A Self-Organising Solution to the Collective Sort Problem in Distributed Tuple Spaces

Mirko Viroli, Matteo Casadei, Luca Gardelli
DEIS
Alma Mater Studiorum - Università di Bologna
via Venezia 52, 47023 Cesena (FC), Italy
{mirko.viroli,m.casadei,luca.gardelli}@unibo.it

ABSTRACT

Coordination languages and models are recently moving towards the application of techniques coming from the research context of complex systems: adaptivity and self-organisation are exploited in order to tackle typical features of systems to coordinate, such as openness, dynamism and unpredictability. In this paper we focus on a paradigmatic problem we call *collective sort*, where autonomous agents are assigned the task of moving tuples across different tuple spaces with the goal of reaching perfect clustering: tuples of the same kind are to be collected in the same, unique tuple space. We describe a self-organising solution to this problem, where each agent moves tuples according to partial observations, still making complete sorting emerge from any initial tuple configuration.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.11 [**Software Engineering**]: Software Architectures; D.3.3 [**Programming Languages**]: Language Constructs and Features

Keywords

Coordination Models, Tuple Spaces, Self-Organization

1. INTRODUCTION

Systems that should self-organise to unpredictable changes in their environment very often need to feature 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 modelled as a stochastic system, namely, a 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 agent coordination. Some works like the TOTA middleware [8], SwarmLinda [9], and stochastic KLAIM [10], though starting from different perspectives, all develop on the idea of extending standard coordination models with features related to adaptivity and self-organisation. They share the idea that tuples in a tuple space eventually spread to other tuple spaces in a non-deterministic way, depending on certain timing and probability issues. 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 system design.

As a reference example, we consider the brood sorting problem for swarm intelligence [3], and recast it to the context of a distributed tuple space scenario: we call this problem *collective sort*—as originally suggested in [4]. This application features autonomous agents managing a closed set of tuple spaces spread over a distributed system. 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.

As a first step, we show the solution to this problem described in [4], where each agent, associated to a single tuple space, has the private goal of moving away tuples which are apparently not contributing to perfect clustering: in spite of this local criterion, complete sorting appear to emerge from initial chaotic tuple configurations. In particular, the decision of which tuple kind will be aggregated in a given tuple space is not taken a priori, but is an emergent effect of the dynamics of tuple ordering. We observe that this solution does not always converge, since—because of its locality character—it does not handle cases where e.g. two different tuple spaces are aggregating the same kind of tuple and nothing else, that is, cases of non-complete sorting. The problem of this approach is that the concept of *vacuum* is not modelled, though it plays an important role in brood sorting, allowing to express the idea of tuple concentration in a given locality. We tackle this issue by introducing in our scenario *vacuum tuples*, which are special kind of tuples that can locally affect tuple sorting. A self-organising technique is again used to dynamically adapt the number of such vacuum tuples so as to fit the requirements of each tuple space, leading to a final solution where *(i)* the overall performance of sorting is not significantly altered, *(ii)* situations of non-complete-sorting are efficiently escaped leading to convergence from arguably *any* initial configuration.

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 [12], SWARM [2] and REPAST [1]. Instead of relying on one of them, in this paper we adopted a general-purpose approach proposed in [4] and based on the MAUDE rewriting system [5], where we developed a framework for allowing a designer to specify in a custom way a system model in terms of a stochastic transition system—a labelled transition system where actions are associated with a *rate* (of occurrence) [13]. 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 describes the collective sort problem, Section 3 presents the full solution to this problem, and finally Section 4 concludes providing final remarks.

2. COLLECTIVE SORTING

2.1 General Scenario

We consider a case of Swarm-like intelligence known as *brood sorting* [3]. It features a multiagent system where the environment is structured and populated with items of different kinds: the goal of agents is to collect and move items across the environment so as to 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 coordination models and languages, we consider the case of a fixed number of tuple spaces hosting tuples of a known set of *tuple templates.* The goal of agents is to move tuples from one tuple space to the other until the tuples are clustered in different tuple spaces according to their template. We call this problem *collective sort.*

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 an 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 probability to pick up P_p and drop P_d 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, apart from visualising it, 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 [7]. 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).

2.2 An Architecture for Implementing Collective Sorting

We conceive a multiagent system as a collection of agents interacting with/via tuple spaces: agents are allowed to read, insert and remove tuples in the tuple spaces. Additionally, and transparently to the agents, an infrastructure provides 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 agents that will be responsible for the sorting task.

