been successfully carried out.

The above description of ReSpecT reactions machinery should the role played by the tuple centre coordination abstraction in supporting situatedness evident—thus, the concept of situated coordination. The reactions annotating Fig. 11 can

be explained in a similar way, thus they are left out from discussion.

Finally, a list of some of the methods available in the `Respect2PLibrary` Java class within TuCSoN distribution follows in the remainder of this section, which exposes the API for ReSpecT programmers. Such API allows inspection of any event property on any TuCSoN event from within any ReSpecT reaction, according to the event model in TABLE I. In the particular scenario depicted by ReSpecT reaction 1.1.1 of Fig. 10, for instance:

event_predicate_1(Term p) — makes it possible to inspect the ⟨*Activity*⟩ | ⟨*Change*⟩ field of the event which directly caused (⟨*Cause*⟩) the triggering of the ReSpecT reaction. In this case, it unifies p with in(sense(temp(T))).

event_source_1(Term s) — makes the ⟨*Source*⟩ of the event observable—that is, who caused event generation. In this case, it unifies s with the ⟨*AgentId*⟩ of the agent issuing the coordination operation (message 1).

event_target_1(Term t) — allows inspection of the ⟨*Target*⟩ field of the event. In this case, it unifies t with the ⟨*TCId*⟩ of the tuple centre target of the event (message 1.1).

event_time_1(Term t) — makes the ⟨*Time*⟩ when the event was generated observable. In this case, it unifies t with the time at which the ACC receives the coordination operation request (message 1).

event_place_1(Term s, Term p) — allows the ⟨*Space:Place*⟩ field of the event to be inspected, once the sort of *space* is chosen from a pre-defined set of admissible spaces—either absolute physical position (s=ph), IP address (s=ip), domain name (s=dns), geographical location (s=map), organisational position (s=org. In this case, it unifies p with, e.g., the network address of the agent which caused reaction triggering.

If the same methods were used in ReSpecT reaction annotating message 1.1.2.3.1, the results would be different—due to situatedness of events:

event_predicate_1(Term p) — would unify p with getEnv(temp, T).

event_source_1(Term s) — would unify s with the ⟨*ProbeId*⟩ of the probe source of the event (message 1.1.2.2).

event_target_1(Term t) — would unify t with the ⟨*TCId*⟩ of the tuple centre target of the event (message 1.1.2.3).

event_time_1(Term t) — would unify t with the time at which the transducer receives the situation

operation completion (message 1.1.2.2).

`event_place_1(Term s, Term p)` — would unify p with, e.g., the network address of the sensor probe that caused reaction triggering.

## V. CONCLUSION

Comparing the methodological approach discussed here with the whole lot of AOSE methodologies available nowadays would be unfeasible for reasons of space. However, it may easily be noted that the only AOSE methodology that clearly resembles our coordination-based approach is SODA[21], in particular of its concern with interaction and environment in MAS. Also, most of the possible remarks can be easily found in [7], where the issues of situatedness and environment engineering in MAS, along with the relationship between agent infrastructure and methodologies, is throughly reviewed.

So, in this paper we introduce the TuCSoN approach to situated coordination in MAS, by describing the support to situated MAS engineering provided by the TuCSoN model and architecture in each typical stage of software development: the abstractions to be used for the requirement analysis step, the architectural components available to model the MAS at hand, the software API to program situatedness-related aspects from a coordination standpoint.

The solutions adopted by the TuCSoN technology to deal with the issues of open and distributed MAS are also highlighted: in particular, the need to rely on mediating abstractions such as ACC, transducers, and tuple centres as the means to uncouple individual components (agents and probes) interactions, along with the need for an asynchronous event-driven communication model to correctly deal with the most common issues of system distribution.

## REFERENCES

[1] J. Ferber and J.-P. Müller, "Influences and reaction: A model of situated multiagent systems," in *2nd International Conference on Multi-Agent Systems (ICMAS-96)*, M. Tokoro, Ed. Tokio, Japan: AAAI Press, Dec. 1996, pp. 72–79. [Online]. Available: http://aaaipress.org/Papers/ICMAS/1996/ICMAS96-009.pdf

[2] 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

[3] L. A. Suchman, "Situated actions," in *Plans and Situated Actions: The Problem of Human-Machine Communication*. New York, NYU, USA: Cambridge University Press, 1987, ch. 4, pp. 49–67.

