Coordinated Change of State for Situated Agents

Giuseppe Vizzari and Stefania Bandini
Dipartimento di Informatica, Sistemistica e Comunicazione
Università degli Studi di Milano–Bicocca
Via Bicocca degli Arcimboldi 8, 20126 Milano, Italy
{giuseppe.vizzari, bandini}@disco.unimib.it

***Abstract*— Situated Multi Agent System models are characterized by the representation and exploitation of spatial information related to agents, the environment they inhabit and their positions. Specific coordination mechanisms exploiting the contextual spatial information can be defined. In particular this paper will focus on issues and proposed solutions related to the coordinated change of state for situated agents.**

I. Introduction

Agent coordination represents a very active and challenging area of the research in Multi-Agent Systems (MAS). The term coordination refers to the interaction mechanisms that allow autonomous agents to select and carry out their actions within a concerted framework. The separation of the agent computation model, specifying the behaviour of a single agent, from the coordination model is a proposal that goes back to the early nineties [7]. In particular, the concept of Linda tuple space [6] and the related coordination language is the most diffused metaphor adopted by current coordination languages and approaches. The basic model has been enhanced, on one hand at a technical level, in order to allow a distributed implementation of the conceptually centered tuple space [16]. On the other hand, tuple spaces have been also extended in order to allow the specification of tuple-based coordination media presenting reactive and programmable behaviours (see, e.g., [14], [15], [4]), and also allowing the specification and enforcement of organizational abstractions and constraints (e.g. roles, access control rules) to agent coordination [17].

Situated MASs (see, e.g., [1], [10], [20]) are particular agent based models which provide the representation and exploitation of spatial information related to agents and their position into the environment they inhabit. While the previously defined approaches to agent coordination provide general-purpose coordination languages and mechanisms, situated MASs present issues that could benefit from specific mechanisms for agent interaction. For instance, the concept of *field* (i.e. a signal that agents may spread in their environment, which can influence the behaviour of other entities) has been widely adopted for the generation of coordinated movements (see, e.g., [2], [9]). This kind of mechanism is devoted to the interaction of agents which may be positioned on distant points of their space, there can be situations in which agents which are in direct contact (considering a discrete representation of agents' environment) may wish to perform a coordinated change in the respective state (for instance in order to model the exchange of information) without causing modifications in the environment. In fact, field based interaction and other approaches focused on modelling agent environment [19], are intrinsically multicast interaction mechanisms that may be useful to represent actions and interactions that should be *observable* by other entities in the system. However this observability property should not automatically characterize all possible actions and interactions of a Multi Agent model. To this purpose, Multilayered Multi Agent Situated System (MMASS) [1] defines the *reaction* action which allows the coordinated change of the states of agents which are positioned in sites forming a clique (i.e. a complete subgraph) in the spatial structure of their environment. This operation, which also allows a direct exchange of information among the involved entities, is not observable by other agents. The aim of this paper is to describe issues related to coordinated changes in the state of situated agents, and propose approaches for the management of these issues, with specific reference to the reaction MMASS action.

The following section will better describe the problem, showing how existing situated MAS approaches tackle the issue of coordinated agent change of state. Section III will focus on the design and implementation of mechanisms supporting coordinated change of state of situated agents, discussing synchronous and asynchronous cases. Conclusions and future developments will end the paper.

II. Coordinated change of state in situated MASs

Despite most agent definitions emphasize the role of the environment, currently most model do not include it as a first class abstraction. The number of situated MAS models (that are models providing a representation of spatial features of agent environment) is thus relatively small, and the topic of coordinating the change of state of situated agents is still not widely analyzed.

One of the first approaches providing the possibility to define the spatial structure of agents' environment is represented by Swarm [12]. Swarm and platforms derived by it (e.g. Repast[1], Mason [8]) generally provide an explicit representation of the environment in which agents are placed, and often provide mechanisms for the diffusion of signals. Nonetheless they generally represent useful libraries and tools for the implementation of simulations, but do not provide a comprehensive, formally defined *interaction model*. In other words they do not provide support to the coordinated change of state among agents, but just define and implement a spatial

[1] http://repast.sourceforge.net

M. Baldoni, F. De Paoli, A. Martelli, and A. Omicini (Eds.): *WOA 2004 – Dagli Oggetti agli Agenti, Sistemi Complessi e Agenti Razionali*, Torino, Italy, November 29th – December 1st 2004. Pitagora Editrice Bologna. ISBN 88-371-1533-4. http://woa04.unito.it

structure in which agents, and sometimes signals, may be placed. Moreover, they generally provide a sequential execution of agents' behaviours (that are triggered by the environment, which is related to the only thread of execution in the whole system). This approach prevents concurrency issues and allows to obtain compact and efficient simulations even with a very high number of entities. The price of these characteristics is essentially that agents are not provided with a thread of execution of their own (i.e. they have a very limited autonomy and proactiveness), and the execution of their behaviours is sequential (although not necessarily deterministic).