We suppose that tuples belong to a well defined and globally shared set of tuple kinds, which are modelled as disjoint tuple templates $K_1, \ldots K_n$—no tuple matches two different templates. In general, an agent can be seen as associated to a tuple space `S`: given a certain tuple kind $\mathtt{K}_i$—e.g. chosen randomly each time—the agent goal is to evaluate whether some tuple of the kind $\mathtt{Ki}_i$ in tuple space `S` should be moved elsewhere. This scenario is depicted in Figure 1.

Since we want to perform this task online and in background, and with a fully-distributed, swarm-like algorithm, we 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 can *(i)* 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 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 takes a finite number of templates and chooses the tuple in a probabilistic way among all the tuples that match any of such templates. For instance, suppose a tuple space has 3 copies of tuple $a(1)$, 7 copies of tuple $a(2)$ and 20 copies of tuple $b(1)$. Then, operation $urd([a(X), b(X)])$ either returns $a(1)$,

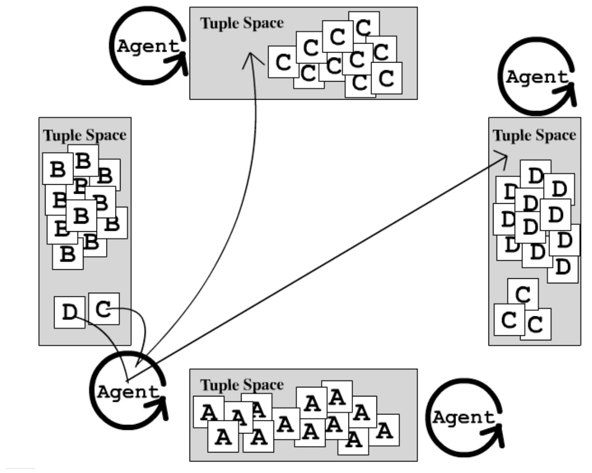


Figure 1: Architecture for collective sort

$a(2)$ or $b(1)$, and furthermore, the probability that it returns a tuple of the kind $b(X)$ is twice as much as $a(X)$'s. This is because the number of tuples matching $b(X)$, that is $b(1)$, is twice as much those matching $a(X)$, which are either $a(2)$ or $a(1)$. Note that this primitive handles probability only at the level of the tuple templates passed as argument: as far as a tuple matching $a(X)$ is returned, nothing is said about the probability this is $a(1)$ or $a(2)$—they are chosen non-deterministically. As standard Linda-like tuple spaces 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.

Thanks to this new primitive, each agent has now also the ability of performing a uniform read over the given templates: in general, if a tuple of kind K is accordingly returned, the agent can locally assume that, probabilistically, the tuple kind K is aggregating there more than others.

2.3 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 agenda of agents, that is, the sequence of tasks each agent locally performs. For one such solution to be adequate, complete sorting has to be achieved as an *emergent* property of the overall system.

Each agent is associated to its source tuple space S, it works at a given rate r, and share with other agents the consensus on the tuple kinds $K_1, \ldots, K_n$ to sort, and the set of tuple spaces hosting them. Rate r is the frequency at which each agent schedules the execution of the agenda—we suppose one such frequency is sufficiently small for an agenda execution being over before the next one starts. The agent agenda we consider is as follows:

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

At the time the second task is executed, the agent is focussing on whether one tuple of kind K has to be moved from S to D. At the fourth 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.

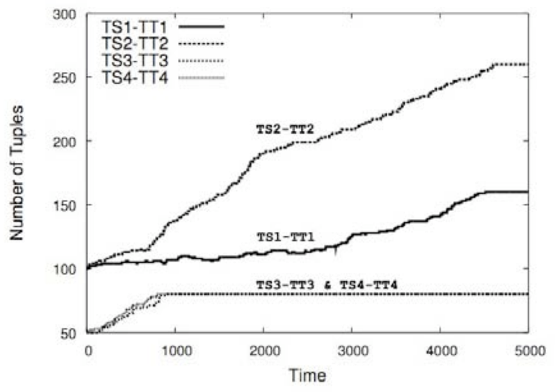
2.4 Simulation

In this section we describe our simulation scenario skipping syntactic and semantic details as implemented in our Maude library—the interested reader should refer to [4]. As first case, we take the following initial tuple configuration.

