A CASE OF SELF-ORGANISING ENVIRONMENT FOR MAS: THE COLLECTIVE SORT PROBLEM

Matteo Casadei Luca Gardelli Mirko Viroli

Alma Mater Studiorum—Università di Bologna
Via Venezia 52, 47023 Cesena, Italy
{ m.casadei, luca.gardelli, mirko.viroli }@unibo.it

Abstract

Environments of multiagent systems typically feature the application of techniques coming from the research context of complex systems: adaptivity and self-organisation are exploited in order to tackle recurrent issues in multiagent systems applications like openness, dynamism and unpredictability. By adopting the "agents and artifacts" meta-model, we conceived the environment as populated by artifacts: as a specific case, we consider them as information repositories storing relevant facts about the external world and agent interaction/coordination. We focus on the coordination problem called *collective sort*, where autonomous agents in charge of managing such artifacts have the goal of moving information across different artifacts according to local criteria, resulting in the emergence of the complete clustering property. Using a library we developed for the Maude term rewriting system, we simulate the behaviour of this system and evaluate a full solution to this problem.

1 Introduction

Systems that should self-organise to face unpredictable changes in the surrounding conditions often need to achieve adaptivity as an emergent property. As this observation was first made in the context of natural systems, it was shortly recognised as an inspiring metaphor for artificial systems as well [3]. However, a main problem with emergent properties is that, by their very definition, they cannot be achieved through a systematic design: their dynamics and outcomes cannot be fully predicted. Nonetheless, providing some design support in this context is still possible. The whole system of interest—that is, the system application and the environment it is immersed in—can be partially modelled, e.g. as a stochastic system (whose dynamics and duration aspects are probabilistic). In this scenario, simulations can be run and used as a fruitful tool to predict certain aspects of the system behaviour, and to support a correct design before actually implementing the application at hand [6].

This scenario is particularly interesting for the design of MAS environments [20]. Some works like the TOTA middleware [9], the AGV application [21], and stigmergic fields [17], though starting from different perspectives, all develop on the idea of developing MAS environments with coordination techniques featuring adaptivity and self-organisation. They share the idea that information situated into the environment eventually spread to other locations in a non-deterministic way, depending on previous interactions and being affected by timing and probability. Accordingly, in this paper we focus on the role that simulation tools can have in this context, towards the identification of some methodological approach to environment design.

We consider the "agent and artifacts" meta-model for MAS as a reference [19, 16]: this is based on the idea of modelling the agent environment in terms of a set of *artifacts*. Each artifact provides one or more services to agents—differently from agents—without the freedom of autonomy: services are accessed through the artifact interface. Typically, artifacts are used to encapsulate shared repositories of information and knowledge: a simple incarnation of this notion can be the tuple space model or any of its variants [18]—e.g. as provided by TuCSoN [12] or TOTA [9] MAS infrastructures.

In particular, when developing an environment built upon tuple spaces, we recognise tuples distribution among tuple spaces as a main concern: indeed, having tuples sorted out may improve

overall system efficiency. Hence, we get inspiration from the brood sorting problem [3] to evaluate a solution for sorting tuples in a distributed tuple space scenario: we call this problem *collective sort*—as we originally suggested in [4]. This application features autonomous agents managing a closed set of tuple spaces situated in different nodes. These agents have the goal of moving tuples from one space to the other until completely "sorting" them, that is, *(i)* tuples of different types reside in different tuple spaces, and *(ii)* tuples of the same kind reside in the same tuple space.

We show a first solution to this problem considering a simplified scenario: environmental agents, each associated to a particular tuple space S and a particular tuple type T, have the private goal of ensuring that S is not wrongly collecting tuples of the type T. This goal is achieved by moving away tuples which are apparently not contributing to perfect clustering. Despite the locality of this criterion, complete sorting emerges from initial chaotic tuple configurations. In order to obtain full convergence from any initial state, we refine the model introducing the concept of space *vacuum*, leading to the full solution of the collective sort problem.

To devise design choices, and provide evidence of correctness and appropriateness of our approach, we relied on simulations throughout. Many simulation tools can be exploited to this end, though they all necessarily force the designer to exploit a given specification language, and therefore better apply to certain scenarios and not to others—examples are SPiM [14], SWARM [2] and REPAST [1]. Instead of relying on one of them, in this paper we adopted a general-purpose approach we proposed in [4]. We evaluate the applicability of the MAUDE specification tool as a general-purpose engine for running simulations [5]. MAUDE allows for modelling syntactic and dynamic aspects of a system in a quite flexible way, supporting e.g. process algebraic, automata, and net-like specifications—all of which can be seen as instantiations of MAUDE term rewriting framework. We developed a library for allowing a system designer to specify in a custom way a model in terms of a stochastic transition system—a labelled transition system where actions are associated with a *rate* (of occurrence) [15]. One such specification is then exploited by the tool to perform simulations of the system behaviour, thus making it possible to observe the emergence of certain (possibly unexpected) properties.

The remainder of this paper is as follows: Section 2 provides some background on coordination techniques featuring adaptivity, Section 3 briefly introduces our simulation framework, Section 4 describes the collective sort scenario, and finally Section 5 presents the full solution we conceived for this problem.