The Co-Fields [10] and the Tuples On The Air (TOTA) middleware [11] provide the definition and implementation of a field based interaction model, which specifically supports this kind of interaction that implies a local modification of agents' environment. However the defined interaction mechanism does not provide the possibility to have a coordinated change of agent state without such a modification.

A different approach to the modelling and implementation of situated MAS [20] instead focuses on the definition of a model for simultaneous agent actions, including centralized and (local) regional synchronization mechanisms for agent coordination. In particular, actions can be independent or interfering among each other; in the latter case, they can be mutually exclusive (*concurrent* actions), requiring a contemporary execution in order to have a successful outcome (*joint* actions), or having a more complex influence among each other (both positive or negative).

The previously introduced MMASS model provides two mechanisms for agent interaction. The first is based on the concept of *field*, that is a signal that may be emitted by agents, and will spread in the environment according to its topology and to specific rules specifying field diffusion functions. These signal may be perceived by agents which will react according to their specific behavioural specification. The model also defines the possibility for having a coordinated change of agent state through the *reaction* operation. The outcome of this joint action depends on three factors:

- agents' *positions*: reacting agents must placed in sites forming a complete subgraph in the spatial structure of the environment;
- agents' *behavioural specifications*: agents must include compatible reaction actions in their behavioural specification;
- agents' *willingness* to perform the joint action: one of the preconditions for the reaction is the agreement among the involved agents.

The following section will discuss issues related to the design and implementation of this operation, but several considerations are of general interest in the development of mechanisms supporting the coordinated change of state for situated agents.

III. Reaction

Reaction is an activity that involves two or more agents that are placed in sites forming a clique (i.e. a complete subgraph) and allows them to change their state in a coordinated way, after they have performed an agreement. The MMASS model does not formally specify what this agreement process consists of, and how the activities related to this process influence agent behaviour. This choice is due to the fact that such an agreement process could be very different in different application domains (e.g. user authentication, transactions). For instance, in some of these situations an agent should block its activities while waiting for the outcome of the agreement process, while in others this would be unnecessary. Especially in a distributed environment this agreement process could bring to possible deadlocks, and in order to better focus this subject, more details on internal mechanisms related to agent, to the environment and its composing parts must be given.

```
begin
turn:=0;
do
    begin
    localContext:=environment.sense(turn);
    nextAction:=actionSelect(localContext);
    outcome:=environment.act(nextAction,turn);
    if outcome<>fail then
        turn:=turn+1;
    end
while(true);
end
```

Fig. 1. Agent behaviour thread in a synchronous situation.

A. Synchronous environments

In synchronous situations a global time step regulates the execution of agents actions; in particular, every agent should be allowed to carry out one action per turn. In order to enforce synchronicity, the management of system time step and agent actions can be delegated to agents' *environment*, that they invoke not only for functional reasons (i.e. perform an action which modifies the environment) but also to maintain system synchronicity (i.e. agent threads are put into a wait condition until the environment signals them that the global system time step has advanced). This proposal assumes that agents are provided with one thread of execution, and also provides that the environment has at least one thread of execution of its own. In fact the environment is responsible for the management of field diffusion (more details on this subject can be found in [3]), other modifications of the environment (as consequences of agents' actions), and to enforce system synchronicity.

In the following, more details on agent and environment activities and threads of execution will be given; the situation that will be considered provides one thread for every agent, and a synchronous system. The described approach is valid both for centralized and for distributed situations; in the latter case one of the sites must be elected as a representative of the whole environment, and interactions with the environment can be implemented through a remote invocation protocol (e.g. RMI or others, according to the chosen implementation platform).

1) Agent behaviour management thread: The sequence of actions performed in the agent behaviour thread is the following:

```
begin turn:=0;
do
    begin
    until(forall i in 1..n, agent_i.actionperformed=true)
        begin
        collect(agent_i,action,agentTurn)
        if agentTurn=turn then
            begin
            manage(agent_i,action, turn);
            agent_i.actionperformed:=true;
            end
        else
            agent_i.wait();
        end
    turn:=turn+1;
    forall i in 1..n
        agent_i.actionperformed:=false;
    notifyAllAgents();
    end
while(true);
end
```

Fig. 2. Environment behaviour thread in a synchronous situation.

```
procedure reactionManagement(agent, action, turn)
    begin
    involvedAgents:=action.getReactionPartners();
    reactingAgents:=new list();
    reactingAgents.add(agent);
    agreed:=true;
    forall agent_i in involvedAgents
        begin
        if agent_i.agreeReaction(involvedAgents) = false then
            begin
            agreed:=false;
            break;
            end
        reactingAgents.add(agent_i);
        end
    if agreed=true then
        forall agent_i in reactingAgents
            agent_i.performReact(turn);
    else
        forall agent_i in reactingAgents
            agent_i.notifyFailure(turn);
    end
```

Fig. 3. Reaction management procedure in a synchronous situation.