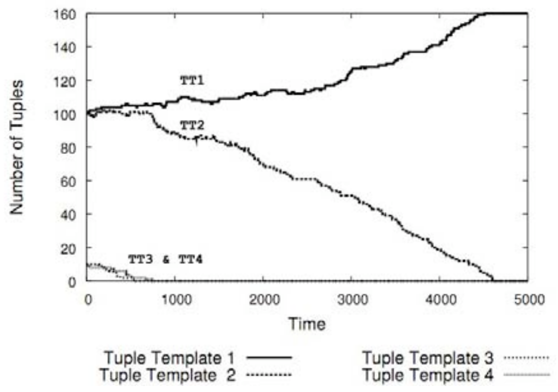
```
< 0 @ (a[100])|(b[100])|(c[10])|(d[10]) > |
< 1 @ (a[0])  |(b[100])|(c[10])|(d[10]) > |
< 2 @ (a[10]) |(b[50]) |(c[50])|(d[10]) > |
< 3 @ (a[50]) |(b[10]) |(c[10])|(d[50]) >
```

It expresses the fact that we work with the tuple kinds `a`, `b`, `c`, and `d` (representing four disjoint tuple templates), and with the tuple spaces identifiers `0`, `1`, `2`, and `3`. The content of tuple space `0` is expressed 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`. Note that this reference scenario provides the same number of tuple kinds and tuple spaces, so that we can seek for a final state where precisely one tuple kind aggregates in one tuple space—studying the more general case is left as future work, as discussed in Section 4. An example of simulation trace is pictorially represented in Figure 2 (a), reporting the dynamics of the winning tuple in each tuple space, showing e.g. that complete sorting is reached at different times in each space. The chart in Figure 2 (b) displays instead the evolution of the tuple space `0`: notice that only tuples of kind `a` aggregates there despite its initial concentration was the same of tuples of kind `b`.

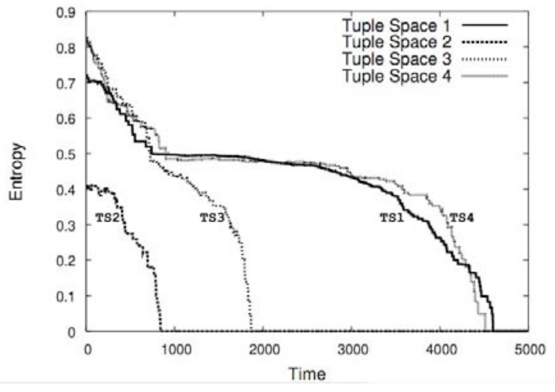
Although it would be possible to make some prediction, we do not know in general which tuple space will host a specific tuple kind at the end of sorting: this is an emergent property of the system and is the very result of the *interaction* of agents through tuple spaces! Indeed, the final result is 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 2 (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



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



(b) Dynamics of tuple space `0`



(c) Entropy of tuple spaces

Figure 2: Charts for a simulation of the Collective Sort

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 $0 \leq H_j \leq 1$.

2.5 On Convergence

It first appeared that this solution always converges to complete sorting from any initial configuration of tuples. However, after some simulation attempts it is actually discovered that there are certain states attracting the system trajectory and having positive entropy, that is, characterised by a non-complete degree of sorting. We call one such state *local minimum*. 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`. It is easy to recognise that once this state is reached, no agent will ever move a tuple, since in no space a tuple kind can be found that aggregates more than elsewhere. Moreover, one such state is an attractor, for simulations starting from states sufficiently near to it appear to converge back to this local minimum. The attempt of solving this problem is what lead us to the solution actually proposed in this paper, as discussed in next section.