2 Background

Environment models and technologies for multiagent systems are moving towards the application of self-organising techniques, most of which are inspired by natural phenomena.

A first example is the TOTA (Tuples On The Air) middleware [9] for pervasive computing applications, inspired by the concept of field in physics—like e.g. the gravitational or magnetic fields. This middleware supports the concept of "spatially distributed tuple": that is, a tuple can be cloned and spread to the neighboring tuple spaces, creating a sort of computational field, which grows when initially pumped and then eventually fade. To this end, when injected in a tuple space, each tuple can be equipped with some application-dependent rules, defining how a tuple should spread across the network, how the content of the tuple should be accordingly affected, and so on. TOTA is mainly targeted to support multiagent systems whose environment is open, dynamic and unpredictable, like e.g. to let mobile agents meet each other in a dynamic network.

Another example is the AGV (automatic guided vehicles) application described in [21], where unmanned vehicles controlled by agents transport various kinds of loads through a warehouse. The "virtual environment" (VE) keeps a consistent and updated map of the physical environment, including vehicles' location and status—such as whether they are executing a job. Moreover, it encapsulates important functionalities to support cooperation between agents: e.g. agents can put marks in the VE that propagate in the network and are used to avoid collisions.

A similar mechanism is generalised and adopted in the Digital Pheromone Infrastructure [17], which provides services for injecting and perceiving *digital pheromones* in different sites of the physical/logical distributed environment. Such an environment has the inner ability of aggregating, maintaining and diffusing information according to spatial and temporal criteria—this idea is inspired by stigmergy [13], similarly also to the work on TOTA co-fields. Applied to the military

context, these features can be exploited for several purposes such as surveillance and patrol, target acquisition, and target tracking. For example, target acquisition can be realised by an agent that keeps injecting a pheromone, with the goal of attracting the target.

In the effort to handle the design of this kind of environments, bridging the gap between abstract model and implementation, it has become very common practice to take into account quantitative aspects like temporal and probabilistic ones, which are necessary in order to tackle systems that should feature emergent properties—see e.g. [14, 6, 10]. This calls for proper analysis and design tools, supporting system development at various levels, from formal specification up to simulations.

3 A Maude Library for Simulation

In [4] we developed a framework in the MAUDE term rewriting language, which is meant to speed up the process of modelling and simulating stochastic systems featuring distributed and interacting systems, like MASs. We here briefly describe some features of this framework—the reader interested in further details should refer to [4].

MAUDE is a high-performance reflective language supporting both equational and rewriting logic, for specifying a wide range of applications [5]. Other than specifying algorithmic aspects through algebraic data types, MAUDE can be used to provide *rewriting laws*—i.e. transition rules—that are typically used to implement a concurrent *rewriting semantics*, and are then able to deal with aspects related to interaction and system evolution. In the course of finding a general simulation tool for stochastic systems, we found MAUDE as a particularly appealing framework, for it allows to directly model a system structure and dynamics, or to prototype a new domain-dependent language to have more expressiveness and compact specifications.

Inspired by the work of Priami on the stochastic π-Calculus [15], we realised in MAUDE a general simulation framework for stochastic systems which does not mandate a specification language as e.g. π-Calculus, but is rather open to any language equipped with a stochastic transition system semantics. In this tool a system is modelled as a LTS (labelled transition system) where transitions are of the kind $S \xrightarrow{r:a} S'$, meaning that the system in state S can move to state S' by action a, where r is the *(global) rate* of action a in state S. The rate of an action in a given state can be understood as the number of times action a could occur in a time-unit (if the system would rest in state S), namely, its occurrence frequency. This idea generalises the activity mechanism of stochastic π-Calculus, where each channel is given a fixed local rate, and the global rate of an interaction is computed as the channel rate multiplied by the number of processes willing to send a message and the number of processes willing to receive a message. Our model is hence a generalisation, in that the way the global rate is computed is custom, and ultimately depends on the application at hand—e.g. the global rate can be fixed, or can depend on the number of system sub-processes willing to execute the action.

Given a transition system of this kind and an initial state, a simulation is simply executed by: *(i)* checking each time the available actions and their rate; *(ii)* picking one of them probabilistically (the higher the rate, the more likely the action should occur); *(iii)* accordingly changing the system state; and finally *(iv)* advancing the time counter according to an exponential distribution, so that the average frequency is the sum of the action rates.

As an example, we consider the $Na - Cl$ chemical reaction dynamics, provided e.g. in SPiM documentation[1]. Syntax and semantics of this system is expressed in our MAUDE library as follows:

```
op <_,_,_,_> : Nat Nat Nat Nat -> State .

vars Na Na+ Cl Cl- : Nat .
eq < Na,Na+,Cl,Cl- > ==> =
   ( ion   # (float(Na * Cl) * 1.0)   ->
                [< p Na,s Na+,p Cl,s Cl- >]  );
   ( deion # (float(Na+ * Cl-) * 2.0) ->
                [< s Na,p Na+,s Cl,p Cl- >]  ) .
```

This system is characterised by a state of the kind `<Na,Na+,Cl,Cl->`, where `Na` is the number of sodium atoms, `Na+` the number of sodium ions, `Cl` is the number of chlorine atoms, `Cl-` the number