- *sense its local context*: in order to understand what are the actions whose preconditions are verified, the agent has to collect information required for action selection, and more precisely:
  - active fields in the site it is positioned on and adjacent ones;
  - agents placed in adjacent sites, and their types;
- *select which action to perform*: according to the action selection strategy specified for the system (or for the specific agent type), the agent must select one action to be performed at that turn (if no action's preconditions are satisfied, the agent will simply skip the turn);
- *perform the selected action*: in order to perform the previously selected action, the agent must notify the environment, because the action provides a modification of agent's local context or even simply to maintain system synchronicity.

The last step in agent behavioural management cycle may cause a suspension of the related thread by the environment. In fact an agent may be trying to perform an action for turn t while other ones still did not perform their actions for turn $t-1$. A pseudo-code specification of agent behavioural thread sequence of activities is shown in Figure 1. Agents must thus keep track of current turn and of the previously performed action. In fact, as will be introduced in the following subsection, system dynamics might require an agent to reconsider its action when it is involved in a reaction process.

2) Environment management thread: The environment, more than just managing information on agents' spatial context, also acts as a monitor in order to handle concurrency issues (e.g. synchronization, agreements among agents). Agents must notify the environment of their actions, and the latter will manage these actions performing modifications to the involved structures (e.g. sites and active fields) related to the following turn. The state of the current one must be preserved, in order to allow its sensing and inspection by agents which still did not act in that turn.

The environment may also put an agent into a *wait* condition, whenever performing its action would break system synchronicity. This wait ends when all agents have performed their action for the current turn, and thus all entities are free to perform actions for the next one. The environment must thus keep track of the actions performed by agents in the current turn, and then notify waiting agents whenever system time advances. More schematically, a pseudo-code description of the environment thread of execution is shown in Figure 2. In particular the `manage` function inspects the specified action (which includes the required preconditions and parameters), checks if it is valid and then calls the appropriate subroutines which effectively perform actions.

The previously introduced sequences require a slight integration to specify how reaction actions are managed. In this case the beginning of an agreement process stops other agent actions until this process is over, either positively (when all other involved agents agreed) or negatively (when the agreement failed). In this way, also system time advancement is stopped until the reaction process is over, preserving system synchronicity.

The reaction is triggered by the agent which first requires the execution of this action to the environment. The latter becomes the leader of the group of involved agents, queries them asking if they agree to take part in the reaction, if an agreement is reached it signals them to change their state, then starts again the normal system behaviour, allowing the advancement of global system time step and thus agent execution. More schematically the environment procedure devoted to the management of reaction is shown in 3. An agent receiving a `notifyFailure` will have a `fail` outcome, and thus will not advance its time step and will start over again its behavioural cycle for the current turn. The `reactionManagement` procedure is one of the specific subroutines invoked by the the environment thread of execution previously shown in Figure 2 through the `manage` function.

3) Examples: A sample scenario illustrating the evolution of a centralized synchronous MMASS system is shown is Figure 4. Scenario (a) provides the presence of a set of agents (Agent-1, ..., Agent-n), which do not require the execution of reaction actions. The system dynamics is the following:

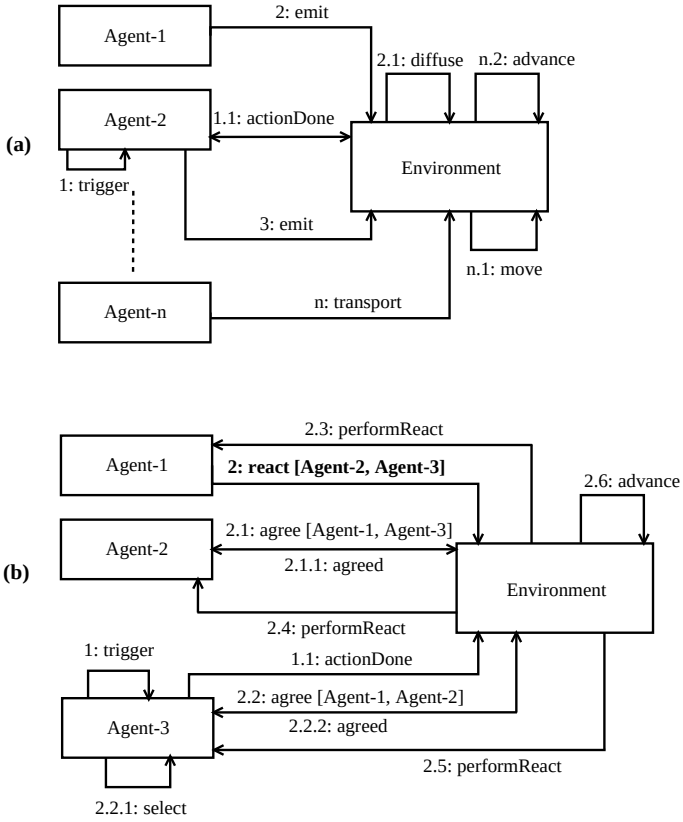

Fig. 4. A sample scenario illustrating the evolution of a centralized synchronous MMASS system. In (a) Agent-2 is put into a wait condition to preserve system synchronicity. In (b) an agreement process for a reaction among Agent-1, Agent-2 and Agent-n is shown.

- Agent-2 performs a trigger (action 1);
- Agent-1 emits a field (action 2) and as a consequence the environment performs its diffusion (action 2.1);
- Agent-2 also tries to perform an emission (action 3), but the environment puts it into a wait condition, as other agents did not perform their actions in that turn;
- agents that are not shown in the Figure perform their actions, which are managed by the environment;
- eventually Agent-n performs a transport action (action n), and as a consequence the environment performs its movement (action n.1), advances system time (action n.2) and eventually notifies agents. Agent-2 emit action (action 3) will now be managed.

A different case is shown in scenario (b), which exemplifies the sequence generated by a reaction request. Agent-1, Agent-2 and Agent-3 are positioned in sites forming a clique. In this case system dynamics is the following:

- Agent-3 performs a trigger (action 1);
- Agent-1 requires the environment to perform a reaction with Agent-2 and Agent-3 (action 2);
- as a consequence to this request, the environment asks Agent-2 if it intends to agree in preforming the reaction (action 2.1) and it receives a positive reply (action 2.1.1);
  the environment then asks Agent-3 if it wishes to reconsider its action for the current turn (action 2.2); the agent performs anew an action selection (action 2.2.1) and decides to agree(action 2.2.1);
- the environment indicates all involved agents that they must perform the reaction (actions 2.3 – 2.5) and then advances system time.

```
begin
do
    begin
    localContext:=mysite.sense();
    nextAction:=actionSelect(localContext);
    outcome:=site.act(nextAction);
while(true);
end
```

Fig. 5. Agent behaviour thread in an asynchronous situation.

4) Discussion: The previously described approach to the management of agents, their cycle of execution, their environment and reaction mechanisms provides a key role of the environment, which represents a sort of medium ensuring specific properties, and especially system synchronicity. This is a global feature of the system, and the simplest way to ensure it is to have a conceptually centralized unit to which all entities must refer in order to perform their actions. This medium and coordination models providing a centralized medium (e.g. a tuple space) seem thus similar, in fact, both provide an indirect interaction among agents and must tackle issues related to the concurrent access to shared resources. The main difference is the fact that, for instance, a Linda tuple space does not provide abstractions for the definition of spatial information (e.g. a topology, an adjacency relation), that should be modelled, represented and implemented. An interesting feature of advanced artifact based interaction models, and more precisely reactive and programmable tuple spaces, is the possibility to specify a behaviour for the artifact, which could be a way to implement interaction mechanisms defined by the MMASS model.

The described approach provides computational costs that could be avoided, in a centralized situation, by providing a single thread of execution, preventing synchronization issues by activating agents in a sequential (although non necessarily deterministic) way (i.e. adopting the approach exploited by Swarm–like simulation platforms). Whenever autonomy and proactiveness are not central elements in agent modelling, this could be a feasible and cost effective choice. It could be the case of simulations characterized by a large number of entities endowed with very simple behavioural specification. However, the described approach can useful when integrating into a single environment entities characterized by a higher degree of autonomy, proactiveness and heterogeneity (for instance, reactive and deliberative agents developed with deeply different approaches).