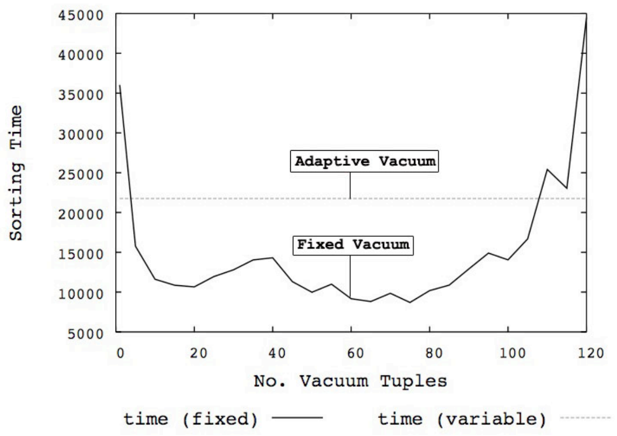
3. SOLVING COLLECTIVE SORT

3.1 Modelling Vacuum

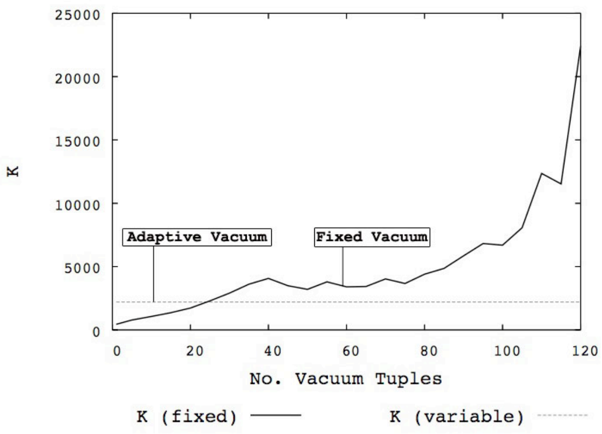
There are two main reasons why the local minimum analysed above cannot be escaped: *(i)* in spite tuple spaces `0` and `1` host a different number of tuples of kind `a` they are perceived in the same way by agents, for they both have 100% of tuples `a`—instead, it would be desirable to consider space `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



(a) Sorting time



(b) Amount of tuples moved

Figure 3: Influence of vacuum tuples on performance (fixed and adaptive vacuum)

`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 initially 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 tuple kind $K \neq$ `vacuum` is drawn randomly among the tuple kinds occurring in S;
3. a `urd` is performed on S, yielding tuple of kind K_S;
4. a `urd` is performed on D, yielding tuple of kind K_D;
5. if $K = K_D \neq K_S$ a tuple of kind K is moved from S to D.
6. 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 ($K_D =$ `vacuum`), then we move a tuple of kind K.

We considered as starting state the following:

```
< 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]) >
```

We noticed that independently from the number of vacuum tuples, the system escapes the local minimum reaching complete ordering, but such a number can potentially influence effectiveness and efficiency of the solution. In Figures 3 (a) and (b) (fixed vacuum) we display how the sorting time and the number of tuples moved varies with the number of vacuum tuples in each tuple space—such number remains fixed throughout the single simulation run. We note the following: *(i)* performance is actually dependent on the number of vacuum tuples; *(ii)* when vacuum tuples tend to equate 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; and *(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). This is also a proper factor for evaluating the network cost of ordering, since the tuple traffic in the network is given by tuples moved and by the number of agent cycles—since each time, an agent performs exactly one remote uniform read.

3.2 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:

7. if $K = K_D \neq K_S$ drop one vacuum tuple from S
8. 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 3 (a) and (b) (adaptive vacuum) also show with the horizontal line how this new technique 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 from bad performance.

Moreover, further simulations we performed on systems that converge without requiring the vacuum management (as in Section 2) show that our adaptive vacuum management does not significantly impact their performance, namely, it is a mechanism exploited on a by-need basis.

4. CONCLUSIONS AND FUTURE WORKS

In this article we focussed on stochastic aspects in the designing of emergent coordination mechanisms—an emergent issue in coordination models and in related research contexts. The collective sort problem discussed is a paradigmatic one, which emphasises the typical unpredictability of environment in coordination, which in tuple space scenarios affect e.g. how tuples are configured in the distributed 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 addressed these problems.

The analysis of the collective sort problem and solution in this paper is of course not complete, for several issues are to be more precisely considered, such as: how the algorithm works when the number of tuple spaces is different from the number of tuple templates, whether the vacuum mechanism can be further improved with respect to performance, how the strategy works when the tuples configuration change dynamically—that is, in a on-line scenario—and so on. Other than this exploration, another interesting future work is to apply our methodological approach to other coordination scenarios, such as e.g. those suggested in the context of SwarmLinda model.

5. 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 C. Canal and M. Viroli, editors, *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 S. A. Brueckner, G. Di Marzo Serugendo, D. Hales, and F. Zambonelli, editors, *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] H. Gutowitz. Complexity-seeking ants. In Deneubourg and Goss, editors, *Proceedings of the Third European Conference on Artificial Life*, 1993.

[8] 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.

[9] R. Menezes and R. Tolksdorf. Adaptiveness in linda-based coordination models. In G. D. M. Serugendo, A. Karageorgos, O. F. Rana, and F. Zambonelli, editors, *Engineering Self-Organising Systems: Nature-Inspired Approaches to Software Engineering*, volume 2977 of *LNAI*, pages 212–232. Springer Berlin / Heidelberg, January 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. Phillips. The Stochastic Pi Machine (SPiM), 2006. Version 0.042 available online at http://www.doc.ic.ac.uk/~anp/spim/.

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