[1] `http://www.doc.ic.ac.uk/~anp/spim/Chemical.pdf`

of chlorine ions. Two kinds of constant actions are then defined: `ion` stands for ionization and `deion` for deionization. Finally, the transition system is expressed by a single equation, associating to any state two possible effects: one in which ionization decrements `Na` and `Cl` (by prefix predecessor function `p`) and increments `Na+` and `Cl-` (by prefix successor function `s`), and the other that behaves in the opposite way. Note that, according e.g. to the Gillespie selection algorithm in [7], the rate of ionization and deionization is here proportional to the product of the two reactants, multiplied by a constant value: we here enforce deionization factor as being twice that of ionization. By a command of the kind

```
rewrite [300: <100,0,100,0> @ 0.0]
```

the system yields a trace of 300 steps, starting from state `<100,0,100,0>` and elapsed time `0.0`; an example of such trace is:

```
[300 : < 100,0,100,0 > @ 0.0],
[299 : < 99,1,99,1 > @ 5.2282294378567067e-5],
[298 : < 98,2,98,2 > @ 6.9551290710937174e-5],
[297 : < 97,3,97,3 > @ 8.5491215950091466e-5],
...
[3 : < 57,43,57,43 > @ 4.0424914101137542e-2],
[2 : < 58,42,58,42 > @ 4.0506028901053114e-2],
[1 : < 59,41,59,41 > @ 4.0661029058233995e-2],
[0 : < 60,40,60,40 > @ 4.0695684943167353e-2]
```

This output can be easily exploited to trace charts of the most relevant quantities for the application at hand, as shown in the following.

4 Collective Sort

4.1 General Scenario

We consider a case inspired by the Swarm Intelligence problem known as *brood sorting* [3]. It features a multiagent system where the environment is distributed and populated with items of different kinds: the goal of agents is to collect and move items across the environment so as to collect and order them according to an arbitrary shared criterion. This problem basically amounts to clustering: homogeneous items should be grouped together and should be separated from others. Moving to a typical context of MAS software environments, we consider the case of a fixed number of artifacts resembling tuple spaces, hosting tuples of a known set of tuple types. The goal of agents is to move tuples from one tuple space to the other until they are clustered in different tuple spaces according to their type: we call this problem *collective sort*. Note that the agents are not aware of their *cooperative attitude* since it is implicit in their goal, and they do not have to interact nor need to perceive other agents.

In several scenarios, sorting tuples may increase the overall system efficiency. For instance, it can make it easier for an agent to find an information of interest based on its previous experience: the probability of finding information where a previous and related one was found is high. Moreover, when tuple spaces contain tuples of one kind only, it is possible to apply aggregation techniques to improve their performance, and it is generally easier to manage and achieve load-balancing.

Increasing system order however comes at a computational price. Achieving ordering is a task that should be generally performed online and in background, i.e. while the system is running and without adding a significant overhead to the main system functionalities. Indeed, it might be interesting to look for suboptimum algorithms, which are able to guarantee a certain degree of ordering in time.

Nature is a rich source of simple but robust strategies; the behaviour we are looking for has already been explored in the domain of social insects—the aforementioned brood sorting. Ants perform similar tasks when organising broods and larvae: although their actual behaviour is still not fully understood, there are several models that are able to mimic the dynamics of the system. Ants wander randomly and their behaviour is modelled by two probabilities, respectively, the

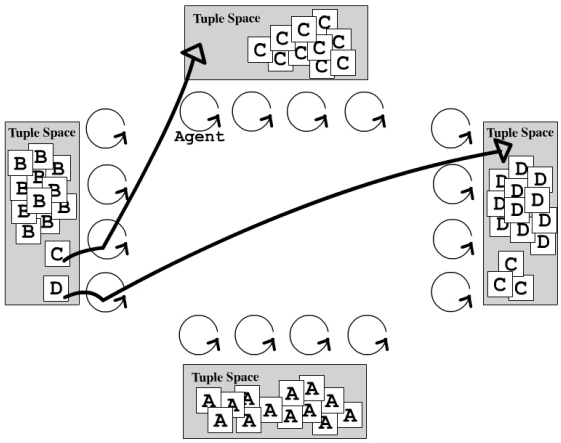


Figure 1: Architecture for collective sort

probability P_p to pick up and P_d drop an item

$$P_p = \left(\frac{k_1}{k_1 + f}\right)^2, \quad P_d = \left(\frac{f}{k_2 + f}\right)^2, \tag{1}$$

where k_1 and k_2 are constant parameters and f is the number of items perceived by an ant in its neighborhood: f may be evaluated with respect to the recently encountered items. To evaluate the system dynamics it can be useful to provide a measure of the system order. Such an estimation can be obtained by measuring the spatial entropy, as done e.g. in [8]. Basically, the environment is subdivided into nodes and P_i is the fraction of items within a node, hence the local entropy is $H_i = -P_i \log P_i$. The sum of H_i having $P_i > 0$ gives an estimation of the order of the entire system, which is supposed to decrease in time, hopefully reaching zero (complete clustering).