B. Asynchronous environments

In an asynchronous situation, the mechanisms for the management of agents and their interactions with the environment, are on one hand simpler than in a synchronous case (i.e. there is no need to ensure that every agent acts once per turn), but can also be more complex as there less constraints on action timings. In a centralized situation, it is still possible to delegate

```
begin
do
    begin
    reactionRequest:=mysite.getReactionRequest();
    newReactManager:=new ReactManagerThread(reactionRequest);
    newReactManager.start();
while(true);
end
```

Fig. 6. Agent reaction detection thread in an asynchronous situation.

the management of shared resources to an environment entity, whose task is actually simpler than in a synchronous situation as it does not have to maintain global system synchronicity, although it must guarantee the consistent access to shared resources. In a distributed and asynchronous situation, even if it would be possible to elect a single representative of agents' environment (like in the synchronous and distributed case, described in the previous Section), this possibility would represent a bottleneck and is not even necessary. In fact, the main reason for the presence of a single representative of agent environment was to assure system synchronicity. This Section will then focus on a distributed and asynchronous scenario, and will describe a distributed approach providing the collaboration of *sites*, instead of a single centralized environment, for the management of coordinated change of agents' states.

1) Agent related threads: As previously introduces, agents will now collaborate directly with the sites they are placed on, and their behavioural threads must thus be changed. A pseudo-code formalization of agent behaviour thread in an asynchronous situation is shown in Figure 5.

Another change that can be introduced in the agent is the presence of a distinct thread for the management of reaction requests. In fact the agreement process required by the reaction process can require a certain number of interaction among agents which are placed in computational units spread over a network. This means that a relevant delay may occur from the beginning of an agreement process and its outcome (either positive or negative). Being in an asynchronous situation there is no need to stop agent behavior in order to wait for this process to end. An agent may be provided with three kinds of threads:

- its behavioural thread, which is very similar to the one related to the synchronous situation, and whose structure is shown in Figure 5;
- a thread which is devoted to the detection of reaction requests; this thread is responsible to query the site for pending reaction requests (which may occur concurrently) and start the third kind of thread which will manage the agreement process; a pseudo-code formalization of this thread is shown in Figure 6;
- threads that are devoted to the effective management of the reaction process; a pseudo-code formalization of this thread is shown in Figure 7. This kind of thread must check if the agent effectively agrees to perform the reaction, through the `checkAgreement` invocation (only if it is not the one which actually started the reaction process). This means that first of all the agent must have a react action matching the one specified by the request

```
begin myReactAction:=this.getAction(reactionRequest); if
myReactAction<>null then
    begin
    if reactionRequest.author <> this then
        begin
        agreed:=checkAgreement(reactionRequest);
        site.replyReactReq(reactionRequest, agreed);
        end
    if agreed=true then
        begin
        agreemReached:=site.getReactAgreement(reactionRequest);
        if agreemReached=true then
            this.changeState(myReaction.nextState);
        end
    end
else
    site.replyReactReq(reactionRequest, false);
end
```

Fig. 7. Agent reaction management thread in an asynchronous situation.

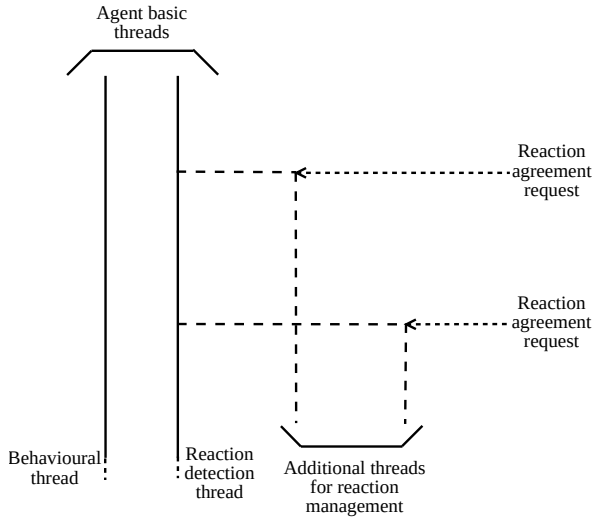

Fig. 8. Threads of execution related to a single MMASS agent in a distributed asynchronous environment.

(this is checked through the `getAction` invocation). Then it must wait the notification of the success or failure of the agreement (the `getReactAgreement` invocation may in fact suspend this thread) and, in the former case, change the agent state.

A diagram showing the three kinds of thread related to a single agent are shown in Figure 8.