[4] 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

[5] A. Omicini and S. Mariani, "Coordination for situated MAS: Towards an event-driven architecture," in *International Workshop on Petri Nets and Software Engineering (PNSE'13)*, ser. CEUR Workshop Proceedings, D. Moldt and H. Rölke, Eds., vol. 989. Sun SITE Central Europe, RWTH Aachen University, 6 Aug. 2013, pp. 17–22, joint Proceedings of the International Workshop on Petri Nets and Software Engineering (PNSE'13) and the International Workshop on Modeling and Business Environments (ModBE'13), co-located with the 34th International Conference on Application and Theory of Petri Nets and Concurrency (Petri Nets 2013). Milano, Italy, 24–25 Jun. 2013. Invited paper. [Online]. Available: http://ceur-ws.org/Vol-989/paper00.pdf

[6] 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. [Online]. Available: http://springerlink.metapress.com/content/uk519681t1r38301/

[7] A. Molesini, E. Nardini, E. Denti, and A. Omicini, "Situated process engineering for integrating processes from methodologies to infrastructures," in *24th Annual ACM Symposium on Applied Computing (SAC 2009)*, S. Y. Shin, S. Ossowski, R. Menezes, and M. Viroli, Eds., vol. II. Honolulu, Hawai'i, USA: ACM, 8–12 Mar. 2009, pp. 699–706. [Online]. Available: http://dl.acm.org/citation.cfm?id=1529429

[8] A. Molesini, A. Omicini, and M. Viroli, "Environment in Agent-Oriented Software Engineering methodologies," *Multiagent and Grid Systems*, vol. 5, no. 1, pp. 37–57, 2009, special Issue "Engineering Environments in Multi-Agent Systems". [Online]. Available: http://iospress.metapress.com/content/q38j30vw612rg5g8/

[9] 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://iospress.metapress.com/link.asp?id=5uefwmu39gt3gmvp

[10] 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

[11] 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.

[12] M. Casadei and A. Omicini, "Situated tuple centres in ReSpecT," in *24th Annual ACM Symposium on Applied Computing (SAC 2009)*, S. Y. Shin, S. Ossowski, R. Menezes, and M. Viroli, Eds., vol. III. Honolulu, Hawai'i, USA: ACM, 8–12 Mar. 2009, pp. 1361–1368. [Online]. Available: http://portal.acm.org/citation.cfm?id=1529282.1529586

[13] A. Omicini, "Formal ReSpecT in the A&A perspective," *Electronic Notes in Theoretical Computer Science*, vol. 175, no. 2, pp. 97–117, Jun. 2007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066107003519

[14] S. Mariani and A. Omicini, "Space-aware coordination in ReSpecT," in *From Objects to Agents*, ser. CEUR Workshop Proceedings, M. Baldoni, C. Baroglio, F. Bergenti, and A. Garro, Eds., vol. 1099. Turin, Italy: Sun SITE Central Europe, RWTH Aachen University, 2–3 Dec. 2013, pp. 1–7, xIV Workshop (WOA 2013). Workshop Notes. [Online]. Available: http://ceur-ws.org/Vol-1099/paper3.pdf

[15] A. Ricci, M. Viroli, and A. Omicini, "The A&A programming model and technology for developing agent environments in MAS," in *Programming Multi-Agent Systems*, ser. LNCS, M. Dastani, A. El Fallah Seghrouchni, A. Ricci, and M. Winikoff, Eds. Springer, Apr. 2008, vol. 4908, pp. 89–106. [Online]. Available: http://www.springerlink.com/content/92370q174328841j/

[16] A. Omicini and E. Denti, "From tuple spaces to tuple centres," *Science of Computer Programming*, vol. 41, no. 3, pp. 277–294, Nov. 2001.

[17] M. Viroli, A. Omicini, and A. Ricci, "Infrastructure for RBAC-MAS: An approach based on Agent Coordination Contexts," *Applied Artificial Intelligence: An International Journal*, vol. 21, no. 4–5, pp. 443–467, Apr. 2007, special Issue: State of Applications in AI Research from AI*IA 2005. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1080/08839510701253674

[18] M. Schumacher, *Objective Coordination in Multi-Agent System Engineering. Design and Implementation*, ser. LNCS. Springer, Apr. 2001, vol. 2039. [Online]. Available: http://www.springerlink.com/content/t65dbp4hmj7r/

[19] A. Omicini and S. Ossowski, "Objective versus subjective coordination in the engineering of agent systems," in *Intelligent Information Agents: An AgentLink Perspective*, ser. LNAI: State-of-the-Art Survey, M. Klusch, S. Bergamaschi, P. Edwards, and P. Petta, Eds. Springer, 2003, vol. 2586, pp. 179–202. [Online]. Available: http://www.springerlink.com/content/cvx82e7ej1j9c65n/

[20] A. Omicini, "SODA: Societies and infrastructures in the analysis and design of agent-based systems," in *Agent-Oriented Software Engineering*, ser. Lecture Notes in Computer Science, P. Ciancarini and M. J. Wooldridge, Eds. Springer-Verlag, 2001, vol. 1957, pp. 185–193, 1st International Workshop (AOSE 2000), Limerick, Ireland, 10 Jun. 2000. Revised Papers. [Online]. Available: http://link.springer.com/chapter/10.1007/3-540-44564-1_12