4.2 An Architecture for Implementing Collective Sort

We conceive a multiagent system where a collection of agents are *users* of some tuple spaces, forming their environment, and interact with/via them to achieve social goals: in particular, agents are allowed to read, insert and remove tuples in the tuple spaces. Additionally, and transparently to these agents, some further infrastructural support exists that provides tuple spaces with a sorting service, in order to maintain a certain degree of order of tuples in tuple spaces. This service is realised by a class of *collector* agents, spread in the nodes of the distributed environment along with tuple spaces, and which will be responsible for the sorting task.

We suppose that tuples belong to a well defined and globally shared set of tuple kinds. Each collector agent is implicitly associated to the tuple space S of the node where it is situated. For a specific single tuple kind K, the individual agent's goal is to make sure that S is not wrongly collecting tuples of the kind K. Since we look for collecting a tuple kind in only one tuple space, this means that the agent should check that there is seemingly no other tuple space R collecting tuples K more than what S is currently doing, in which case some tuple K has to be moved to R. This scenario is depicted in Figure 1: an agent is moving a tuple of kind `C` from the left tuple space to the top one, since the latter is collecting tuples `C` more—and similarly for `D`. Since we want to perform this task online and in background, and with a fully-distributed, swarm-like algorithm, agents cannot compute the probabilities in Equation 1 to decide whether to move or not a tuple: the approach would not be scalable since it requires to count all the tuples for each tuple space, which might not be practical.

In general, a new primitive to access tuple spaces is needed which *(i)* can feature the locality character, *(ii)* can take into account the different quantities of tuples occurring in the space at a given time, and *(iii)* can possibly fit the semantics of some existing tuple space primitive—so as not to impose a too severe semantic change. A reading primitive `urd` called *uniform read* is hence introduced. This is a variant of the standard `rd` primitive that takes a tuple template and yields any tuple matching the template: primitive `urd` instead chooses the tuple in a probabilistic way

```
[5000 : < 0 @ ('a[100]) | ('b[100]) | ('c[10]) | ('d[10]) > |
       < 1 @ ('a[0]) | ('b[100]) | ('c[10]) | ('d[10]) > |
       < 2 @ ('a[10]) | ('b[50]) | ('c[50]) | ('d[10]) > |
       < 3 @ ('a[50]) | ('b[10]) | ('c[10]) | ('d[50]) >
 @ 0.0],
[4000 : < 0 @ ('a[107]) | ('b[89]) | ('c[0]) | ('d[0]) > |
       < 1 @ ('a[0]) | ('b[136]) | ('c[0]) | ('d[0]) > |
       < 2 @ ('a[0]) | ('b[35]) | ('c[80]) | ('d[0]) > |
       < 3 @ ('a[53]) | ('b[0]) | ('c[0]) | ('d[80]) >
 @ 9.7664497212663287e+2],
...
[2000 : < 0 @ ('a[127]) | ('b[50]) | ('c[0]) | ('d[0]) > |
       < 1 @ ('a[0]) | ('b[210]) | ('c[0]) | ('d[0]) > |
       < 2 @ ('a[0]) | ('b[0]) | ('c[80]) | ('d[0]) > |
       < 3 @ ('a[33]) | ('b[0]) | ('c[0]) | ('d[80]) >
 @ 3.0679938546387184e+3],
...
[1000 : < 0 @ ('a[142]) | ('b[18]) | ('c[0]) | ('d[0]) > |
       < 1 @ ('a[0]) | ('b[242]) | ('c[0]) | ('d[0]) > |
       < 2 @ ('a[0]) | ('b[0]) | ('c[80]) | ('d[0]) > |
       < 3 @ ('a[18]) | ('b[0]) | ('c[0]) | ('d[80]) >
 @ 4.0271359303450395e+3],
...
[438 :  < 0 @ ('a[160]) | ('b[0]) | ('c[0]) | ('d[0]) > |
       < 1 @ ('a[0]) | ('b[260]) | ('c[0]) | ('d[0]) > |
       < 2 @ ('a[0]) | ('b[0]) | ('c[80]) | ('d[0]) > |
       < 3 @ ('a[0]) | ('b[0]) | ('c[0]) | ('d[80]) >
 @ 4.6001450653146167e+3],
...
[0 :    < 0 @ ('a[160]) | ('b[0]) | ('c[0]) | ('d[0]) > |
       < 1 @ ('a[0]) | ('b[260]) | ('c[0]) | ('d[0]) > |
       < 2 @ ('a[0]) | ('b[0]) | ('c[80]) | ('d[0]) > |
       < 3 @ ('a[0]) | ('b[0]) | ('c[0]) | ('d[80]) >
 @ 5.0313233386068514e+3]
```

Figure 2: A trace of simulation for the Collective Sort

among all the tuples that could be returned. For instance, if a tuple space has 10 copies of tuple $t(1)$ and 20 copies of tuple $t(2)$, then the probability that operation $urd(t(X))$—where X is a logic variable—returns $t(2)$ is twice as much as $t(1)$'s. As standard tuple space models typically do not directly implement this variant, it can e.g. be easily supported by some more expressive model like ReSpecT tuple centres [11]—but we shall not deepen this implementation aspect in the following.