2) Site related threads: Similar considerations on the internal structure of agents may be also done for sites. The latter act as a interfaces between agents and the rest of the environment, and must manage events generated both internally and externally. In particular, *internal events* are generated by an agent that is positioned on the site, and more precisely they are the following ones:

- *sense the local context*: the site must provide an agent with the information it needs to select which action it may perform (active fields in the site and adjacent ones, agents in adjacent positions and related types);
- *transport request*: when an agent attempts a transport action, the site it is positioned on must communicate with the destination one in order to verify if it is empty, and eventually allow the agent movement, which frees the

```
procedure reactionManagement(agent, action)
    begin
    involvedAgents:=action.getReactionPartners();
    reactingAgents:=new list();
    reactingAgents.add(agent);
    agreed:=true;
    forall agent_i in involvedAgents
        begin
        adjSite:=agent_i.getSite();
        adjSite.reqAgreement(action);
        end
    until(forall a in involvedAgents, a.gotResponse)
        begin
        if receiveAgreeResp(agent_i,action) = false then
            begin
            agreed:=false;
            break;
            end
        reactingAgents.add(agent_i)
        end
    if agreed=true then
        forall agent_i in reactingAgents
            begin
            adjSite:=agent_i.getSite();
            adjSite.performReact(action);
            end
    else
        forall agent_i in reactingAgents
            begin
            adjSite:=agent_i.getSite();
            adjSite.performReact(adjSite);
            end
    end
```

Fig. 9. Reaction management procedure for the leader site in an asynchronous situation.

current site;

- *reaction request*: upon reception of a reaction request by the overlaying agent, the site must propagate it to involved agents' sites, which in turn will notify them. The site must wait for their replies and then notify all involved entities of the agreement operation outcome; in other words, the site where the reaction is generated is the *leader* of the group of involved sites; a pseudo-code formalization of the reaction management procedure for the leader site is shown in Figure 9;
- *field emission*: when a field is generated in a site it must be added to the set of active fields present in the site, and it must be propagated to other adjacent sites according to the chosen diffusion algorithm.

With reference to reaction, and especially on the selection of a leader site, there are some additional elements that must be integrated with the previous description of site behaviour. In an asynchronous environment, there is the possibility that two agents concurrently start two related reactions. For instance, given three agents A, B and C, placed in sites forming a clique, agent A and Agent B require their respective sites to react among themselves and with agent C. There is not a single site which started the reaction, so a leader must be chosen. Whenever this kind of situation occurs an election protocol must be invoked. The first and probably simplest solution, is to associate a unique identifier related to every site (a very simple way of obtaining it could be the adoption of a combination of the IP address and TCP port related to the site) and assume that the one with the lowest identifier becomes the leader of the reaction group, and others will behave as the reaction request was generated by the leader.

```
procedure reactionManagement(site, action)
    begin
    if this.agent <> null then
        begin
        this.agent.notifyReaction(action);
        agreed:=getReactReply(agent,action);
        site.replyReact(agreed);
        if agreed=true then
            if site.reqAgreement()=true then
                this.agent.setReactAgreement(action,true);
        end
    else
        site.replyReact(false);
    end
```

Fig. 10. Reaction management procedure for non-leader sites in an asynchronous situation.

Externally generated events are consequences of internal events generated by agents in other sites; more precisely they are the following ones:

- *inspect the site*: upon request, the site must provide to adjacent sites information related to active fields and to the presence (or absence) of an agent in it;
- *diffusion propagation*: when a field generated in a different site is propagated to the current one the latter must evaluate its value through the related diffusion function and, if the value is not null, it must propagate the field to other adjacent sites according to the adopted diffusion algorithm;
- *reaction request*: upon reception of a reaction request by the leader of a reaction group, the site must forward it to the overlaying agent, wait for its response and transmit it back to the leader; then it must wait for the outcome of the reaction and notify the overlaying agent; a more schematic description of non-leader sites behavior for management of reaction is shown in Figure 10;
- *transport*: when a remote agent attempts a transport action, the destination site must verify if its state has changed from the previous inspection performed by the agent, and if it is still empty will allow the transport action, blocking subsequent incoming transports.

Site is thus responsible for many concurrent activities; the proposed structure of threads for a site is shown in Figure 11: there are two threads respectively detecting internal and external events, and these two threads generate additional ones in order to effectively manage them.

3) Inter-thread communication: Both agents and sites are provided with a set of threads which must be able to communicate among themselves in a safe and consistent way. For instance, agent reaction management thread in an asynchronous situation communicates to the underlying site by means of a `replyReactRequest` invocation (see Figure 7). The latter performs a write operation on a thread-safe queue, that is a structure with synchronized accessors (observers and modifiers) that may be accessed by site threads but also by the ones related to the agent that is placed on it. The `replyReactRequest` invocation inserts an event in this queue, and notifies threads that were waiting for the generation of events. In this case the thread interested in the agent reply to the reaction request is the one related to the underlying site which effectively manages the agreement process with other

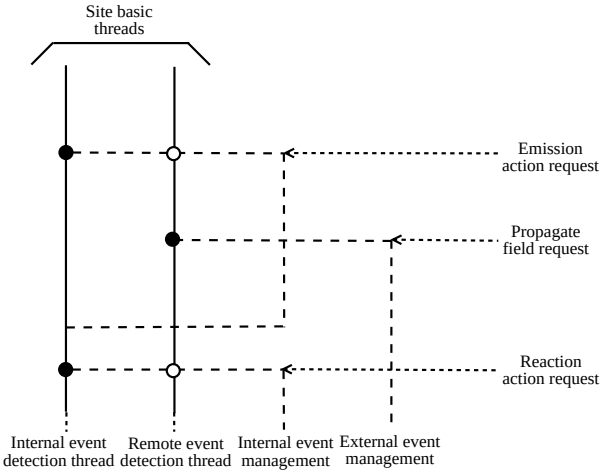

Fig. 11. Threads of execution related to a single MMASS site in a distributed asynchronous environment.

involved entities. It could be either the leader, which is put into a wait condition by the and the `receiveAgreeResp` invocation (see Figure 9), or any other involved site, which is put into a wait condition by a `getReactReply` invocation (see Figure 10).

4) Precautions on network communication: What was still not considered is the possibility to have failures in network transmission, even if to design a robust distributed protocol for reaction management is not the focus of this work. Moreover the chosen technologies supporting network communication could implement mechanisms assuring a reliable form of communication. However, considering the simple loss of messages related to the orchestration of reaction, a simple protocol providing the transmission acknowledgements and the definition of timeouts in order to avoid deadlock situations could be easily implemented. Whenever this kind of issue is detected, the agents' threads related to the management of reaction could simply try to repeat the whole process from the beginning. Moreover, the fact that every agent is related to multiple threads of control, greatly reduces the dangers and issues related to possible deadlocks. In fact, the agent behaviour thread is separated from the management of reactions, and the same can be said for what concerns site specific functions (e.g. threads related to field diffusion are separated from those managing reactions). In this way a failure in a reaction process does not hinder the possibility of the agent to continue its common behaviour, leaving aside the specific reaction that caused the problem. This price of these advantages is that agents and sites are more complex from a computational perspective, and require more resources both in terms of memory and processor time.

There are also some functional requirements that must be considered: the execution of an action during an agreement process might change the preconditions that brought an agent to accept the proposed agreement. In specific cases this could represent a serious issue, and in this case the possibility of the reaction management thread to temporarily block the agent behavioural one should be introduced, suitably exploiting the inter thread interaction mechanism.

5) Discussion: Some of the concurrency issues that were described in this Section are common also in direct agent interaction models. In fact, they are generally designed to work in an asynchronous situation in which messages may be sent and received at any time. In order not to miss any message, the communication partners require some kind of indirection mechanism and structure. For instance, the abstraction of *mailbox* is adopted by Zeus [13], and Jade [18] uses *queues* for managing agent messages. In both cases, specific threads of execution, in addition to those that are related to agents, are adopted to manage communication channels and message exchange.

Unlike the synchronous approach, in this case no single entity managing the coordinated change of state among agents is provided. While managing this kind of operation in a distributed way provides a more complex implementation of sites, to which this activity is delegated, this approach seems more suitable in distributed situations, unless synchronization is absolutely necessary. In fact, a single entity managing this operation may represent a bottleneck and a single point of failure, hindering system robustness.

IV. Conclusions and future developments

The paper has discussed issues related to the coordinated change of state for situated MASs, proposing specific solutions for synchronous and asynchronous situations. In particular, the MMASS reaction action was considered as a specific case of coordinated change of state in situated agents, but several considerations are of general interest in the design and implementation of mechanisms supporting this form of coordinated action in situated MASs. In particular the approach described in [20] provides a similar approach to situated agents coordination: in fact it provides a centralized synchronization, similar to the one provided by the environment described in Section III-A. A distributed mechanism for agent coordination is also described, but it provides a personal synchronizer for every agent while in the approach described in Section III-B every site is responsible for providing this kind of service to the hosted agent.

This work is part of a wider research aimed at the design and development of a platform supporting the development of MMASS based systems. In this framework, another work focused on supporting field diffusion [3], while agent movement will be object of a through analysis: in fact this mechanism requires an attention to functional aspects (e.g. concurrent agents' attempts to move towards the same empty site) and also non-functional ones related to agent mobility in distributed environments. In particular the latter represents a whole area in agent research and software engineering in general (see, e.g., [5]).

References

[1] Stefania Bandini, Sara Manzoni, and Carla Simone, "Heterogeneous agents situated in heterogeneous spaces.," *Applied Artificial Intelligence*, vol. 16, no. 9-10, pp. 831–852, 2002.

[2] Stefania Bandini, Sara Manzoni, and Giuseppe Vizzari, "Situated cellular agents: a model to simulate crowding dynamics," *IEICE Transactions on Information and Systems: Special Issues on Cellular Automata*, vol. E87-D, no. 3, pp. 669–676, 2004.