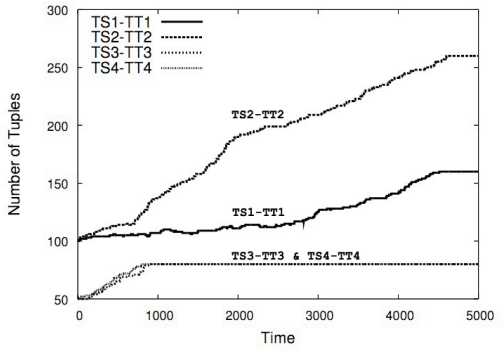
Going back to our collective sort scenario, each agent has now the ability of performing a uniform read over all tuple kinds of a tuple space: in general, if a tuple of kind K is (uniformly) read, the agent can locally assume that, probabilistically, the tuple kind K is aggregating there more than others.

4.3 A Prototype Solution to the Problem

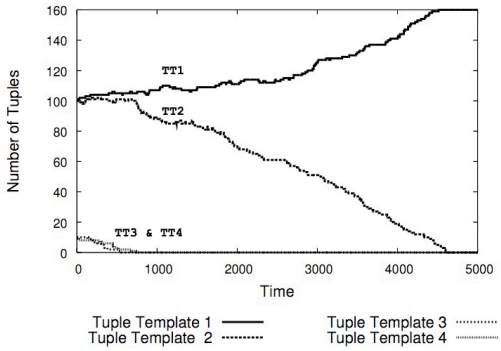
Given this general architecture for modelling the collective sort problem, providing a self-organising solution here means to identify the agents' agenda—sequence of tasks—that could lead each agent to achieve its individual goal. For one such solution to be adequate, complete sorting has to be achieved as an *emergent* property of the overall system, form possibly any initial configuration of tuples.

Each agent is associated to its source tuple space S, seeks for moving tuples of kind K, and works at a given rate r—the frequency at which each agent schedules the execution of the agenda. The agent agenda we consider is as follows:

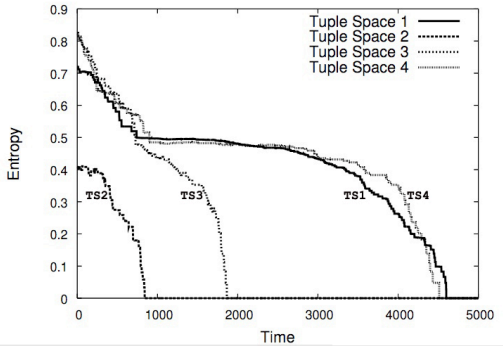
1. a destination tuple space $D \neq S$ is drawn randomly;



(a) Dynamics of the winning tuple in each tuple space



(b) Dynamics of tuple space 0



(c) Entropy of tuple spaces

Figure 3: Charts for a simulation of the Collective Sort

2. a `urd` is performed on S, yielding tuple kind K_S;
3. a `urd` is performed on D, yielding tuple kind K_D;
4. if $K = K_D \neq K_S$, then a tuple of kind K is moved from S to D.

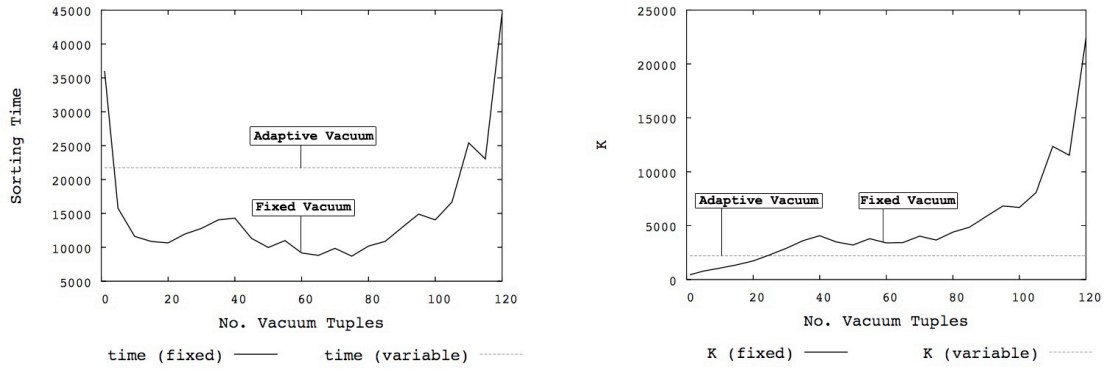
At the time the first task is executed, the agent is focussing on whether one tuple of kind K has to be moved from S to D. At the third task, the agent perceives kind K_S as the one aggregating most in S, and K_D as the one aggregating most in D. Then, the point of last task is that a tuple of kind K is to be moved if and only if K is aggregating in D but not in S.

The success of this distributed algorithm is clearly affected by both probability and timing aspects. Will complete ordering be reached starting from a completely chaotic situation? Will complete ordering be reached starting from the case where all tuples occur in just one tuple space? And if ordering is reached, how many moving attempts are globally necessary? These are the sort of questions that a designer would like to address at the early stages of design without actually resorting to implementation, and that simulation can help addressing.