[3] Stefania Bandini, Sara Manzoni, and Giuseppe Vizzari, "Towards a specification and execution environment for simulations based on mmass: Managing at–a–distance interaction," in *Proceedings of the 17th European Meeting on Cybernetics and Systems Research*, Robert Trappl, Ed. 2004, pp. 636–641, Austrian Society for Cybernetic Studies.

[4] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli, "Mars: a programmable coordination architecture for mobile agents," *IEEE Internet Computing*, vol. 4, no. 4, pp. 26–35, 2000.

[5] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna, "Understanding code mobility," *IEEE Transactions on Software Engineering*, vol. 24, no. 5, pp. 342–361, 1998.

[6] David Gelernter, "Generative communication in linda," *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 1, pp. 80–112, 1985.

[7] David Gelernter and Nicholas Carriero, "Coordination languages and their significance," *Communications of the ACM*, vol. 35, no. 2, pp. 97–107, 1992.

[8] Sean Luke, G. C. Balan, Liviu A. Panait, C. Cioffi-Revilla, and S. Paus, "Mason: a java multi-agent simulation library," in *Proceedings of Agent 2003 Conference on Challenges in Social Simulation*, 2003.

[9] Marco Mamei, Letizia Leonardi, and Franco Zambonelli, "A physically grounded approach to coordinate movements in a team," in *Proceedings of the 1st International Workshop Mobile Teamwork*. 2002, pp. 373–378, IEEE Computer Society.

[10] Marco Mamei, Franco Zambonelli, and Letizia Leonardi, "Co-fields: Towards a unifying approach to the engineering of swarm intelligent systems," in *Engineering Societies in the Agents World III: Third International Workshop (ESAW2002)*. 2002, vol. 2577 of *Lecture Notes in Artificial Intelligence*, pp. 68–81, Springer–Verlag.

[11] Marco Mamei and Franco Zambonelli, "Programming pervasive and mobile computing applications with the tota middleware," in *2nd IEEE International Conference on Pervasive Computing and Communication (Percom2004)*. 2004, pp. 263–273, IEEE Computer Society.

[12] Nelson Minar, Roger Burkhart, Chris Langton, and Manor Askenazi, "The swarm simulation system: A toolkit for building multi-agent simulations," Working Paper 96-06-042, Santa Fe Institute, 1996.

[13] Hyacinth S. Nwana, Divine T. Ndumu, Lyndon C. Lee, and Jaron C. Collis, "Zeus: A toolkit for building distributed multiagent systems," *Applied Artificial Intelligence*, vol. 13, no. 1-2, pp. 129–185, 1999.

[14] Andrea Omicini and Enrico Denti, "From tuple spaces to tuple centres," *Science of Computer Programming*, vol. 41, no. 3, pp. 277–294, 2001.

[15] Andrea Omicini and Franco Zambonelli, "Coordination for Internet application development," *Autonomous Agents and Multi-Agent Systems*, vol. 2, no. 3, pp. 251–269, Sept. 1999, Special Issue: Coordination Mechanisms for Web Agents.

[16] Gian Pietro Picco, Amy L. Murphy, and Gruia-Catalin Roman, "Lime: Linda meets mobility," in *Proceedings of the 21st International Conference on Software Engineering (ICSE99)*. 1999, pp. 368–377, ACM press.

[17] Alessandro Ricci, Mirko Viroli, and Andrea Omicini, "Agent coordination context: From theory to practice," in *Cybernetics and Systems 2004*, Robert Trappl, Ed., Vienna, Austria, 2004, vol. 2, pp. 618–623, Austrian Society for Cybernetic Studies, 17th European Meeting on Cybernetics and Systems Research (EMCSR 2004), Vienna, Austria, 13–16 Apr. 2004. Proceedings.

[18] Giovanni Rimassa, *Runtime Support for Distributed Multi-Agent Systems*, Ph.D. thesis, University of Parma, January 2003.

[19] Luca Tummolini, Cristiano Castelfranchi, Alessandro Ricci, Mirko Viroli, and Andrea Omicini, ""Exhibitionists" and "voyeurs" do it better: A shared environment approach for flexible coordination with tacit messages," in *1st International Workshop on "Environments for MultiAgent Systems" (E4MAS 2004)*, Danny Weyns, H. Van Dyke Parunak, and Fabien Michel, Eds., 2004, pp. 97–111.

[20] Danny Weyns and Tom Holvoet, "Model for simultaneous actions in situated multi-agent systems," in *First International German Conference on Multi-Agent System Technologies, MATES*. 2003, vol. 2831 of *LNCS*, pp. 105–119, Springer–Verlag.