4.4 Simulation

We represent the distributed state of a system in MAUDE using a syntax of the kind:

```
[ 'a , 'b , 'c , 'd ] | { 0 , 1 , 2 , 3 } |
< 0 @ ('a[100])| ('b[100]) | ('c[10]) | ('d[10]) > |
< 1 @ ('a[0])  | ('b[100]) | ('c[10]) | ('d[10]) > |
< 2 @ ('a[10]) | ('b[50])  | ('c[50]) | ('d[10]) > |
< 3 @ ('a[50]) | ('b[10])  | ('c[10]) | ('d[50]) >
```



(a) Sorting time

(b) Amount of tuples moved

Figure 4: Influence of vacuum tuples on performance: (a) show the case of fixed amount of vacuum tuples, while (b) the case of adaptation.

meaning that we allow for the tuple kinds `'a`, `'b`, `'c`, and `'d`, and the tuple spaces identified by `0`, `1`, `2`, and `3`. Hence, the content of the tuple space `0` is represented as

```
< 0 @ ('a[100])|('b[100])|('c[10])|('d[10]) >,
```

meaning we have 100 tuples of kind `'a`, 100 of kind `'b`, 10 of `'c`, and 10 of `'d`. The formal definition of the agents' agenda is defined in terms of easy transition rules—which are not reported for the sake of brevity: interested readers can refer to [4] for specifications description[2].

We simulated traces of execution starting from the above state, obtaining e.g. the behaviour shown in Figure 2. After some steps, some tuple starts moving from one space to the others. After 2024 time units, for instance, tuple kind `'c` is already completely collected in tuple space `2`. After 4600 time units, the system converged to complete sorting, as we expected. The chart in Figure 3 (a) reports the dynamics of the winning tuple in each tuple space, showing that complete sorting is reached at different times in each space. The chart in Figure 3 (b) displays instead the evolution of the tuple space `0`: notice that only the tuple kind `'a` aggregates here despite its initial concentration was the same of tuple kind `'b`.

Although it would be possible to make some prediction, we do not know in general which tuple space will host a specific tuple kind at the end of sorting: this is an emergent property of the system and is the very result of the *interaction* of the tuple spaces through the agents! Indeed, the final result is not completely random and the concentration of tuples will evolve in the same direction *most* of the times. It is interesting to analyse the trend of the entropy of each tuple space as a way to estimate the degree of order in the system through a single value: since the strategy we simulate is trying to increase the inner order of the system we expect the entropy to decrease, as actually shown in Figure 3 (c). If we denote with q_{ij} the amount of tuples of the kind i within the tuple space j, n_j the total number of tuples within the tuple space j, and k the number of tuple kinds, then, the entropy associated with the tuple kind i within the tuple space j is

$$H_{ij} = \frac{q_{ij}}{n_j} \log_2 \frac{n_j}{q_{ij}} \tag{2}$$

and it is easy to notice that $0 \leq H_{ij} \leq \frac{1}{k} \log_2 k$. The entropy associated with a single tuple space is then computed as

$$H_j = \frac{\sum_{i=1}^{k} H_{ij}}{\log_2 k} \tag{3}$$

where the division by $\log_2 k$ is introduced in order to obtain $0 \leq H_j \leq 1$.

[2]All the specifications cited in this article, charts and instructions for running simulations are available online at http://www.alice.unibo.it

4.5 On Convergence

Considering the reference configuration of four tuple spaces and four tuple types, we made several simulations varying the initial condition, i.e. tuples distribution among tuple spaces: different configurations, i.e. consisting of a different numbers of tuple spaces and tuple types, will be investigated in future works. Although, it initially seemed that the solution adopted always converged to complete sorting, we later identified certain configurations that do not evolve properly. In particular, there are certain states attracting the system trajectory and having positive entropy, that is, characterised by a non-complete degree of sorting: we call such states *local minima*. In particular, local minima seem not to be related to tuples distribution rather to the absence of tuples: an example of such minimum is the following state

```
< 0 @ ('a[20])  | ('b[0])  | ('c[0])  | ('d[0])  > |
< 1 @ ('a[140]) | ('b[0])  | ('c[0])  | ('d[0])  > |
< 2 @ ('a[0])   | ('b[260])| ('c[0])  | ('d[0])  > |
< 3 @ ('a[0])   | ('b[0])  | ('c[80]) | ('d[80]) >
```

Tuple kind `'a` is the only one aggregating in both spaces `0` and `1`, and at the same time, kinds `'c` and `'d` aggregate both in space `3`. Once the system reaches this state, no agent will ever move a tuple for they all reached their individual goal—in no space a tuple kind can be found that aggregates more than elsewhere. Moreover, such states are attractors—when the system starts in a state sufficiently near to them, it ends up quickly converging to one such local minimum.

The attempt of solving this problem is what lead us to the solution proposed in this article.

5 Solving Collective Sort

5.1 Modelling Vacuum

There are two main reasons why the local minimum analysed above cannot be escaped: *(i)* in spite tuple spaces `0` and `1` host a different number of tuples of kind `'a` they are perceived in the same way by agents, for they both have 100% of tuples `'a`—instead, it would be desirable to consider space `1` as a stronger aggregator—; *(ii)* there is no chance of moving a tuple `'c` or `'d` away from space `3`, for no other space aggregates them at all.

These two issues can actually find a common solution by more carefully analysing the brood sorting problem for social insects. There, an ant takes some brood and releases it where a new place is found where brood has a greater concentration. Such a concentration is expressed as quantity of brood over a unit of space. That is, implicitly the ant compares the amount of brood with that of "vacuum" in the unit of space.

If a similar notion of vacuum would be defined in our collective sort example, and e.g. the same amount of vacuum would exist in each space, that could in principle allow to solve the two issues above. On the one hand, space `0` could be recognised as having "less" tuples `'a` than space `1`—since space `1` occupies less space for vacuum, relatively—and hence movements from space `0` to `1` could be promoted most. On the other hand, some tuples `'c` or `'d` could be moved from space `3` to another one following the reasonable idea that "tuples that are not aggregating much should fill the vacuum elsewhere".

To evaluate this solution, we add to tuple spaces another kind of tuple called `vacuum`, and suppose the quantity of vacuum tuples in each space is the same and is fixed statically since the beginning. The uniform read operation can now possibly yield a vacuum tuple: the more such tuples exist with respect to those to be sorted, the more this event is likely. Then, following the above discussion, we change the agent agenda as follows:

1. a destination tuple space $D \neq S$ is drawn randomly;
2. a `urd` is performed on S, yielding tuple kind K_S;
3. a `urd` is performed on D, yielding tuple kind K_D;
4. if $K = K_D \neq K_S$ a tuple of kind K is moved from S to D.
5. if $K \neq K_S$ and $K_D =$ `vacuum` a tuple of kind K is moved from S to D.

Now both K_S and K_D could be vacuum. Our last task says that if the kind K is not aggregating locally ($K \neq K_S$) and we find significant vacuum in D ($K_D = \texttt{vacuum}$), then we move a tuple of kind K.

We considered the following as starting state:

```
< 0 @ ('a[50])| ('b[0])   | ('c[0])   | ('d[0]) > |
< 1 @ ('a[50])| ('b[0])   | ('c[0])   | ('d[0]) > |
< 2 @ ('a[0]) | ('b[100])| ('c[0])   | ('d[0]) > |
< 3 @ ('a[0]) | ('b[0])   | ('c[100])| ('d[100]) >
```

By simulation we noticed that, independently from the number of vacuum tuples, the system escapes the local minimum reaching complete ordering; But of course such a number can potentially influence effectiveness and efficiency of the solution. Hence, in Figures 4 (a) and (b) we display how the sorting time and the number of tuples moved varies with the number of vacuum tuples in each tuple space. We note the following: *(i)* performance is actually dependent on the number of vacuum tuples; *(ii)* as vacuum tuples tend to be the same as the final number of tuples in a tuple space, i.e. 100%, performance degrades dramatically; *(iii)* sorting time has minima values around 20 and 75 vacuum tuples; *(iv)* the number of tuples moved increases with the vacuum. What we learn from these charts is that on the one hand, this technique brings anyway to convergence, but on the other, good performance is achieved if the number of vacuum tuples is around 20% of the final number of tuples expected in each tuple space. There in fact, we have a good combination of convergence time and resources allocated to sorting (i.e., number of tuples moved).

5.2 Adapting Vacuum

If we require a *truly self-organising* solution, we must devise a solution which works without knowing a priori any information about tuple distribution. Hence, we cannot statically design the number of vacuum tuples to be used in each tuple space: an adaptive technique has to be used to make this number dynamic, namely, to make vacuum adapting to the situation at hand. In principle, here we seek for a solution where vacuum is initially very low—guaranteeing to move tuples in a proper way—and then, when/if we are about to reach a local minimum, agents make vacuum locally increase guaranteeing to leave perilous states.

Informally, the idea is to increase vacuum each time an agent discovers two spaces aggregating the same tuple, and decrease vacuum when a tuple successfully moves towards an aggregator. That is, we add to the above agent agenda the following tasks:

6. if $K = K_D \neq K_S$ drop one vacuum tuple from S
7. if $K = K_D = K_S$ add one vacuum tuple to S

In particular, we start from one vacuum tuple, and make sure that this level is never decreased. The charts in Figure 4 (a) and (b) show how this new technique (horizontal line) compares with the one with fixed vacuum. Namely, the behaviour we obtain has average values of convergence time and tuples moved, staying sufficiently far from divergence and bad performance.

Moreover, further simulations we performed on systems that converge with requiring the vacuum management (as in Section 4), show that our adaptive vacuum management does not significantly impact their performance, namely, it is a mechanism exploited on a by-need basis.

6 Conclusion and Future Works

In this article we focussed on stochastic aspects in the designing of emergent coordination mechanisms, an emergent issue in the research on MAS environments and in related research contexts. The collective sort problem discussed is a paradigmatic one: indeed, it emphasises the typical unpredictability of environment in coordination, e.g. in tuple space scenario, tuples distribution in space. Any attempt to find general mechanisms to achieve global properties about such configurations, will likely issue the same problems we identified in collective sort: complete/partial convergence, performance vs. resource usage, need for adaptive mechanisms. Starting from the

work in [4], where the MAUDE library for simulation is described in detail and the collective sort problem is sketched, in this paper we solved these problems developing a full solution.

Several interesting future works can be pursued:

- In the context of collective sort, we plan to evaluate other load-balancing approaches, optimising the convergence to complete order, the vacuum mechanism, and studying the performance of our solution as tuple configurations change dynamically.
- The library itself is currently a very simple prototype, but we believe it could be improved in several ways and become a very practical simulation tool.
- Another interesting idea would be to apply our library to some existing coordination models like TOTA, and provide the necessary tests for the proposed algorithms.

References

[1] Recursive porous agent simulation toolkit (repast), 2006. Available online at http://repast.sourceforge.net/.

[2] Swarm, 2006. Available online at http://www.swarm.org/.

[3] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems*. Santa Fe Institute Studies in the Sciences of Complexity. Oxford University Press, Inc., 1999.

[4] M. Casadei, L. Gardelli, and M. Viroli. Simulating emergent properties of coordination in Maude: the collective sorting case. In *5th International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA'06)*, pages 57–75, CONCUR 2006, Bonn, Germany, 31 Aug. 2006. University of Málaga, Spain. Proceedings.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude Manual*. Department of Computer Science University of Illinois at Urbana-Champaign, 2.2 edition, December 2005. Version 2.2 is available online at http://maude.cs.uiuc.edu.

[6] L. Gardelli, M. Viroli, and A. Omicini. On the role of simulations in engineering self-organising MAS: The case of an intrusion detection system in TuCSoN. In *Engineering Self-Organising Systems*, volume 3910 of *LNAI*, pages 153–168. Springer, 2006. 3rd International Workshop (ESOA 2005), Utrecht, The Netherlands, 26 July 2005. Revised Selected Papers.

[7] D. T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 81(25):2340–2361, 1977.

[8] H. Gutowitz. Complexity-seeking ants. In *Proceedings of the Third European Conference on Artificial Life*, 1993.

[9] M. Mamei and F. Zambonelli. Programming pervasive and mobile computing applications with the tota middleware. In *Pervasive Computing and Communications, 2004. PerCom 2004. Proceedings of the Second IEEE Annual Conference on*, pages 263– 273. IEEE, March 2004.

[10] R. D. Nicola, D. Latella, and M. Massink. Formal modeling and quantitative analysis of KLAIM-based mobile systems. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 428–435, New York, NY, USA, 2005. ACM Press.

[11] A. Omicini and E. Denti. From tuple spaces to tuple centres. *Science of Computer Programming*, 41(3):277–294, Nov. 2001.

[12] A. Omicini and F. Zambonelli. Coordination for Internet application development. *Autonomous Agents and Multi-Agent Systems*, 2(3):251–269, Sept. 1999.

[13] V. D. Parunak. 'Go To The Ant': Engineering principles from natural agent systems. *Annals of Operations Research*, 75:69–101, 1997.

[14] A. Phillips. The Stochastic Pi Machine (SPiM), 2006. Version 0.042 available online at http://www.doc.ic.ac.uk/~anp/spim/.

[15] C. Priami. Stochastic pi-calculus. *The Computer Journal*, 38(7):578–589, 1995.

[16] A. Ricci, M. Viroli, and A. Omicini. *Construenda est* CArtAgO: Toward an infrastructure for artifacts in MAS. In *Cybernetics and Systems 2006*, volume 2, pages 569–574, Vienna, Austria, 18–21 Apr. 2006. Austrian Society for Cybernetic Studies. 18th European Meeting on Cybernetics and Systems Research (EMCSR 2006), 5th International Symposium "From Agent Theory to Theory Implementation" (AT2AI-5). Proceedings.

[17] J. A. Sauter, R. S. Matthews, H. V. D. Parunak, and S. Brueckner. Performance of digital pheromones for swarming vehicle control. In *4rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005), July 25-29, 2005, Utrecht, The Netherlands*, pages 903–910. ACM, 2005.

[18] M. Viroli, T. Holvoet, A. Ricci, K. Schelfthout, and F. Zambonelli. Infrastructures for the environment of multiagent systems. *Autonomous Agents and Multiagent Systems*, 2006. In Press, available online.

[19] M. Viroli, A. Omicini, and A. Ricci. Engineering MAS environment with artifacts. In *2nd International Workshop "Environments for Multi-Agent Systems" (E4MAS 2005)*, AAMAS 2005, Utrecht, The Netherlands, 26 July 2005.

[20] D. Weyns, H. V. D. Parunak, F. Michel, T. Holvoet, and J. Ferber. Environments for multiagent systems state-of-the-art and research challenges. In *Environments for MultiAgent Systems*, volume 3374 of *LNAI*, pages 1–47. Springer-Verlag, Jan. 2005. 1st International Workshop (E4MAS 2004), New York, NY, USA, 19 July 2004. Revised Selected and Invited Papers.

[21] D. Weyns, K. Schelfthout, T. Holvoet, and T. Lefever. Decentralized control of e'gv transportation systems. In *4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005), July 25-29, 2005, Utrecht, The Netherlands*, pages 67–74. ACM, 2005.